Containerized HPCC Systems® Platform

Boca Raton Documentation Team

Containerized HPCC Systems® Platform

Boca Raton Documentation Team

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com>

Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Version Number in the text of the message.

2026 Version 10.4.10-1

Containerized HPCC Overview

Since version 8.0, the HPCC Systems® Platform started focusing significantly on containerized deployments. This is useful for cloud-based deployments (large or small) or local testing/development deployments.

Docker containers managed by Kubernetes (K8s) is a new target operating environment, alongside continued support for traditional “bare-metal” installations using .deb or .rpm installer files. Support for traditional installers continues and that type of deployment is viable for bare-metal deployments or manual setups in the Cloud.

This is not a *lift and shift* type change, where the platform runs its legacy structure unchanged and treat the containers as just a way of providing *virtual machines* on which to run, but a significant change in how components are configured, how and when they start up, and where they store their data.

This book focuses on these containerized deployments. The first section is about using Docker containers and Helm charts locally. Docker and Helm do a lot of the work for you. The second part uses the same techniques in the cloud.

For local small deployments (for development and testing), we suggest using Docker Desktop and Helm. This is useful for learning, development, and testing.

For Cloud deployments, you can use any flavor of Cloud services, if it supports Docker, Kubernetes, and Helm. This book, however, will focus on Microsoft Azure for Cloud Services.

If you want to manually manage your local or Cloud deployment, you can still use the traditional installers and Configuration Manager, but that removes many of the benefits that Docker, Kubernetes, and Helm provide, such as, instrumentation, monitoring, scaling, and cost control.

HPCC Systems adheres to standard conventions regarding how Kubernetes deployments are normally configured and managed, so it should be easy for someone familiar with Kubernetes and Helm to install and manage the HPCC Systems platform.

Bare-metal vs Containers

If you are familiar with the traditional bare-metal HPCC Systems platform deployments, there are a few fundamental changes to note.

Processes and Pods, not Machines

Anyone familiar with the existing configuration system will know that part of the configuration involves creating instances of each process and specifying on which physical machines they should run.

In a Kubernetes world, this is managed dynamically by the K8s system itself (and can be changed dynamically as the system runs).

Additionally, a containerized system is much simpler to manage if you stick to a one process per container paradigm, where the decisions about which containers need grouping into a pod and which pods can run on which physical nodes, can be made automatically.

Helm Charts

In the containerized world, the information that the operator needs to supply to configure an HPCC Systems environment is greatly reduced. There is no need to specify any information about what machines are in use by what process, as mentioned above, and there is also no need to change a lot of options that might be dependent on the operating environment, since much of that was standardized at the time the container images were built.

Therefore, in most cases, most settings should be left to use the default. As such, the new configuration paradigm requires only the bare minimum of information be specified and any parameters not specified use the appropriate defaults.

The default **environment.xml** that we include in our bare-metal packages to describe the default single-node system contains approximately 1300 lines and it is complex enough that we recommend using a special tool for editing it.

The **values.yaml** from the default Helm chart is relatively small and can be opened in any editor, and/or modified via command-line overrides. It also is self-documented with extensive comments.

Static vs On-Demand Services

In order to realize the potential cost savings of a cloud environment while at the same time taking advantage of the ability to scale up when needed, some services which are always-on in traditional bare-metal installations are launched on-demand in containerized installations.

For example, an eclccserver component launches a stub requiring minimal resources, where the sole task is to watch for workunits submitted for compilation and launch an independent K8s job to perform the actual compile.

Similarly, the eclagent component is also a stub that launches a K8s job when a workunit is submitted and the Thor stub starts up a Thor cluster only when required. Using this design, not only does the capacity of the system automatically scale up to use as many pods as needed to handle the submitted load, it scales down to use minimal resources (as little as a fraction of a single node) during idle times when waiting for jobs to be submitted.

ESP and Dali components are always-on as long as the K8s cluster is started--it isn't feasible to start and stop them on demand without excessive latency. However, ESP can be scaled up and down dynamically to run as many instances needed to handle the current load.

Topology Settings – Clusters vs Queues

In bare-metal deployments, there is a section called **Topology** where the various queues that workunits can be submitted to are set up. It is the responsibility of the person editing the environment to ensure that each named target has the appropriate eclccserver, hThor (or ROXIE) and Thor (if desired) instances set up, to handle workunits submitted to that target queue.

This setup has been greatly simplified when using Helm charts to set up a containerized system. Each named Thor or eclagent component creates a corresponding queue (with the same name) and each eclcc-server listens on all queues by default (but you can restrict to certain queues only if you really want to). Defining a Thor component automatically ensures that the required agent components are provisioned.

Local Deployment (Development and Testing)

While there are many ways to install a local single node HPCC Systems Platform, this section focuses on using Docker Desktop locally.

Prerequisites

Windows	Mac	Linux
• Docker Desktop & WSL 2 • Helm OR • Docker Desktop & Hyper-V • Helm OR • Docker • Kubectl • Helm • Minikube	• Docker Desktop • Helm	• Docker • Helm • Minikube

All third-party tools should be 64-bit versions.

Note: When you install Docker Desktop, it installs Kubernetes and the kubectl command line interface. You merely need to enable it in Docker Desktop settings.

Add a Repository

To use the HPCC Systems Helm chart, you must add it to the Helm repository list, as shown below:

```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

Expected response:

```
"hpcc" has been added to your repositories
```

To update to the latest charts:

```
helm repo update
```

You should update your local repo before any deployment to ensure you have the latest code available.

Expected response:

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "hpcc" chart repository
Update Complete. Happy Helming!
```

Start a Default System

The default Helm chart starts a simple test system with Dali, ESP, ECL CC Server, two ECL Agent queues (ROXIE and hThor mode), and one Thor queue.

To start this simple system:

```
helm install mycluster hpcc/hpcc --version=8.6.14
```

Note: The --version argument is optional, but recommended. It ensures that you know which version you are installing. If omitted, the latest non-development version is installed. This example uses 8.6.14, but you should use the version you want.

Expected response:

```
NAME: mycluster
LAST DEPLOYED: Tue Apr 5 14:45:08 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Thank you for installing the HPCC chart version 8.6.14 using image "hpccsystems/platform-core:8.6.14"
**** WARNING: The configuration contains ephemeral planes: [dali sasha dll data mydropzone debug] ****
This chart has defined the following HPCC components:
dali.mydali
dfuserver.dfuserver
eclagent.hthor
eclagent.roxie-workunit
eclccserver.myeclccserver
eclscheduler.eclscheduler
esp.eclwatch
esp.eclservices
esp.eclqueries
esp.esdl-sandbox
esp.sql2ecl
esp.dfs
roxie.roxie
thor.thor
dali.sasha.coalescer
sasha.dfurecovery-archiver
sasha.dfuwu-archiver
sasha.file-expiry
sasha.wu-archiver
```

Notice the warning about ephemeral planes. This is because this deployment has created temporary, ephemeral storage to use. When the cluster is uninstalled, the storage will no longer exist. This is useful for a quick test, but for more involved work, you will want more persistent storage. This is covered in a later section.

To check status:

```
kubectl get pods
```

Expected response:

```
NAME                                    READY   STATUS    RESTARTS   AGE
eclqueries-7fd94d77cb-m7lmb             1/1     Running   0          2m6s
eclservices-b57f9b7cc-bhwtm             1/1     Running   0          2m6s
eclwatch-599fb7845-2hq54                1/1     Running   0          2m6s
```

```
esdl-sandbox-848b865d46-9bv9r                 1/1     Running   0          2m6s
hthor-745f598795-ql9dl                        1/1     Running   0          2m6s
mydali-6b844bfcfb-jv7f6                       2/2     Running   0          2m6s
myeclccserver-75bcc4d4d-gflfs                 1/1     Running   0          2m6s
roxie-agent-1-77f696466f-tl7bb                1/1     Running   0          2m6s
roxie-agent-1-77f696466f-xzrtf                1/1     Running   0          2m6s
roxie-agent-2-6dd45b7f9d-m22wl                1/1     Running   0          2m6s
roxie-agent-2-6dd45b7f9d-xmlmk                1/1     Running   0          2m6s
roxie-toposerver-695fb9c5c7-9lnp5             1/1     Running   0          2m6s
roxie-workunit-d7446699f-rvf2z                1/1     Running   0          2m6s
sasha-dfurecovery-archiver-78c47c4db7-k9mdz   1/1     Running   0          2m6s
sasha-dfuwu-archiver-576b978cc7-b47v7         1/1     Running   0          2m6s
sasha-file-expiry-8496d87879-xct7f            1/1     Running   0          2m6s
sasha-wu-archiver-5f64594948-xjblh            1/1     Running   0          2m6s
sql2ecl-5c8c94d55-tj4td                       1/1     Running   0          2m6s
dfs-4a9f12621-jabc1                           1/1     Running   0          2m6s
thor-eclagent-6b8f564f9c-qnczz                1/1     Running   0          2m6s
thor-thoragent-56d788869f-7trxk               1/1     Running   0          2m6s
```

Note: It may take a while before all components are running, especially the first time as the container images need to be downloaded from Docker Hub.

Access the Default System

Your system is now ready to use. The usual first step is to open ECL Watch.

Note: Some pages in ECL Watch, such as those displaying topology information, are not yet fully functional in containerized mode.

Use this command to get a list running services and IP addresses:

```
kubectl get svc
```

Expected response:

```
NAME                 TYPE         CLUSTER-IP      EXTERNAL-IP  PORT(S)           AGE
eclqueries           LoadBalancer 10.108.171.35   localhost    8002:31615/TCP    2m6s
eclservices          ClusterIP    10.107.121.158  <none>       8010/TCP          2m6s
eclwatch             LoadBalancer 10.100.81.69    localhost    8010:30173/TCP    2m6s
esdl-sandbox         LoadBalancer 10.100.194.33   localhost    8899:30705/TCP    2m6s
kubernetes           ClusterIP    10.96.0.1       <none>       443/TCP           2m6s
mydali               ClusterIP    10.102.80.158   <none>       7070/TCP          2m6s
roxie                LoadBalancer 10.100.134.125  localhost    9876:30480/TCP    2m6s
roxie-toposerver     ClusterIP    None            <none>       9004/TCP          2m6s
sasha-dfuwu-archiver ClusterIP    10.110.200.110  <none>       8877/TCP          2m6s
sasha-wu-archiver    ClusterIP    10.111.34.240   <none>       8877/TCP          2m6s
sql2ecl              LoadBalancer 10.107.177.180  localhost    8510:30054/TCP    2m6s
dfs                  LoadBalancer 10.100.52.9     localhost    8520:30184/TCP    2m6s
```

Locate the ECL Watch service and identify the EXTERNAL-IP and PORT(S) for eclwatch. In this case, it is localhost:8010.

Open a browser and access ECLWatch, press the ECL button, and select the Playground tab.

From here you can use the example ECL or enter other test queries and pick from the available clusters available to submit your workunits.

Terminate (Decommission) the System

To check which Helm charts are currently installed, run this command:

```
helm list
```

This displays the installed charts and their names. In this example, mycluster.

To stop the HPCC Systems pods, use Helm to uninstall:

```
helm uninstall mycluster
```

This stops the cluster, deletes the pods, and with the default settings and persistent volumes, it also deletes the storage used.

Persistent Storage for a Local Deployment

When running on a single-node test system such as Docker Desktop, the default storage class normally means that all persistent volume claims (PVCs) map to temporary local directories on the host machine. These are typically removed when the cluster is stopped. This is fine for simple testing but for any real application, you want persistent storage.

To persist data with a Docker Desktop deployment, the first step is to make sure the relevant directories exist:

1. Create data directories using a terminal interface:

   For Windows, use this command:

   ```
   mkdir c:\hpccdata
   mkdir c:\hpccdata\dalistorage
   mkdir c:\hpccdata\hpcc-data
   mkdir c:\hpccdata\debug
   mkdir c:\hpccdata\queries
   mkdir c:\hpccdata\sasha
   mkdir c:\hpccdata\dropzone
   ```

   For macOS, use this command:

   ```
   mkdir -p /Users/myUser/hpccdata/{dalistorage,hpcc-data,debug,queries,sasha,dropzone}
   ```

   For Linux, use this command:

   ```
   mkdir -p ~/hpccdata/{dalistorage,hpcc-data,debug,queries,sasha,dropzone}
   ```

   Note: If all of these directories do not exist, your pods may not start.

2. Install the hpcc-localfile Helm chart.

   This chart creates persistent volumes based on host directories you created earlier.

   ```
   # for a WSL2 deployment:
   helm install hpcc-localfile hpcc/hpcc-localfile --set common.hostpath=/run/desktop/mnt/host/c/hpccdata

   # for a Hyper-V deployment:
   helm install hpcc-localfile hpcc/hpcc-localfile --set common.hostpath=/c/hpccdata

   # for a macOS deployment:
   helm install hpcc-localfile hpcc/hpcc-localfile --set common.hostpath=/Users/myUser/hpccdata

   # for a Linux deployment:
   helm install hpcc-localfile hpcc/hpcc-localfile --set common.hostpath=~/hpccdata
   ```

   The **--set common.hostpath=** option specifies the base directory:

   The path **/run/desktop/mnt/host/c/hpccdata** provides access to the host file system for WSL2.

   The path **/c/hpccdata** provides access to the host file system for Hyper-V.

   The path **/Users/myUser/hpccdata** provides access to the host file system for Mac OSX.

   The path **~/hpccdata** provides access to the host file system for Linux.

Note: The value passed to --set common.hostpath is case sensitive.

3. Copy the output from the *helm install* command in the previous step from the word **storage:** to the end, and save it to a text file.

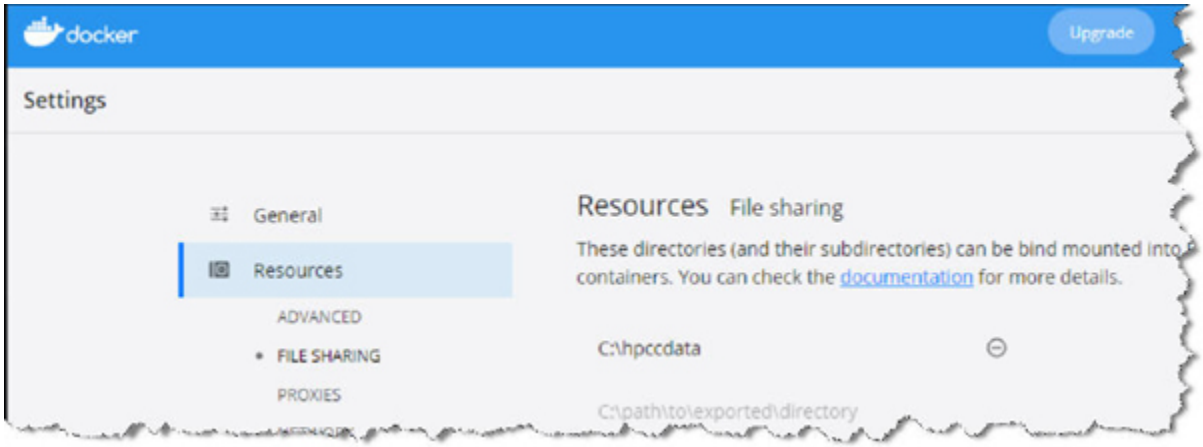
   In this example, we will call the file *mystorage.yaml*. The file should look similar to this:

```
storage:
  planes:
  - name: dali
    pvc: dali-hpcc-localfile-pvc
    prefix: "/var/lib/HPCCSystems/dalistorage"
    category: dali
  - name: dll
    pvc: dll-hpcc-localfile-pvc
    prefix: "/var/lib/HPCCSystems/queries"
    category: dll
  - name: sasha
    pvc: sasha-hpcc-localfile-pvc
    prefix: "/var/lib/HPCCSystems/sasha"
    category: sasha
  - name: debug
    pvc: debug-hpcc-localfile-pvc
    prefix: "/var/lib/HPCCSystems/debug"
    category: debug
  - name: data
    pvc: data-hpcc-localfile-pvc
    prefix: "/var/lib/HPCCSystems/hpcc-data"
    category: data
  - name: mydropzone
    pvc: mydropzone-hpcc-localfile-pvc
    prefix: "/var/lib/HPCCSystems/dropzone"
    category: lz

sasha:
  wu-archiver:
    plane: sasha
  dfuwu-archiver:
    plane: sasha
```

4. If you are using Docker Desktop with Hyper-V, add the shared data folder (in this example, C:\hpccdata) in Docker Desktop's settings by pressing the Add button and typing c:\hpccdata.

   This is **not** needed in a MacOS or WSL 2 environment.



5. Finally, install the hpcc Helm chart, and provide a yaml file that provides the storage information created by the previous step.

```
helm install mycluster hpcc/hpcc --version=8.6.14 -f mystorage.yaml
```

Note: The --version argument is optional, but recommended. It ensures that you know which version you are installing. If omitted, the latest non-development version is installed. This example uses 8.6.14, but you should use the version you want.

6. To test, open a browser and access ECLWatch, press the ECL button, and select the Playground tab, then create some data files and workunits by submitting to Thor some ECL code like the following:

```
LayoutPerson := RECORD
  UNSIGNED1 ID;
  STRING15  FirstName;
  STRING25  LastName;
END;
allPeople := DATASET([ {1,'Fred','Smith'},
                       {2,'Joe','Jones'},
                       {3,'Jane','Smith'}],LayoutPerson);
OUTPUT(allPeople,,'MyData::allPeople',THOR,OVERWRITE);
```

7. Use the *helm uninstall* command to terminate your cluster, then restart your deployment.

8. Open ECL Watch and notice your workunits and logical files are still there.

Using Minikube

To use Minikube make sure the relevant directories exist. These directories are *dalistorage, hpcc-data, debug, queries, sasha, dropzone*, and the parent directory *hpccdata*. If any directory is missing, pods may fail to start.

1. Start the Minikube engine, this example is using Hyper-V as the virtual machine manager.

```
minikube start --vm-driver=hyperv --cpus=4 --memory=1200
```

2. Mount the Windows directory.

   To mount a local Windows directory (e.g., C:\hpccdata) to a directory inside the Minikube VM (/mnt/hpccdata), use the following command:

```
minikube mount --ip 192.168.56.1 "C:\hpccdata:/mnt/hpccdata" --gid=10001 --uid=10000
```

   Use the IP address to bind the mount server (typically your Hyper-V default switch IP).

   The *minikube mount* process must remain running for the directories to stay accessible. Run the command in a separate terminal window, or start it as a background process. If the terminal window running the mount process is closed, the mount will be lost, and your containers will no longer have access to the mounted directory.

3. Install the localfile Helm chart

```
helm install localfile examples/localfile/hpcc-localfile --set common.hostpath=/mnt/hpccdata
```

   Copy the output from the 'helm install' command in this step as detailed in step 3 of the preceding section, and save it to a text file such as *mystorage.yaml.*

4. Finally, install the hpcc Helm chart, and provide the yaml file that provides the storage information created in the previous step such as *mystorage.yaml*.

```
helm install mycluster hpcc/hpcc --version=9.14.2 -f mystorage.yaml
```

Note: The --version argument is optional, but recommended. It ensures that you know which version you are installing. If omitted, the latest non-development version is installed.

Import: Storage Planes and How To Use Them

Storage planes provide the flexibility to configure where the data is stored within a deployed HPCC Systems platform, but it doesn't directly address the question of how to get data onto the platform in the first place.

Containerized platforms support importing data in two ways:

- Upload a file to a Landing Zone and Import (Spray)
- Copy a file to a Storage Plane and access it directly

Beginning with version 7.12.0, new ECL syntax was added to access files directly from a storage plane. This is similar to the **file::** syntax used to directly read files from a physical machine, typically a landing zone.

The new syntax is:

```
~plane::<storage-plane-name>::<path>::<filename>
```

Where the syntax of the path and filename are the same as used with the **file::** syntax. This includes requiring uppercase letters to be quoted with a ^ symbol. For more details, see the Landing Zone Files section of the *ECL Language Reference.*

If you have storage planes configured as in the previous section, and you copy the **originalperson** file to **C:\hpccdata\hpcc-data\tutorial**, you can then reference the file using this syntax:

```
'~plane::hpcc-data::tutorial::originalperson'
```

Note: The **originalperson** file is available from the HPCC Systems Web site:

(https://cdn.hpccsystems.com/install/docs/3_8_0_8rc_CE/OriginalPerson).

Azure Deployment (Development, Testing, and Production)

This section should apply for most Azure subscriptions. You may need to adjust some commands or instructions according to your subscription's requirements.

Using Azure

Though there are many ways to interact with Azure, this section will use the Azure cloud shell command line interface.

The major advantage to using the cloud shell is that it will also have the other prerequisites installed for you.

Azure Prerequisites

To deploy an HPCC Systems containerized platform instance to Azure, you should have:

- A working computer that supports Linux, MacOS, or Windows OS.
- A web browser, such as Chrome or Firefox.
- An Azure account with sufficient permissions, rights, and credentials. To obtain this, please go to www.azure.com or talk to your manager if you believe that your employer might have a corporate account.
- A text editor. You can use one of the editors available in the Azure cloud shell (code, vi, or nano) or any other text editor of your preference.
- At minimum using the 64-bit Helm 3.5 or higher - even if using the Azure cloud shell.

Assuming you have an Azure account with adequate credits, you can make use of Azure's browser-based shell, known as the Azure cloud shell, to deploy and manage your resources. The Azure cloud shell comes with pre-installed tools, such as Helm, Kubectl, Python, Terraform, etc.

```
https://portal.azure.com/
```

If this is your first time accessing the cloud shell, Azure will likely notify you about the need for storage in order to save your virtual machine settings and files.

- Click through the prompts to create your account storage.

You should now be presented with an Azure cloud shell which is ready to use. You can now proceed to the next section.

Third Party Tools

Should you decide not to use the Azure cloud shell, you will need to install and configure the Azure CLI on your host machine in order to deploy and manage Azure resources. In addition, you will also need to install Helm and Kubectl to manage your Kubernetes packages and clusters respectively.

- Azure Client Interface (CLI)
- Kubectl
- Helm 3.5 or greater

All third-party tools listed above should use the 64-bit architecture.

The documentation and instructions for how to install and set up the third party tools are available from the respective vendors on their websites.

Azure Resource Group

An Azure resource group is similar to a folder where a group of related resources are stored. Generally, you should only use one resource group per deployment. For instance, deploying two Kubernetes clusters in one resource group can cause confusion and difficulties to manage. Unless you or someone in your organization has already created a resource group and specified to work in that pre-existing resource group, you will need to create one.

To create a new resource group, you must choose a name and an Azure location. Additionally, you may choose to use tags for ease of management of your resource groups. Some of the details around this may be subject to you or your organization's subscriptions, quotas, restrictions or policies. Please ensure that you have a properly configured Azure subscription with a sufficient access level and credits for a successful deployment.

Run the following command to create a new resource group called rg-hpcc in Azure location eastus:

```
az group create --name rg-hpcc --location eastus
```

The following message indicates that the resource group has been successfully created.

```
{
  "id": "/subscriptions/<my_subscription_id>/resourceGroups/rsg-hpcc",
  "location": "eastus",
  "managedBy": null,
  "name": "rg-hpcc",
  "properties": {
    "provisioningState": "Succeeded"
  },

 "tags": null,
 "type": "Microsoft.Resources/resourceGroups"
 }
```

Please note that the list of regions available to you might vary based on your company's policies and/or location.

Azure Kubernetes Service Cluster

Next we will create an Azure Kubernetes Service (AKS) cluster. AKS stands for Azure Kubernetes Service. It is a service provided by Azure that offers serverless Kubernetes, which promotes rapid delivery, scaling, etc.

You can choose any name for your Kubernetes cluster, we will use aks-hpcc. To create a Kubernetes cluster, run the following command:

```
az aks create --resource-group rg-hpcc --name aks-hpcc --location <location>
```

NOTE There are some optional parameters including `--node-vm-size` and `--node-count`. Node size refers to the specs of your VM of choice while node count refers to the number of VMs you wish to use. In Azure the names VM and node are used interchangeably. For more on node sizes, please visit https://docs.microsoft.com/en-us/azure/virtual-machines/sizes

This step can take a few minutes. The time it takes for Azure to create and provision the requested resources can vary. While you wait, for your deployment to complete, you can view the progress in the Azure portal. To view the progress, open another browser tab to:

```
https://portal.azure.com/#blade/HubsExtension/BrowseAll
```

Azure Node Pools

The Azure Kubernetes Service (AKS) automatically creates one node pool. It is a system node pool, by default. There are two node pool types: *system node pools* and *user node pools*. The system node pool is reserved for core Kubernetes services and workloads, such as kubelets, kube-proxies, etc. A user node pool should be used to host your application services and workloads. Additional node pools can be added after the deployment of the AKS cluster.

To follow the recommendations for reserving the system node pool only for the core AKS services and workloads. You will need to use a node taint on the newly created system node pool. Since you can't add taints to any pre-existing node pool, swap the default system node pool for the newly created one.

In order to do this, enter the following command (all on one line, if possible, and remove the connectors "\" as they are only included here for the code to fit on a single page):

```
az aks nodepool add \
--name sysnodepool \
--cluster-name aks-hpcc \
--resource-group rg-hpcc \
--mode System \
--enable-cluster-autoscaler \
--node-count=2 \
--min-count=1 \
--max-count=2 \
--node-vm-size \
--node-taints CriticalAddonsOnly=true:NoSchedule
```

Delete the automatically created default pool, which we called "nodepool1" as an example, the actual name may vary.

Once again enter the following command on one line, (without connectors "\" if possible).

```
az aks nodepool delete \
--name nodepool1 \
--cluster-name aks-hpcc \
--resource-group rg-hpcc
```

Having at least one user node pool is recommended.

Next add a *user node pool* which will schedule the HPCC Systems pods. Also remember to do so on a single line without the connectors, if possible:

```
az aks nodepool add \
--name usrnodepool1 \
--cluster-name aks-hpcc \
--resource-group rg-hpcc \
--enable-cluster-autoscaler \
--node-count=2 \
--min-count=1 \
--max-count=2 \
--mode User
```

For more information about Azure virtual machine pricing and types, please visit https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

Configure Credentials

To manage your AKS cluster from your host machine and use *kubectl*, you need to authenticate against the cluster. In addition, this will also allow you to deploy your HPCC Systems instance using Helm. To configure the Kubernetes client credentials enter the following command:

```
az aks get-credentials --resource-group rg-hpcc --name aks-hpcc --admin
```

Installing the Helm Charts

This section will demonstrate how to fetch, modify, and deploy the HPCC Systems charts. First we will need to access the HPCC Systems repository.

Add, or update if already installed, the HPCC Systems Helm chart repository:

```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

To update the repository:

```
helm repo update
```

You should always update the repository before deploying. That allows you to get the latest versions of the chart dependencies.

Installing the HPCC Systems Components

In order for a even a basic installation to succeed, it must have some type of storage enabled. The following steps will create ephemeral storage using the *azstorage* utility that will allow the HPCC Systems to start and run but will not persist. To do this we will deploy the *hpcc-azurefile* chart which will set up Azure's ephemeral storage for the HPCC Systems deployment.

To Install the hpcc-azurefile chart:

```
helm install azstorage hpcc/hpcc-azurefile
```

The goal here is to get the default values from this *azstorage* chart and create a customization file that will pass in the appropriate values to the HPCC Systems instance.

Copy the output from the helm install command that you issued in the previous step, from the **storage:** parameter through the end of the file and save the file as *mystorage.yaml*. The *mystorage.yaml* file should look very similar to the following:

```
storage:
  planes:
  - name: dali
    pvc: dali-azstorage-hpcc-azurefile-pvc
    prefix: "/var/lib/HPCCSystems/dalistorage"
    category: dali
  - name: dll
    pvc: dll-azstorage-hpcc-azurefile-pvc
    prefix: "/var/lib/HPCCSystems/queries"
    category: dll
  - name: sasha
    pvc: sasha-azstorage-hpcc-azurefile-pvc
    prefix: "/var/lib/HPCCSystems/sasha"
    category: sasha
  - name: data
    pvc: data-azstorage-hpcc-azurefile-pvc
    prefix: "/var/lib/HPCCSystems/hpcc-data"
    category: data
  - name: mydropzone
    pvc: mydropzone-azstorage-hpcc-azurefile-pvc
    prefix: "/var/lib/HPCCSystems/dropzone"
    category: lz


sasha:
  wu-archiver:
    plane: sasha
  dfuwu-archiver:
    plane: sasha
```

Note: The indentation, syntax, and characters are very critical, please be sure those are an exact match to the above sample. A single extra space in this file can cause unnecessary headaches.

We can now use this *mystorage.yaml* file to pass in these values when we start up our HPCC Systems cluster.

Enable Access the ESP Services

To access your HPCC Systems cloud instance you must enable the visibility of the ESP services. As delivered the ESP services are private with only local visibility. In order to enable global visibility, we will be installing the HPCC Systems cluster using a customization file to override the ESP dictionary. There is more information about customizing your deployment in the *Containerized HPCC Systems* documentation.

The goal here is to get the values from this delivered chart and create a customization file that will pass in the values you want to the HPCC Systems instance. To get the values from that chart, enter the following command:

```
helm show values hpcc/hpcc > defaultvalues.yaml
```

IMPORTANT: The indentation, syntax, characters, as well as every single key-value pair are very critical. Please be sure these are an exact match to the sample below. A single extra space, or missing character in this file can cause unnecessary headaches.

Using the text editor, open the *defaultvalues.yaml* file and copy the **esp:** portion from that file, as illustrated below:

```
esp:
- name: eclwatch
  ## Pre-configured esp applications include eclwatch, eclservices, and eclqueries
  application: eclwatch
  auth: none
  replicas: 1
# Add remote clients to generated client certificates and make the ESP require that one of
r to connect
#   When setting up remote clients make sure that certificates.issuers.remote.enabled is set
# remoteClients:
# - name: myclient
#   organization: mycompany
  service:
    ## port can be used to change the local port used by the pod. If omitted, the default por
    port: 8888
    ## servicePort controls the port that this service will be exposed on, either internally
    servicePort: 8010
    ## Specify visibility: local (or global) if you want the service available from outside
externally, while eclservices is designed for internal use.
    visibility: local
    ## Annotations can be specified on a service - for example to specify provider-specific i
-balancer-internal-subnet
    #annotations:
    #  service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "mysubnet"
    #  The service.annotations prefixed with hpcc.eclwatch.io should not be declared here. T
    #  in other services in order to be exposed in the ECLWatch interface. Similar function c
    #  applications. For other applications, the "eclwatch" inside the service.annotations sh
    #  their application names.
    #  hpcc.eclwatch.io/enabled: "true"
    #  hpcc.eclwatch.io/description: "some description"
    ## You can also specify labels on a service
    #labels:
    #  mylabel: "3"
    ## Links specify the web links for a service. The web links may be shown on ECLWatch.
    #links:
    #- name: linkname
```

```
    #  description: "some description"
    #  url: "http://abc.com/def?g=1"
    ## CIDRS allowed to access this service.
    #loadBalancerSourceRanges: [1.2.3.4/32, 5.6.7.8/32]
  #resources:
  #  cpu: "1"
  #  memory: "2G"
- name: eclservices
  application: eclservices
  auth: none
  replicas: 1
  service:
    servicePort: 8010
    visibility: cluster
  #resources:
  #  cpu: "250m"
  #  memory: "1G"
- name: eclqueries
  application: eclqueries
  auth: none
  replicas: 1
  service:
    visibility: local
    servicePort: 8002
    #annotations:
    #  hpcc.eclwatch.io/enabled: "true"
    #  hpcc.eclwatch.io/description: "Roxie Test page"
    #  hpcc.eclwatch.io/port: "8002"
  #resources:
  #  cpu: "250m"
  #  memory: "1G"
- name: esdl-sandbox
  application: esdl-sandbox
  auth: none
  replicas: 1
  service:
    visibility: local
    servicePort: 8899
  #resources:
  #  cpu: "250m"
  #  memory: "1G"
- name: sql2ecl
  application: sql2ecl
  auth: none
  replicas: 1
# remoteClients:
# - name: sqlclient111
  service:
    visibility: local
    servicePort: 8510
  #domain: hpccsql.com
  #resources:
  #  cpu: "250m"
  #  memory: "1G"
- name: dfs
  application: dfs
  auth: none
  replicas: 1
  service:
    visibility: local
    servicePort: 8520
  #resources:
  #  cpu: "250m"
  #  memory: "1G"
```

Save that ESP portion off into a new file called *myesp.yaml.* You need to modify that file then use it to override those default values into your deployment.

In order to access the HPCC Systems services you must override these default settings to make them visible. We will now set the visibility for *eclwatch* and *eclqueries* from local to global as in the below example.

Edit the *myesp.yaml* file and change the two sections highlighted in the code examples below:

```
esp:
- name: eclwatch
  ## Pre-configured esp applications include eclwatch, eclservices, and eclqueries
  application: eclwatch
  auth: none
  replicas: 1
  service:
    ## port can be used to change the local port used by the pod. If omitted, the default por
    port: 8888
    ## servicePort controls the port that thi cesps service will be exposed on, either intern
    servicePort: 8010
    ## Specify visibility: local (or global) if you want the service available from outside t
externally, while eclservices is designed for internal use.
    visibility: global
    ## Annotations can be specified on a service - for example to specify provider-specific i
```

```
- name: eclqueries
  application: eclqueries
  auth: none
  replicas: 1
  service:
    visibility: global
    servicePort: 8002
```

Save that modified *myesp.yaml* customization file.

We can now use this *myesp.yaml* file to pass in these values when we start up our HPCC Systems cluster.

Install the Customized HPCC Systems Chart

This section will install the delivered HPCC Systems chart where we supply the *myesp.yaml* and *mystorage.yaml* customization files created in the previous section. You should create or add your own additional customizations in one of these or even another customization *yaml* file specific to your requirements. Creating and using customized versions of the HPCC Systems *values.yaml* file are described in the *Customizing Configurations* section of the *Containerized HPCC Systems* docs. To install your customized HPCC Systems charts:

```
helm install myhpcccluster hpcc/hpcc -f myesp.yaml -f mystorage.yaml
```

Where the -f option forces the system to merge in the values set in the *myesp.yaml* and *mystorage.yaml* files.

Note: You can also use the `--values` option as a substitute for `-f`

If successful, your output will be similar to this:

```
NAME: myhpcccluster
LAST DEPLOYED: Wed Dec 15 09:41:38 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

At this point, Kubernetes should start provisioning the HPCC Systems pods. To check their status run:

```
kubectl get pods
```

Note: If this is the first time helm install has been run, it will take some time for the pods to get to a Running state, since Azure will need to pull the container images from Docker. Once all the pods are running, the HPCC Systems Cluster is ready to be used.

Accessing ECLWatch

To access ECLWatch, an external IP to the ESP service running ECLWatch is required. If you successfully deployed your cluster with the proper visibility settings, then this will be listed as the *eclwatch* service. The IP address can be obtained by running the following command:

```
kubectl get svc
```

Your output should be similar to:

```
NAME         TYPE          CLUSTER-IP    EXTERNAL-IP     PORT(S)          AGE
eclservices  ClusterIP     10.0.44.11    <none>          8010/TCP         11m
eclwatch     LoadBalancer  10.0.21.16    12.87.156.228   8010:30190/TCP   11m
kubernetes   ClusterIP     10.0.0.1      <none>          443/TCP          4h28m
mydali       ClusterIP     10.0.195.229  <none>          7070/TCP         11m
```

Use the EXTERNAL-IP address listed for the ECLWatch service. Open a browser and go to http://<external-ip>:8010/. For example in this case, go to http://12.87.156.228:8010. If everything is working as expected, the ECLWatch landing page will be displayed.

Uninstall Your Cluster

When you are done using your HPCC Systems cluster, you may destroy it to avoid incurring charges for unused resources. A storage account is recommended to save your HPCC Systems data outside of the Azure Kubernetes Service. That allows you to destroy the service without losing your data.

The various storage options and strategies are discussed elsewhere in addition to the HPCC Systems documentation.

Stopping Your HPCC Systems Cluster

This will simply stop your HPCC Systems instance. If you are deleting the resource group, as detailed in the following section, that will destroy everything in it, including your HPCC Systems cluster. Uninstalling the HPCC Systems deployment in that case, is redundant. You will still be charged for the AKS. If, for whatever reason, you can't destroy the resource group, then you may follow the steps in this section to shut down your HPCC Systems cluster.

To shut down your HPCC Systems cluster, you would issue the helm uninstall command.

Using the Azure cloud shell, enter:

```
helm list
```

Enter the helm uninstall command using your clusters name as the argument, for example:

```
helm uninstall myhpcccluster
```

This will remove the HPCC Systems cluster named <myhpcccluster> you had previously deployed.

Removing the Resource Group

Removing the resource group will irreversibly destroy any pods, clusters, contents, or any other work stored on there. Please carefully consider these actions, before removing the resource group. Once removed it can not be undone.

To remove the entire resource group *rg-hpcc* which we created earlier, and all the entirety of its contents, issue the following command:

```
az group delete --name rg-hpcc
```

It will prompt you if you are sure you want to do this, and if you confirm it will delete the entire resource group.

Deploying HPCC Systems® with Terraform

Manual deployments can be error-prone and inconsistent. As your deployments become more customized and your need for additional resources grows it can become exponentially more difficult and time consuming.

Fortunately, there are multiple IaC (infrastructure as code) orchestration tools available that can simplify the deployment process. One of those orchestration tools is Terraform. This chapter provides instructions on using Terraform modules to deploy an HPCC Systems instance specifically on the Azure Cloud.

These modules were developed by the HPCC Systems platform team for general open-source community usage. You may require specific customizations for your particular needs. For example, your organization may require opinionated modules for production systems. You can develop your own customized modules, per your requirements and utilize them in the same manner outlined here.

Interactive Terraform Deployment

This section details deploying the containerized HPCC Systems platform onto Azure using Terraform. Using the open source and additional modules from the HPCC Systems Terraform open-source repository. No previous knowledge of Terraform, Kubernetes, or Azure is required.

The steps to deploy an HPCC Systems instance using our provided Terraform modules are detailed in the subsequent sections. A short summary of these steps is as follows.

1. Clone the HPCC Systems Terraform module repository
2. Copy the configuration files (admin.tfvars) from the /examples directory to the corresponding module directory
3. Modify the configuration files for each module
4. Initialize the modules
5. Apply the initialized modules

The strength of using Terraform modules to deploy your system, you only need to set them up once. After they are in place and configured, you can reuse them to stand up an identical instance of your system. You can do so by initializing and then applying them.

Requirements

What you will need in order to deploy an HPCC Systems instance with Terraform:

- A Linux, MacOS, or Windows OS computer system.
- A browser. Such as Chrome or Firefox.
- Git and a Github account that you can access and clone the repository.
- An Azure account with sufficient permissions, rights, credits, and credentials. To obtain one, go to www.azure.com or talk to your manager if you believe that your employer might have a corporate account.

- A code editor of your choice. There are a few editors integrated with Azure such as VS Code, vi the Visual Editor, Nano, or you can choose to use any another.

The easiest option which also ensures you have all the tools required is to use Azure is the command portal. Assuming you have an Azure account with all the appropriate credentials you can just go to the Azure command portal

```
https://portal.azure.com/
```

If this is the first time you have accessed the cloud shell, Azure will prompt you that storage is required for the cloud shell to persist account settings and files. Click through the prompts to create the storage. You should be presented with a shell. At this point, the cloud shell should already be logged into to your Azure account. The major advantage to using the cloud shell is that It will also have other prerequisites installed for you, namely, git, kubectl, helm, and even an editor. Otherwise in addition to installing the Azure CLI you would also need to install the other prerequisites.

Terraform Repository

The HPCC Systems Terraform Repository, *terraform-azurerm-hpcc* is a code repository where the HPCC Systems Terraform modules are stored. The Terraform repository contains three independent modules required to instantiate an HPCC Systems cluster. These are the network, storage, and AKS (Azure Kubernetes Service) modules. The network module must be deployed first, followed by the storage module. Only then can the AKS or root module, be deployed. These modules automatically call other dependent modules upon initialization. There are dependencies which must be in place in order for all the modules to work appropriately.

Cloning the Terraform Repository

Clone the Terraform Repository, *terraform-azurerm-hpcc* hosted on the HPCC Systems GitHub account.

https://github.com/hpcc-systems/terraform-azurerm-hpcc.git

To clone the repository:

1. Open your command line or terminal
2. Determine where to store the repository. Choose a location that is easy to find and remember. This will become the Terraform root directory.
3. Change directory to your chosen location.
4. Run the following command :

```
git clone https://github.com/hpcc-systems/terraform-azurerm-hpcc.git
```

Once the repository is cloned, you will traverse into each module's directory, and configure/modify the admin.tfvars file there, and then apply it.

The Modules to Modify

Once in place these modules can be reused to stand up an exact copy of the instance.

The order of deployment for these Terraform modules is in fact important.

The order of deployment that you must follow is:

1. Virtual network
2. Storage accounts
3. Root module (AKS)

Modules	Location
Virtual network	terraform-azurerm-hpcc/modules/virtual_network
Storage accounts	terraform-azurerm-hpcc/modules/storage_accounts
AKS	terraform-azurerm-hpcc

These modules must be applied in that order since they build on the resources raised by the previous module.

After you clone the terraform-azurerm-hpcc repository you have access to the modules in that repository.

Modify the Modules

First you will copy the configuration file, *admin.tfvars* from the examples subdirectory into that modules directory. Then you will modify that file you just copied. You must repeat this step for each module.

1. Change directory to the virtual network directory first.

```
cd terraform-azurerm-hpcc/modules/virtual_network
```

2. Copy the admin.tfvars files from ./examples to ./virtual_network.

```
cp examples/admin.tfvars ./admin.tfvars
```

To modify the module you can enter the following command (**Note** using the code editor in the example, if you prefer you can use nano, vi, or any text editor):

```
code terraform-azurerm-hpcc/modules/virtual_network/examples/admin.tfvars
```

With the admin.tfvars file open, you can go through each object block or argument and set it to your preferred values.

More information about the module files is available in the *README.md* in the HPCC Systems terraform-azurerm-hpcc repository:

https://github.com/hpcc-systems/terraform-azurerm-hpcc#readme

```
admin = {
  name  = "YourName"

  email = "YourEmail@example.com"
        }

 metadata = {
  project             = "hpccdemo"
  product_name        = "vnet"
  business_unit       = "commercial"
  environment         = "sandbox"
  market              = "us"
  product_group       = "contoso"
  resource_group_type = "app"
  sre_team            = "hpccplatform"
```

```
subscription_type   = "dev"
        }

tags = { "justification" = "testing" }

resource_group = {
  unique_name = true
  location    = "eastus2"
       }
```

1. Modify this file and replace the values for the **name** and **email** fields with your user name and your email address.

2. Save the File as admin.tfvars in the module's directory.

Modifying the AKS Module

The AKS Module is a little different from the other modules. It is not in the modules subdirectory, it is in the base root directory where you previously cloned the Terraform repository. You still need to copy the admin.tfvars file from the examples directory into that root directory, just as you did for the other modules. However, there are a few additional modifications you need to make to this file.

1. Copy the admin.tfvars files from ./examples to the AKS directory.

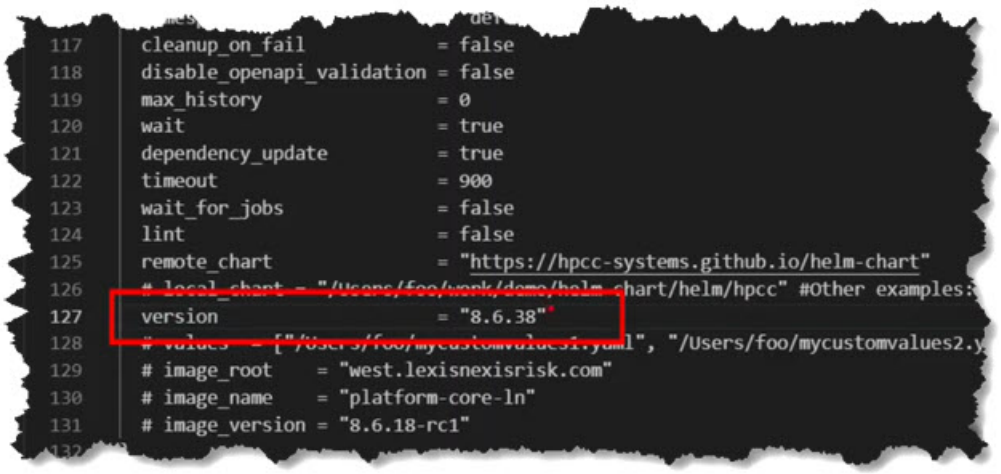
```
cp examples/admin.tfvars ./admin.tfvars
```

2. Modify the admin.tfvars file, once again add your user name and your email.

3. If you are using the Azure Cloud Shell, find the setting for **auto_launch_eclwatch** and set it to false as follows:

   ```
   auto_launch_eclwatch = false
   ```

4. Additionally there is a setting for **version** which by default is commented out. Optionally, uncomment the version setting and set to a specific version.



5. Make any other configuration changes and save the admin.tfvars file.

   Note: You can create multiple configuration files for different deployments. Such as the multiple versions which we just described. In that case you may want to save each configuration with a different name.

Initializing the Terraform Modules

After configuring the modules, the next step is to initialize. The *Terraform init* command declares the current working directory as the root or the calling module. During this operation, Terraform downloads all the child modules from their sources and place them in the appropriate relative directories.

Once again, the order is important. Initialize the modules in the same order of precedence, virtual network first, the storage account second, and then the AKS, or root.

Note: Whilst the order the files are applied is important, you can perform the initialization and apply steps after you modify the files while already in the respective directory.

To Initialize the Modules

1. Change directory to the modules directory.
2. Run terraform init in that directory:

```
terraform init
```

3. Confirm the module has been successfully initialized.
4. Apply the Module

Applying the Terraform Modules

This step generates a Terraform Plan to confirm your configuration choices. A Terraform plan displays exactly what it is going to do so you can review it before applying it. You can review and either approve to implement the plan or abort the plan and review your configuration modules for further changes.

When you issue the Terraform apply command it will validate the Terraform code and generate the plan, which you will then accept or reject to proceed. As with the previous steps, the order the modules are applied is important. You must apply the virtual network first, then the storage, and finally the root.

Note: Whilst the order the files are applied is important, you can perform the initialization and apply steps after you modify the files while already in the respective directory.

To Generate a Plan and Apply the Modules:

1. Change directory to that modules directory.
2. Run Terraform apply, specifying to use the admin.tfvars file you configured previously.

```
terraform apply –var-file=admin.tfvars
```

   Note: If you created multiple configuration files as described in the previous section (for the AKS module) you can specify to use that specific var-file.

3. The Terraform plan displays, review the plan and if it aligns with what you expected, approve the plan and enter yes.

   Note: If something does not look correct, do not enter yes. Anything other than yes will abort the application. You can then go and re-examine the admin.tfvars files from the previous steps and make any necessary changes.

4. Terraform initializes all the declared resources until they are all in a ready state. This can take a little time, as it is initializing several resources.

Successful completion displays a message similar to the following:

```
Apply complete! Resources: 11 added, 0 changed, 0 destroyed.
```

Note: The number of resources added, changed, or destroyed should match what the plan indicated in the previous step.

5. Repeat these steps for the storage_accounts directory and then for the root module directory.

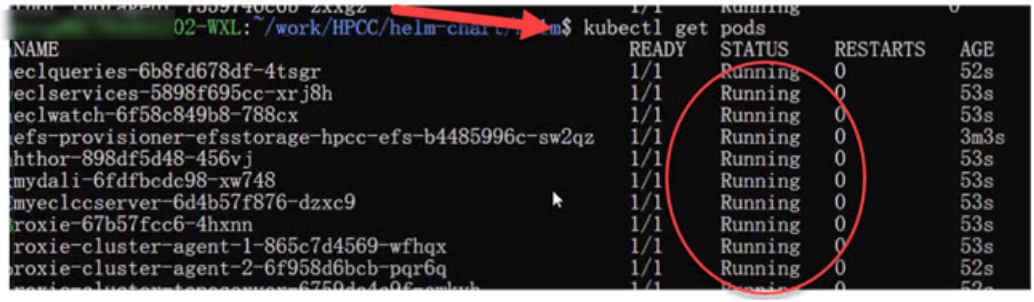
Once Terraform successfully applies all the modules in the correct sequence, and they all initialize and enter a ready state, your HPCC Systems cluster is up and running.

Verify the Installation

With your successful Terraform deployment Kubernetes has provisioned all the required HPCC Systems pods. To check their status run:

```
kubectl get pods
```

Note: If this is the first time helm install has been run, it may take some time for the pods to all get into a *Running* state. Azure needs to pull container images from Docker, bring each component online, etc.



Once all the pods STATUS is Running, the HPCC Systems cluster is ready to be use.

Accessing ECLWatch

To access ECLWatch, an external IP for the ESP running ECLWatch is required. This will be listed as the *eclwatch* service, and can be obtained by running the following command:

```
kubectl get svc
```

Your output should be similar to:

```
NAME        TYPE         CLUSTER-IP    EXTERNAL-IP     PORT(S)          AGE
eclservices ClusterIP    10.0.44.11    <none>          8010/TCP         11m
eclwatch    LoadBalancer 10.0.21.16    12.87.156.228   8010:30190/TCP   11m
kubernetes  ClusterIP    10.0.0.1      <none>          443/TCP          4h28m
mydali      ClusterIP    10.0.195.229  <none>          7070/TCP         11m
```

Use the EXTERNAL-IP listed for the ECLWatch service. Open a browser and go to http://<external-ip>:8010/, for example in this case, http://12.87.156.228:8010. If everything is working as expected, the ECLWatch landing page displays.

Taking Down The AKS Cluster

Destroying the AKS Cluster will do just that - completely destroy it. That is the Terraform term for taking down and removing all resources and processes Terraform deployed.

Just as with the installation, the order that modules are destroyed is also important. Keep in mind that **the AKS module must be destroyed before the Virtual network module.** Attempting to destroy resources in the wrong order could leave your deployment in an odd state and may incur unnecessary costs. To help reduce your total costs, always destroy your AKS when you do not intend on using it further.

Once configured the persisting Terraform modules can easily bring your deployment back up. An exact copy of the instance, can be raised simply by issuing the Terraform apply step you did earlier. This is the real beauty of the Terraform modules, once created they can be reused to generate an exact copy of your deployment. You could also have other configuration options readily available for deployment.

To destroy the Modules

1. Change directory to the root AKS directory: terraform-azurerm-hpcc
2. Run Terraform destroy

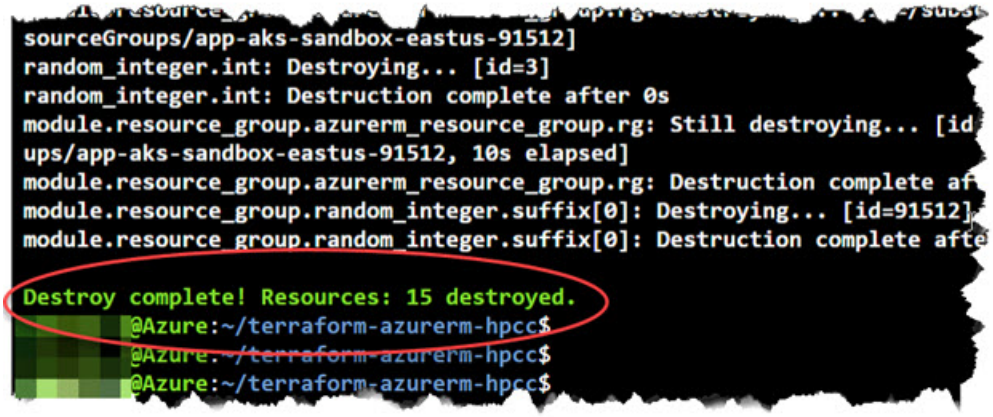
   ```
   terraform destroy –var-file=admin.tfvars
   ```

3. The Terraform plan displays, review the plan and if it looks correct approve the plan by entering yes.

   Entering anything other than yes aborts.

4. Repeat as necessary for the other modules. However ensure that the virtual network module is the last one to destroy, if you even choose to destroy it. (Apparently the cost for leaving the virtual network running is minimal, but check with your provider or manager for confirmation)

Terraform, much like the apply step, may take a few minutes to complete the destruction of all the resources. It will confirm the results once competed.



Customizing Configurations

Customization Techniques

This section walks through creating a customized configuration YAML file and deploying an HPCC Systems® platform using the default configuration plus the customizations. Once you understand the concepts in this chapter, you can refer to the next chapter for a reference to all configuration value settings.

There are several ways to customize a platform deployment. We recommend using methods that allow you to best take advantage of the configuration as code (CaC) practices. Configuration as code is the standard of managing configuration files in a version control system or repository.

The following is a list of common customization techniques:

- The first way to override a setting in the default configuration is via the command line using the **--set** parameter.

  This is the easiest, but the least compliant with CaC guidelines. It is also harder to keep track of overrides this way.

- The second way is to modify the default values saved using a command like:

  ```
  helm show values hpcc/hpcc > myvalues.yaml
  ```

  This could comply with CaC guidelines if you place that file under version control, but it makes it harder to utilize a newer default configuration when one becomes available.

- The third way, is the one we typically use. Use the default configuration plus a customization YAML file and use the -f parameter (or --values parameter) to the helm command. This uses the default configuration and only overrides the settings specified in the customization YAML. In addition, you can pass multiple YAML files in the same command, if desired.

  You can also use the -f option to pass a YAML file using a URL. For example:

  ```
  helm install mycluster hpcc/hpcc -f https://raw.githubusercontent.com/JD/MyHelm/main/noroxie.yaml
  ```

  For this tutorial, we will use the third method to stand up a platform with all the default settings but add some customizations. In the first example, instead of one Roxie, it will have two. In the second example, it will add a second 10-way Thor.

Create a Custom Configuration Chart for Two Roxies

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

   ```
   helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
   ```

   If you have added it, update to the latest charts:

   ```
   helm repo update
   ```

2. Create a new text file and name it **tworoxies.yaml** and open it in a text editor.

   You can use any text editor.

3. Save the default values to a text file:

   ```
   helm show values hpcc/hpcc > myvalues.yaml
   ```

4. Open the saved file (myvalues.yaml) in a text editor.

5. Copy the entire **roxie:** section and paste it into the new tworoxies.yaml file.

6. Copy the entire contents of the new tworoxies.yaml file, except the first line (roxie:), and paste it at the end of the file.

7. In the second block, edit the value for **name:** and change it to **roxie2.**

8. In the second block, edit the value for **prefix:** and change it to **roxie2.**

9. In the second block, edit the value for **name:** under **services:** and change it to **roxie2**.

10. Save the file and close the text editor.

    The resulting tworoxies.yaml file should look like this

    Note: The comments have been removed to simplify the example:

```
roxie:
- name: roxie
  disabled: false
  prefix: roxie
  services:
  - name: roxie
    servicePort: 9876
    listenQueue: 200
    numThreads: 30
    visibility: local
  replicas: 2
  numChannels: 2
  serverReplicas: 0
  localAgent: false
  traceLevel: 1
  topoServer:
    replicas: 1

- name: roxie2
  disabled: false
  prefix: roxie2
  services:
  - name: roxie2
    servicePort: 9876
```

```
    listenQueue: 200
    numThreads: 30
    visibility: local
  replicas: 2
  numChannels: 2
  serverReplicas: 0
  localAgent: false
  traceLevel: 1
  topoServer:
    replicas: 1
```

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the tworoxies.yaml file.
2. Deploy your HPCC Systems Platform, adding the new configuration to your command:

   ```
   helm install mycluster hpcc/hpcc -f tworoxies.yaml
   ```

3. After you confirm that your deployment is running, open ECL Watch.

   You should see two Roxie clusters available as Targets -- roxie and roxie2.

Create a Custom Configuration Chart for Two Thors

You can specify more than one custom configuration by repeating the -f parameter.

For example:

```
helm install mycluster hpcc/hpcc  -f tworoxies.yaml -f twothors.yaml
```

In this section, we will add a second 10-way Thor.

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

   ```
   helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
   ```

   If you have added it, update to the latest charts:

   ```
   helm repo update
   ```

2. Create a new text file and name it **twothors.yaml** and open it in a text editor.

   You can use any text editor.

3. Open the default values file that you saved earlier (myvalues.yaml) in a text editor.
4. Copy the entire **thor:** section and paste it into the new twothors.yaml file.
5. Copy the entire contents of the new twothors.yaml file, except the first line (thor:), and paste it at the end of the file.
6. In the second block, edit the value for **name:** and change it to **thor10.**
7. In the second block, edit the value for **prefix:** and change it to **thor10.**
8. In the second block, edit the value for **numWorkers:** and change it to **10.**
9. Save the file and close the text editor.

The resulting twothors.yaml file should look like this

Note: The comments have been removed to simplify the example:

```
thor:
- name: thor
  prefix: thor
  numWorkers: 2
  maxJobs: 4
  maxGraphs: 2
- name: thor10
  prefix: thor10
  numWorkers: 10
  maxJobs: 4
  maxGraphs: 2
```

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the twothors.yaml file.
2. Deploy your HPCC Systems Platform, adding the new configuration to your command:

```
# If you have previously stopped your cluster

helm install mycluster hpcc/hpcc -f tworoxies.yaml -f twothors.yaml

# To upgrade without stopping

helm upgrade mycluster hpcc/hpcc -f tworoxies.yaml -f twothors.yaml
```

3. After you confirm that your deployment is running, open ECL Watch.

   You should see two Thor clusters available as Targets -- thor and thor10.

Create a Custom Configuration Chart to AllowPipePrograms

You can specify more than one custom configuration by repeating the -f parameter.

For example:

```
helm install mycluster hpcc/hpcc  -f tworoxies.yaml -f thorWithPipe.yaml
```

In this section, we will modify the Thor to allow some Pipe Programs. In version 9.2.0 and greater, commands used in PIPE are restricted by default in containerized deployments unless explicitly allowed in the Helm chart.

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

   If you have added it, update to the latest charts:

```
helm repo update
```

2. Create a new text file and name it **thorWithPipe.yaml** and open it in a text editor.

   You can use any text editor.

3. Open the default values file that you saved earlier (myvalues.yaml) in a text editor.

4. Copy the entire **thor:** section and paste it into the new thorWithPipe.yaml file.

5. Add a block at the end:

```
  allowedPipePrograms:
  - sort
  - grep
  - echo
```

   This example enables three common programs. You can use the ones you want instead.

6. Save the file and close the text editor.

   The resulting thorWithPipe.yaml file should look like this

   Note: The comments have been removed to simplify the example:

```
thor:
- name: thor
  prefix: thor
  numWorkers: 2
  maxJobs: 4
  maxGraphs: 2
  allowedPipePrograms:
  - sort
  - grep
  - echo
```

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the thorWithPipe.yaml file file.

2. Deploy your HPCC Systems Platform, adding the new configuration to your command:

```
# If you have previously stopped your cluster

helm install mycluster hpcc/hpcc -f thorWithPipe.yaml

# To upgrade without stopping

helm upgrade mycluster hpcc/hpcc -f thorWithPipe.yaml
```

3. After you confirm that your deployment is running, submit a job that uses a PIPE action and specifies one of the programs you specified.

   Note: If the job is too simple, it will execute on hThor instead of Thor and this example doesn't enable Pipe programs on hThor.

   You can create another yaml file to allow Pipe Programs on ECL Agent or you can use:

```
#OPTION('pickBestEngine',FALSE);
```

   to force the job to run on Thor.

Create a Custom Configuration Chart for No Thor

In this section, we will create a YAML file to specify a platform deployment with no Thor.

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

If you have added it, update to the latest charts:

```
helm repo update
```

2. Create a new text file and name it **nothor.yaml** and open it in a text editor.

   You can use any text editor.

3. Edit the file so it disables Thor as follows:

```
thor: []
```

4. Save the file and close the text editor.

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the nothor.yaml file.

2. Deploy your HPCC Systems Platform, adding the new configuration to your command:

```
# If you have previously stopped your cluster

helm install mycluster hpcc/hpcc -f nothor.yaml

# To upgrade without stopping

helm upgrade mycluster hpcc/hpcc -f nothor.yaml
```

3. After you confirm that your deployment is running, open ECL Watch.

   You should not see any Thor cluster available as a Target.

Create a Custom Configuration Chart for No Roxie

In this section, we will create a YAML file to specify a platform deployment with no Roxie. While the outcome is similar to what we did in the previous section for no Thor, the technique is different.

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

   ```
   helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
   ```

   If you have added it, update to the latest charts:

   ```
   helm repo update
   ```

2. Create a new text file and name it **noroxie.yaml** and open it in a text editor.

   You can use any text editor.

3. Save the default values to a text file:

   ```
   helm show values hpcc/hpcc > myvalues.yaml
   ```

4. Open the saved file (myvalues.yaml) in a text editor.

5. Copy the entire **roxie:** section and paste it into the new noroxie.yaml file.

6. Copy the entire **eclagent:** section and paste it into the new noroxie.yaml file.

7. In the **roxie** block, edit the value for **disabled:** and change it to **true**

   You can remove everything else from the roxie: block except name.

8. In the **eclagent** block, delete the entire **name: roxie-workunit** block.

   This removes the instance of a Roxie acting as an ECL Agent.

9. Save the file and close the text editor.

   The resulting noroxie.yaml file should look like this:

   Note: The comments have been removed to simplify the example:

   ```
   roxie:
   - name: roxie
     disabled: true

   eclagent:
   - name: hthor
     replicas: 1
     maxActive: 4
     prefix: hthor
     useChildProcesses: false
     type: hthor
   ```

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the noroxie.yaml file.

2. Deploy your HPCC Systems Platform, adding the new configuration to your command:

   ```
   helm install mycluster hpcc/hpcc -f noroxie.yaml
   ```

3. After you confirm that your deployment is running, open ECL Watch.

   You should not see any Roxie cluster available as a Target.

Create a Custom Configuration Chart for Multiple Thors Listening to a Common Queue

In this section, we will create three Thors that listen to a common queue (in addition to their own queue). This provides the ability to define distinct Thor cluster configurations but allow them to form a single target behind a single queue. These clusters can be bound to certain node pools in different availability zones, if desired. You can use this example as a starting point and adjust the number of Thor clusters you want.

This is accomplished by defining additional auxiliary target queues for each Thor definition and using a common name as an auxiliary queue.

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

   ```
   helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
   ```

   If you have added it, update to the latest charts:

   ```
   helm repo update
   ```

2. Create a new text file and name it **threethorsonequeue.yaml** and open it in a text editor.

   You can use any text editor.

3. Open the default values file that you saved earlier (myvalues.yaml) in a text editor.

4. Copy the entire **thor:** section and paste it into the new threethorsonequeue.yaml file.

5. Copy the entire contents of the new yaml file, except the first line (thor:), and paste it at the end of the file **twice**.

   This creates three - name: sections.

6. Edit the file in the following manner:

   a. Give each Thor a unique value for **name:**.

      In this example, we use **thor1, thor2,** and **thor3**.

   b. Add an **auxQueues:** entry to each Thor block using a common name

      In this example, we are using

      auxQueues: [thorQ]

   c. Make sure the **prefix:** is the same in each Thor block.

7. Save the file and close the text editor.

   The resulting threethorsonequeue.yaml file should look like this:

   Note: The comments have been removed to simplify the example:

   ```
   thor:
   ```

```
- name: thor1
  auxQueues: [ thorQ ]
  maxGraphs: 2
  maxJobs: 2
  numWorkers: 4
  numWorkersPerPod: 2
  prefix: thor
- name: thor2
  maxGraphs: 2
  maxJobs: 2
  numWorkers: 4
  numWorkersPerPod: 2
  prefix: thor
  auxQueues: [ thorQ ]
- name: thor3
  maxGraphs: 2
  maxJobs: 2
  numWorkers: 4
  numWorkersPerPod: 2
  prefix: thor
  auxQueues: [ thorQ ]
```

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the threethorsonequeue.yaml file.

2. Deploy your HPCC Systems Platform, adding the new configuration to your command:

```
# If you have previously stopped your cluster

helm install mycluster hpcc/hpcc -f threethorsonequeue.yaml

# To upgrade without stopping

helm upgrade mycluster hpcc/hpcc -f threethorsonequeue.yaml
```

3. After you confirm that your deployment is running, open ECL Watch.

   You should see four Thor clusters available as Targets -- thor1, thor2, thor3, and a fourth queue that all three Thors listen to-- thorQ.

Create a Custom Configuration Chart for a Landing Zone only

In this section, we will create a custom configuration that deploys a "platform" containing only a Landing Zone. This can be useful if all you need is a landing zone server with dafilesrv running.

Note: This can only be deployed to a different namespace than any other platform instance.

1. If you have not already added the HPCC Systems repository to the helm repository list, add it now.

```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

   If you have added it, update to the latest charts:

```
helm repo update
```

2. Create a new text file and name it **lz.yaml** and open it in a text editor.

   You can use any text editor.

3. Copy and paste this code into the file:

```
dafilesrv:
- name: direct-access
  application: directio
  service:
    servicePort: 7100
    visibility: local
    tls: false
  resources:
    cpu: "2"
    memory: "8G"
dali: []
dfuserver: []
eclagent: []
eclccserver: []
eclscheduler: []
esp: []
roxie: []
sasha: null
thor: []
```

4. Save the file and close the text editor.

Deploy using the new custom configuration chart.

1. Open a terminal and navigate to the folder where you saved the lz.yaml file.

2. Deploy this LZ only "platform" with the the new configuration added to your command:

```
helm install mylz hpcc/hpcc -f lz.yaml
```

3. Confirm it is installed using this command:

```
helm list
```

Container Cost Tracking

With the advent of the containerized HPCC Systems platform, we have introduced cost tracking information. This is particularly useful when using cloud native HPCC Systems platform instances in a cloud setting where some planning and configuration can help reduce expenses.

New columns have been added to the workunits and the logical files pages in ECL Watch. These columns can be sorted by any cost columns, just like the other columns in ECL Watch, by clicking at the top of the column. In addition, file operation costs executed by workunits are provided in the workunit's metrics.

As the cost tracking matures, the cost calculations also improve significantly providing more accurate cost tracking. For example, data that was accessed from page cache doesn't incur any file access cost. HPCC Systems cost tracking now detects data that has been returned from the page cache and adjusts cost calculations appropriately resulting in more accurate cost calculations.

Types of Costs

There are several types of costs that are tracked.

- Execution Cost
- Compile Cost
- File Access Cost
- File Cost at Rest

NOTE: While reporting accuracy has significantly improved, all cost values calculated and displayed remain approximate. There are many variables that can result in inaccuracies. These cost values are only intended to be used as a guide.

Execution Cost

Execution Cost is the cost of executing the workunit, graph, and subgraphs on the Thor cluster. It includes the cost of all the nodes directly required to execute the job and includes the cost of:

- Worker nodes
- Compiler nodes
- Agent nodes and the manager node

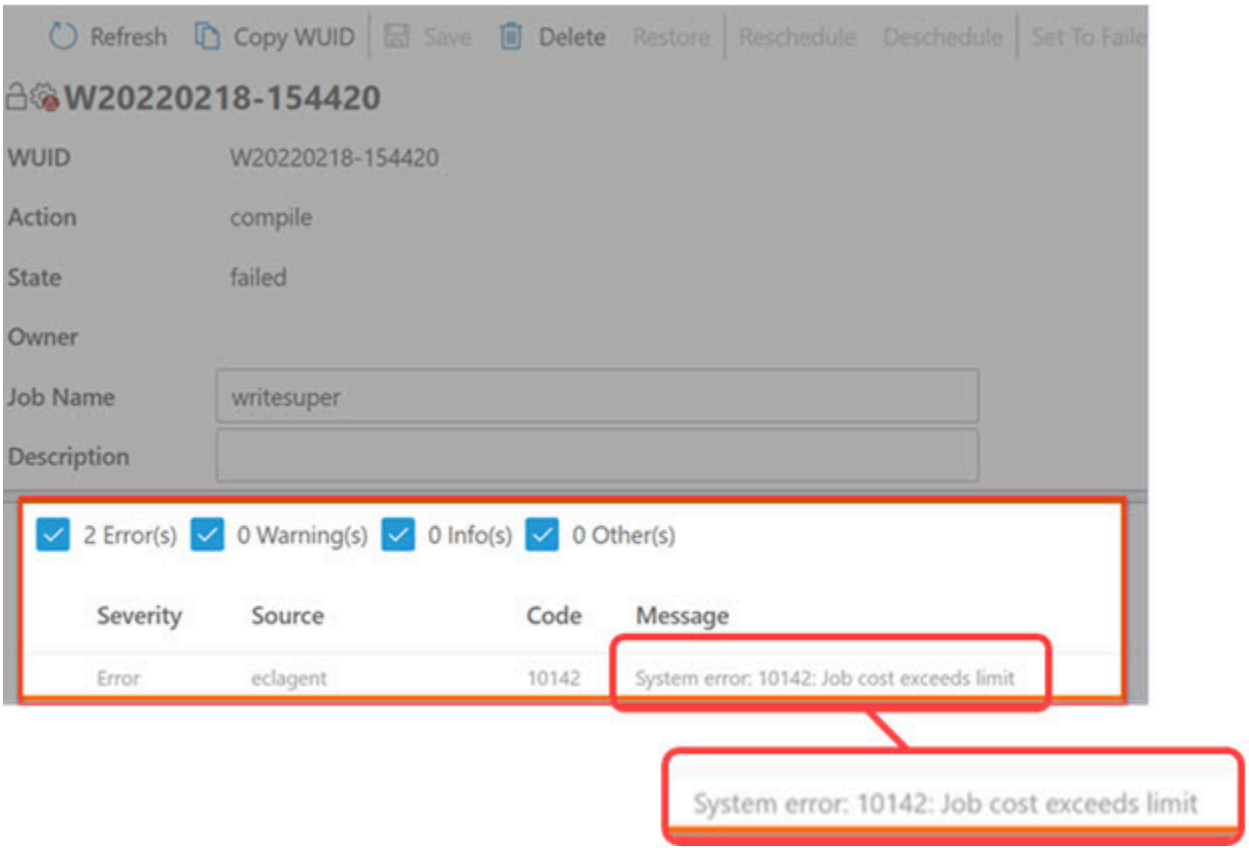
A workunit's execution cost value is displayed in ECL Watch on its summary page and is broken down at the graph, subgraph, and activity level. The graph and subgraph cost values are available in the metrics and graph viewer.

Note: The execution cost of ROXIE workunits is not currently implemented.

Job Guillotine

The risk of runaway costs is a concern for potentially uncapped usage-based charging. Thus the job guillotine feature is provided to manage this risk by setting limits on the costs using the *limit* and *hardlimit* values. When a jobs cost reaches a set amount, the job can be terminated and limit the costs that job may incur.

Note: This feature is only supported for Thor jobs currently.



Compile Cost

Compile costs are the costs of compiling the ECL code. The cost of compiling the ECL code is included as a column in the workunit list. In the workunit summary page there is a compile cost field.

Storage Costs

This is the cost of hosting the data in the storage plane or the *File Cost At Rest* value and the cost of reading/writing to the storage, is the *File Access Cost* value.

Note: Costs are not recorded for temporary or spill files, because the local storage is included in the price of the VM used to calculate the execution costs.

For logical files, the costs calculated are based on two types of file access.

- File Access Cost
- File Cost at Rest

File Access Cost

File access costs are the costs of reading and writing to the files. Many storage planes do have a separate charge for data operations. The cost of reading and writing to the file is included in the file access cost value. Any other cost associated with file operations (such as delete or copy) will not be tracked or included as part of file access cost at this time.

The costs incurred by a workunit for accessing logical files is also recorded in the workunit's statistics and attributes. The read/write cost is recorded at the activity record and cumulated at the graph, the subgraph,

and the workflow scope level. Cost Tracking detects data that has been returned from the page cache and adjusts cost calculations appropriately to produce more accurate cost calculations. These file access costs for a workunit are recorded with the workunit and displayed in the summary page and in the workunit's metrics.

File Cost at Rest

The File Cost at Rest field is shown in the Logical File summary page. It is the cost of storing the file without accessing the data. This includes only the storage costs associated with housing the file in the cloud. This value has been added to better differentiate between file storage costs.

Spill Files

Spill planes typically do not incur costs, as they are generally located on local storage. Starting with version 9.12, spill planes no longer inherit the global cost configuration. If a spill plane does not have its own storage cost configuration, any files stored on that spill plane will have their costs calculated as zero.

Costs Configuration

This section details setting the job costs configuration parameters. Job costs configuration on a cloud native HPCC Systems instance is done using the helm chart. By default the delivered *values.yaml* file contains a section for configuring costs. The costs are calculated using the default delivered values. Any desired changes can be done as a custom configuration similar to the customizations in the previous sections.

For example:

1. Create a new text file and name it **mycosts.yaml** and open it in a text editor.

   You can use any text editor.

2. Save the default values to a text file:

   ```
   helm show values hpcc/hpcc > myvalues.yaml
   ```

3. Open the saved file (myvalues.yaml) in a text editor.

4. Copy the entire **cost:** section and paste it into the new mycosts.yaml file.

5. Change any desired cost related values as appropriate.

6. Save the file and close the text editor.

7. Deploy your HPCC Systems Platform, adding the new configuration to your command:

   ```
   helm install mycluster hpcc/hpcc -f mycosts.yaml
   ```

The configuration values provide the pricing information and currency formatting information. The following cost configuration parameters are supported:

currencyCode	Used for currency formatting of cost values.
perCpu	Cost per hour of a single cpu.
storageAtRest	Storage cost per gigabyte per month.
storageReads	Cost per 10,000 read operations.
storageWrites	Cost per 10,000 write operations.

Configuring Cloud Costs

The default *values.yaml* configuration file is configured with the following cost parameters in the global/cost section:

```
  cost:
    currencyCode: USD
    perCpu: 0.126
    storageAtRest: 0.0135
    storageReads: 0.0485
    storageWrites: 0.0038
```

The **currencyCode** attribute should be configured with the ISO 4217 country code. (HPCC Systems platform defaults to USD if the currency code is missing).

The **perCpu** from the global/cost section applies to every component that has not been configured with its own perCpu value.

A perCpu value specific to a component may be set by adding a cost/perCPU attribute under that component section.

For example Dali:

```
  dali:
    - name: mydali
      cost:
        perCpu: 0.24
```

Thor Cost Configurations

The Thor components support additional cost parameters which are used for the job guillotine feature:

limit	Sets the “soft” cost limit that a workunit may incur. The limit is “soft” in that it may be overridden by the **maxCost** ECL option. A node will be terminated if it exceeds its **maxCost** value (if set) or the limit attribute value (if the **maxCost** not set).
hardlimit	Sets the absolute maximum cost limit, a limit that may not be overridden by setting the ECL option. The **maxCost** value exceeding the hardlimit will be ignored.

The following example sets the jobs cost limits, by adding the attributes to the Thor section of the configuration yaml.

```
thor:
- name: thor
  prefix: thor
  numWorkers: 2
  maxJobs: 4
  maxGraphs: 2
  cost:
    limit: 10.00      # maximum cost is $10, overridable with maxCost option
    hardlimit: 20.00  # maximum cost is $20, cannot be overridden
```

Storage Cost Parameters

The storage cost parameters (**storageAtRest**, **storageReads** and **storageWrites**) may be added under the storage plane cost section to set cost parameters specific to the storage plane.

For example:

```
storage:
  planes:
  - name: dali
    storageClass: ""
    storageSize: 1Gi
    prefix: "/var/lib/HPCCSystems/dalistorage"
    pvc: mycluster-hpcc-dalistorage-pvc
    category: dali
    cost:
      storageAtRest: 0.01
      storageReads: 0.001
      storageWrites: 0.04
```

The storage cost parameters under the global section are only used if no cost parameters are specified on the storage plane.

Cost Optimizer

The Cost Optimizer is a tool that analyzes Thor jobs to identify potential issues that could cause unnecessary costs. The optimizer is enabled by default and the cost optimizer analysis is generated automatically for all Thor jobs.

Cost Optimizer Functionality

The Cost Optimizer functions by examining the workunit's graph and metrics to highlight issues such as skews, spills, and other suboptimal operations. Addressing these issues can improve performance and/or reduce costs.

For example, the Cost Optimizer may detect among the following noteworthy issues:

Skews: Imbalanced data distribution causing some nodes to process more data than others.

Spills: Excessive data spilling to disk due to insufficient memory allocation.

Inefficient Joins: Suboptimal join operations leading to increased processing time.

These issues result in longer execution times for the cluster, thereby incurring additional costs.

Cost Estimation

If your HPCC Systems deployment is configured to use the cost tracking settings, the Cost Optimizer provides an estimated monetary value for each identified issue.

All identified issues are reported on the workunit's summary page in ECL Watch.

Review the reported issues to determine their validity and impact to your work and consider making changes to improve efficiency and reduce unnecessary costs.

Securing Credentials

Utilizing HPCC Systems in a containerized environment has some unique security concerns by externalizing typically internalized components, such as the LDAP administrators credentials.

Securing the LDAP administrators credentials is accomplished by using either Kubernetes or Hashicorp Vault secrets. As a prerequisite you should be familiar with setting up Kubernetes and/or Hashicorp Vault secrets.

The LDAP administrators account must have administrator rights to all of the Base DNs used by the HPCC Systems platform. In a cloud deployment these credentials can be exposed. A good practice then, for these administrator credentials is to be secured either using Kubernetes secrets or the Hashicorp Vault.

Securing Credentials in Kubernetes

To create a secret in Kubernetes to store the LDAP administrators user account credentials, use a command line interface to Kubernetes, and execute a command similar to the following example:

```
kubectl create secret generic admincredssecretname --from-literal=username=hpcc_admin \
  --from-literal=password=t0pS3cr3tP@ssw0rd
```

In the above example, the Kubernetes secret name, is "admincredssecretname" and it contains the LDAP administrators account "username" and "password" key/values. This stores the LDAP administrators username and password as a Kubernetes secret. Any additional properties are ignored.

You can verify the secret you just created by executing the following Kubernetes command.

```
kubectl get secret admincredssecretname
```

For more information about Kubernetes see the appropriate Kubernetes documentation for your implementation.

Using the Kubernetes Secret

To deploy the Kubernetes secrets override the "secrets:" section in HPCC-Platform/helm/hpcc/values.yaml, or deploy with your own customized chart. For more information about customizing your HPCC Systems containerized deployment see the above sections on customization techniques.

In your chart, create a unique key name used to reference the secret, and set it to the name of secret that you created in the previous step. In the above example it was "admincredssecretname".

You can optionally define additional secrets as required by your platform security configuration. Each of these secrets would be created as described above and given unique names. The example below indicates how you can add any additional credentials or secrets to your Helm chart(s) if necessary.

The "admincredsmountname" key/value pair already exists by default in the delivered HPCC Systems values.yaml file. The key is referenced in the component's ldap.yaml file. You may override these and add additional key/values as needed. The following example illustrates adding "additionalsecretname" and that name must match the name of the additional secret created using the steps above.

```
  secrets:
     authn:
       admincredsmountname: "admincredssecretname"   #exernalize HPCC Admin creds
```

```
      additionalmountname: "additionalsecretname"   #alternate HPCC Admin creds
```

Enable LDAP Authentication

In the delivered HPCC-Platform/esp/applications/common/ldap/ldap.yaml file, the "ldapAdminSecretKey" is already set to the key mount name illustrated in the example above. To enable LDAP authentication and to modify this value, you or your systems administrator can override the ESP/ECLWatch Helm component located in values.yaml chart as illustrated in the following example:

```
esp:
- name: eclwatch
  application: eclwatch
  auth: ldap
  ldap:
    ldapAddress: "myldapserver"
    ldapAdminSecretKey: "additionaltmountname"  # use alternate secrets creds
```

Securing credentials in HashiCorp Vault

To create and store secrets in the HashiCorp Vault, from the command command line interface, execute the following Vault commands. The secret name used in the example below is "myvaultadmincreds" and must be prefixed with "secret/authn/" as illustrated. The LDAP administrator "username" and "password" key/values are required. Additional properties are ignored.

```
vault kv put secret/authn/myvaultadmincreds username=hpcc_admin password=t0pS3cr3tP@ssw0rd
```

Where the "secret/authn/myvaultadmincreds" is the name of the secret containing the LDAP administrator username and password.

To verify and confirm the secret values, execute the following command:

```
vault kv get secret/authn/myvaultadmincreds
```

For more information about creating secrets for HashiCorp Vault see the appropriate HashiCorp documentation for your implementation.

Deploy the HashiCorp Vault

Deploy the HashiCorp Vault secrets when you override the "secrets:" section in HPCC-Platform/helm/hpcc/values.yaml, or in your customized configuration chart. For more information about customizing your HPCC Systems containerized deployment see the above sections on customization techniques.

The Vault name value is defined for this example in the values-secrets.yaml configuration chart. You can find an example of this chart in the HPCC-Platform repository under /helm/examples/secrets/values-secrets.yaml.

```
  vaults:
  authn:
    - name: my-authn-vault
      #The data node in the URL is there for use by the REST API
      #The path inside the vault starts after /data
      url: http://${env.VAULT_SERVICE_HOST}:${env.VAULT_SERVICE_PORT}/v1/secret/data/authn/${secret}
      kind: kv-v2
```

You could put this into your own customization chart where you supply your deployment with the name of the vault containing the credentials.

Deploy the Akeyless Vaultless Platform

To use Akeyless, set the vault type to *akeyless* and the kind to *akeyless*. The URL should point to the Akeyless API endpoint or your gateway. Provide an access ID and access key, typically from environment variables or a secure secret. A token can also be supplied via the client-secret if desired.

```
  vaults:
  authn:
    - name: my-authn-akeyless
      type: akeyless
      kind: akeyless
      url: https://api.akeyless.io
      accessId: ${env.AKEYLESS_ACCESS_ID}
      accessKey: ${env.AKEYLESS_ACCESS_KEY}
```

The secret name referenced by HPCC (for example, in LDAP settings) is passed to Akeyless as the secret name in the get-secret-value API. If a version is provided, it will be used when retrieving the secret.

Referencing Vault Stored Authentication

The key names "ldapAdminSecretKey" and "ldapAdminVaultId" are used by the HPCC Systems security manager to resolve the secrets, and must match exactly when using the Vault name set up in the previous steps.

```
esp:
- name: eclwatch
  application: eclwatch
  auth: ldap
  ldap:
    ldapAddress: "myldapserver"
    ldapAdminSecretKey: "myvaultadmincreds"
    ldapAdminVaultId: "my-authn-vault"
```

Configuration Values

This chapter describes the configuration of HPCC Systems for a Kubernetes Containerized deployment. The following sections detail how configurations are supplied to helm charts, how to find out what options are available and some details of the configuration file structure. Subsequent sections will also provide a brief walk through of some of the contents of the default *values.yaml* file used in configuring the HPCC Systems for a containerized deployment.

The Container Environment

One of the ideas behind our move to the cloud was to try and simplify the system configuration while also delivering a solution flexible enough to meet the demands of our community while taking advantage of container features without sacrificing performance.

The entire HPCC Systems configuration in the container space, is governed by a single file, a *values.yaml* file, and its associated schema (*values.schema.json*) file.

The *values.yaml* and how it is used

The supplied stock *values.yaml* file provided in the HPCC Systems repository is the delivered configuration values for the "hpcc" Helm chart. The *values.yaml* file is used by the Helm chart to control how HPCC Systems is deployed to the cloud. This *values.yaml* file is a single file used to configure and get an HPCC Systems instance up and running on Kubernetes. The *values.yaml* file defines everything that happens to configure and/or define your system for a containerized deployment. You should use the values file provided as a basis for modeling customizations for your deployment specific to your requirements.

The HPCC Systems *values.yaml* file can be found in the HPCC Systems github repository. To use the HPCC Systems Helm chart, first add the hpcc chart repository using Helm, then access the Helm chart values from the charts in that repository.

For example, when you add the "hpcc" repository, as recommended prior to installing the Helm chart with the following command:

```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart
```

You can now view the HPCC Systems delivered charts and see the values there by issuing:

```
helm show values hpcc/hpcc
```

You can capture the output of this command, look at how the defaults are configured and use it as a basis for your customization.

The values-schema.json

The *values-schema.json* is a JSON file that declares what is valid and what is not within the sum total of the merged values that are passed into Helm at install time. It defines what values are allowed, and validates the values file against them. All the core items are declared in the schema file, which contains comments and descriptions. While the default *values.yaml* file also contains comments on the most important elements.

If you wanted to know what options are available for any particular component then the schema is a good place to start. If the value exists and is allowed per the schema it can then be added when you install.

The schema file typically contains (for a property) a name and a description. It will often include details of the type, and items it can contain, or if it is a list or dictionary. For instance:

```
    "roxie": {
      "description": "roxie process",
      "type": "array"
      "items": { "$ref": "#/definitions/roxie" }
    },
```

Each plane, in the schema file has a list of properties generally containing a prefix (path), a subpath (sub-path), and additional properties. For example, for a storage plane the schema file has a list of properties including the prefix. The "planes" in this case are a reference ($ref) to another section. The schema file is complete, and contains everything required including descriptions which should be relatively self-explanatory.

```
    "storage": {
      "type": "object",
      "properties": {
        "hostGroups": {
          "$ref": "#/definitions/hostGroups"
        },
        "planes": {
          "$ref": "#/definitions/storagePlanes"
        }
      },
      "additionalProperties": false
```

Note the *additionalProperties* value typically at the end of each section in the schema. It specifies whether the values allow for additional properties or not. If that *additionalProperties* value is present and set to false, then no other properties are allowed and the property list is complete.

In working with the HPCC Systems *values.yaml*, the values file must validate against this schema. If there is a value that is not allowed as defined in the schema file it will not start and instead generate an ERROR.

HPCC Systems Components and the *values.yaml* File

The HPCC Systems Helm charts all ship with stock/default values. These Helm charts have a set of default values ideally to be used as a guide in configuring your deployment. Generally, every HPCC Systems component is a list. That list defines the properties for each instance of the component.

This section will provide additional details and any noteworthy insight for the HPCC Systems components defined in the *values.yaml* file.

The HPCC Systems Components

One of the key differences between the bare-metal and container/cloud is that in bare-metal storage is directly tied to the Thor or the Thor worker nodes, and the Roxie worker nodes, or even in the case of the ECLCC Server the DLLs. In containers these are completely separate and anything having to do with files is defined in the *values.yaml*.

In containers component instances run dynamically. For instance, if you have configured your system to use a 50-way Thor, then a 50-way Thor will be spawned when a job is queued to it. When that job is finished that Thor instance will disappear. This is the same pattern for the other components as well.

Every component should have a resources entry, in the delivered *values.yaml* file the resources are present but commented out as indicated here.

```
#resources:
#  cpu: "1"
#  memory: "4G"
```

The stock values file will work and allow you to stand up a functional system, however you should define the component resources in a manner that corresponds best to your operational strategy.

The Systems Services

Most of the HPCC Systems components have a service definition entry, similar to the resources entry. All the components that have service definitions follow this same pattern.

Any service related info needs to be under a service object, for example:

```
service:
  servicePort: 7200
  visibility: local
```

This applies to most all of the HPCC Systems components, ESP, Dali, dafilesrv, and Sasha. Roxie's specification is slightly different, in that it has its service defined under "roxieservice". Each Roxie can then have multiple "roxieservice" definitions. (see schema).

Dali

When configuring Dali, which also has a resources section, it is going to want plenty of memory and a good amount of CPU as well. It is very important to define these carefully. Otherwise Kubernetes could assign all the pods to the same virtual machine and components fighting for memory will crush them. Therefore more memory assigned the better. If you define these wrong and a process uses more memory than configured, Kubernetes will kill the pod.

Components: dafilesvrs, dfuserver

The HPCC Systems components of dafilesvrs, eclccservers, dfuserver, are declared as lists in the YAML file, as is the ECL Agent.

Consider the dfuserver which is in the delivered HPCC Systems *values.yaml* as:

```
dfuserver:
- name: dfuserver
  maxJobs: 1
```

If you were to add a mydfuserver as follows

```
dfuserver:
- name: dfuserver
  maxJobs: 1
- name: mydfuserver
  maxJobs: 1
```

In this scenario you would have another item here named mydfuserver and it would show up in ECLWatch and you can submit items to that.

If you wanted to add another dfuserver, you can add that to the list similarly. You can likewise instantiate other components by adding them to their respective lists.

ECL Agent and ECLCC Server

Values of note for the ECL Agent and ECLCC Server.

useChildProcess -- As defined in the schema, launches each workunit compile as a child process rather than in its own container. When you submit a job or query to compile it gets queued and processed, with this option set to true it will spawn a child process utilizing almost no additional overhead in starting. Ideal for sending many small jobs to compile. However, because each compile job is no longer executed as an independent pod with it's own resource specifications, but instead runs as a child process within the ECLCC Server pod itself, the ECLCC Server pod must be defined with adequate resources for itself (minimal for listening to the queue etc.) and all the jobs it may have to run in parallel.

For example, imagine *maxJobs* is set to 4, and 4 large queries are queued rapidly, that will mean 4 child processes are launched each consuming cpu and memory within the ECLCC Server pod. With the component configured with *useChildProcesses* set to true, each job will run in the same pod (up to the value of *maxJobs* in parallel). Therefore with *useChildProcesses* enabled, the component resources must be defined such that the pod has enough resources to handle the resource demands of all those jobs to be able to run in parallel.

With useChildProcess enabled it could be rather expensive in most cloud pricing models, and rather wasteful if there aren't any jobs running. Instead you can set this *useChildprocess* to false (the default) to start a pod to compile each query with only the required memory for the job which will be disposed of when done. Now this model also has overheard, perhaps 20 seconds to a minute to spawn the Kubernetes cluster to process

the job. Which may not be ideal for an environment which is sending several small jobs, but rather larger jobs which would minimize the effect of the overhead in starting the Kubernetes cluster.

Setting *useChildProcess* to false better allows for the possibility of dynamic scaling. For jobs which would take a long while to compile, the extra (start up) overhead is minimal, and that would be the ideal case to have the *useChildProcess* as false. Setting *useChildProcess* to false only allows 1 pod per compile, though there is an attribute for putting a time limit on that compilation.

ChildProcessTimeLimit is the time limit (in seconds) for child process compilation before aborting and using a separate container, when the *useChildProcesses* is false.

maxActive -- The maximum number of jobs that can be run in parallel. Again use caution because each job will need enough memory to run. For instance, if *maxActive* is set to 2000, you could submit a very big job and in that case spawn some 2000 jobs using a considerable amount of resources, which could potentially run up a rather expensive compilation bill, again depending on your cloud provider and your billing plan.

Sasha

The configuration for Sasha is an outlier as it is a dictionary type structure and not a list. You can't have more than one archiver or dfuwu-archiver as that is a value limitation, you can choose to either have the service or not (set the 'disabled' value to true).

global-message-housekeeping

The *global-message-housekeeping* Sasha service manages the lifecycle of global messages stored in the system. It periodically scans persisted global messages and permanently deletes those that exceed the configured retention period.

The following configuration parameters are available for the *global-message-housekeeping* service:

disabled -- Setting this value to *true* disables the service. The default value is *false*.

interval -- The interval (in hours) between housekeeping maintenance cycles. Setting this value to zero disables the service. The default value is *24*.

retentionDays -- The number of days to retain global messages before they are permanently deleted. Messages older than this threshold are removed during each maintenance cycle. The default value is *30*.

Example Helm configuration:

```
sasha:
  global-message-housekeeping:
    disabled: false
    interval: 24
    retentionDays: 60
```

Thor

Thor instances run dynamically, as do the other components in containers. The configuration for Thor also consists of a list of Thor instances. Each instance dynamically spawns a collection of pods (manager + N workers) when jobs are queued to it. When idle there are no worker (or manager) pods running.

If you wanted a 50-way Thor you set the number of workers, the **numWorkers** value to 50 and you would have a 50-way Thor. As indicated in the following example:

```
thor:
- name: thor
```

```
  prefix: thor
  numWorkers: 50
```

In doing so, ideally you should rename the resource to something which clearly describes it, such as *thor_50* as in the following example.

```
- name: thor_50
```

Updating the *numWorkers* value will restart the Thor agent listening to the queue, causing all new jobs to use the new configuration.

maxJobs -- Controls the number of jobs, specifically *maxJobs* sets the maximum number of jobs.

maxGraphs -- Limits the maximum amount of graphs. It generally makes sense to keep this value below or at the same number as *maxJobs*, since not all jobs submit graphs and when they do the Thor jobs are not executing graphs all the time. If there are more than 2 submitted (Thor) graphs, the second would be blocked until the next Thor instance becomes available.

The idea here is that jobs may spend significant amount of time outside of graphs, such as waiting on a workflow state (outside of the Thor engine itself), blocked on a persist, or updating super files, etc. Then it makes sense for Thor to have a higher limit of concurrent jobs (*maxJobs*) than graphs (*maxGraphs* / Thor instances). Since Thor instances (graphs) are relatively expensive (lots of pods/higher resource use), while workflow pods (jobs) are comparatively cheap.

Thus, the delivered (example) chart values defines *maxJobs* to be greater than *maxGraphs*. Jobs queued to a Thor aren't always running graphs. Therefore it can make sense to have more of these jobs, which are not consuming a large Thor and all its resources, but restrict the max number of Thor instances running.

Thor has 3 components (that correspond to the resource sections).

1. Workflow

2. Manager

3. Workers

The Manager and Workers are launched together and consume quite a bit of resoures (and nodes) typically. While the Workflow is inexpensive and usually doesn't require as many resources. You might expect in a Kubernetes world, many of them would co-exist on the same node (and therefore be inexpensive). So it makes sense for *maxJobs* to be higher, and *maxGraphs* to be lower

In Kubernetes, jobs run independently in their own pods. While in bare-metal we can have jobs that could affect other jobs because they are running in the same process space.

Thor and hThor Memory

The Thor and hThor *memory* sections allow the resource memory of the component to be refined into different areas.

For example, the "workerMemory" for a Thor defined as:

```
thor:
- name: thor
  prefix: thor
  numWorkers: 2
  maxJobs: 4
  maxGraphs: 2
  managerResources:
```

```
    cpu: "1"
    memory: "2G"
  workerResources:
    cpu: "4"
    memory: "4G"
  workerMemory:
    query: "3G"
    thirdParty: "500M"
  eclAgentResources:
    cpu: "1"
    memory: "2G"
```

The "*workerResources*" section will tell Kubernetes to resource 4G per worker pod. By default Thor will reserve 90% of this memory to use for HPCC query memory (roxiemem). The remaining 10% is left for all other non-row based (roxiemem) usage, such as general heap, OS overheads, etc. There is no allowance for any 3rd party library, plugins, or embedded language usage within this default. In other words, if for example embedded Python allocates 4G, the process will soon fail with an out of memory error, when it starts to use any memory, since it was expecting 90% of that 4G to be freely available to use for itself.

These defaults can be overridden by the memory sections. In this example, *workerMemory.query* defines that 3G of the available resourced memory should be assigned to query memory, and 500M to "thirdParty" uses.

This limits the HPCC Systems memory *roxiemem* usage to exactly 3G, leaving 1G free other purposes. The "thirdParty" is not actually allocated, but is used solely as part of the running total, to ensure that the configuration doesn't specify a total in this section larger than the resources section, e.g., if "thirdParty" was set to "2G" in the above section, there would be a runtime complaint when Thor ran that the definition exceeded the resource limit.

It is also possible to override the default recommended percentage (90% by default), by setting *maxMem-Percentage*. If "query" is not defined, then it is calculated to be the recommended max memory minus the defined memory (e.g., "thirdParty").

In Thor there are 3 resource areas, *eclAgent*, *ThorManager*, and *ThorWorker*(s). Each has a *Resource area that defines their Kubernetes resource needs, and a corresponding *Memory section that can be used to override default memory allocation requirements.

These settings can also be overridden on a per query basis, via workunit options following the pattern: <memory-section-name>.<property>. For example: #option('workerMemory.thirdParty', "1G");

Note: Currently there is only "query" (HPCC roxiemem usage) and "thirdParty" for all/any 3rd party usage. It's possible that further categories will be added in future, like "python" or "java" - that specifically define memory uses for those targets.

The HPCC Systems *values.yaml* file

The delivered HPCC systems *values.yaml* file is more of an example providing a basic type configuration which should be customized for your specific needs. One of the main ideas behind the values file is to be able to relatively easily customize it to your specific scenario. The delivered chart is set up to be sensible enough to understand, while also allowing for relatively easy customization to configure a system to your specific requirements. This section will take a closer look at some aspects of the delivered *values.yaml*.

The delivered HPCC Systems Values file primarily consists of the following areas:

global	storage	visibilities
data planes	certificates	security
secrets	components	

The subsequent sections will examine some of these more closely and why each of them is there.

Storage

Containerized Storage is another key concept that differs from bare-metal. There are a few differences between container and bare-metal storage. The Storage section is fairly well defined between the schema file, and the *values.yaml*. A good approach towards storage is to clearly understand your storage needs, and to outline them, and once you have that basic structure in mind the schema can help to fill in the details. The schema should have a decent description for each attribute. All storage should be defined via planes. There is a relevant comment in the *values.yaml* further describing storage.

```
## storage:
##
## 1. If an engine component has the dataPlane property set,
#       then that plane will be the default data location for that component.
## 2. If there is a plane definition with a category of "data"
#       then the first matching plane will be the default data location
##
## If a data plane contains the storageClass property then an implicit pvc
#       will be created for that data plane.
##
## If plane.pvc is defined, a Persistent Volume Claim must exist with that name,
#       storageClass and storageSize are not used.
##
## If plane.storageClass is defined, storageClassName: <storageClass>
## If set to "-", storageClassName: "", which disables dynamic provisioning
## If set to "", choosing the default provisioner.
#       (gp2 on AWS, standard on GKE, AWS & OpenStack)
##
## plane.forcePermissions=true is required by some types of provisioned
## storage, where the mounted filing system has insufficient permissions to be
## read by the hpcc pods. Examples include using hostpath storage (e.g. on
## minikube and docker for desktop), or using NFS mounted storage.
```

There are different categories of storage, for an HPCC Systems deployment you must have at a minimum a dali category, a dll category, and at least 1 data category. These types are generally applicable for every configuration in addition to other optional categories of data.

All storage should be in a storage plane definition. This is best described in the comment in the storage definition in the values file.

```
planes:
  #   name: <required>
```

```
  #   prefix: <path>                        # Root directory for accessing the plane
                                            # (if pvc defined),
  #                                         # or url to access plane.
  #   category: data|dali|lz|dll|spill|temp # What category of data is stored on this plane?
  #
  # For dynamic pvc creation:
  #   storageClass: ''
  #   storageSize: 1Gi
  #
  # For persistent storage:
  #   pvc: <name>                           # The name of the persistant volume claim
  #   forcePermissions: false
  #   hosts: [ <host-list> ]                 # Inline list of hosts
  #   hostGroup: <name>                     # Name of the host group for bare-metal
  #                                         # must match the name of the storage plane..
  #
  # Other options:
  #   subPath: <relative-path>              # Optional sub directory within <prefix>
  #                                         # to use as the root directory
  #   numDevices: 1                         # number of devices that are part of the plane
  #   secret: <secret-id>                   # what secret is required to access the files.
  #                                         # This could optionally become a list if required
                                            # (or add secrets:).

  #   defaultSprayParts: 4                  # The number of partitions created when spraying
                                            # (default: 1)

  #   cost:                                 # The storage cost
  #     storageAtRest: 0.0135               # Storage at rest cost: cost per GiB/month
```

Each plane has 3 required fields: The name, the category and the prefix.

When the system is installed,using the stock supplied values it will create a storage volume which has 1 GB capacity via the following properties.

For example:

```
- name: dali
  storageClass: ""
  storageSize: 1Gi
  prefix: "/var/lib/HPCCSystems/dalistorage"
  category: dali
```

Most commonly the prefix: defines the path within the container where the storage is mounted. The prefix can be a URL for blob storage. All pods will use the (prefix:) path to access the storage.

For the above example, when you look at the storage list, the *storageSize* will create a volume with 1 GB capacity. The prefix will be the path, the category is used to limit access to the data, and to minimize the number of volumes accessible from each component.

The dynamic storage lists in the *values.yaml* file are characterized by the storageClass: and storageSize: values.

storageClass: defines which storage provisioner should be used to allocate the storage. A blank storage class indicates it should use the default cloud providers storage class.

storageSize: As indicated in the example, defines the capacity of the volume.

Storage Category

Storage category is used to indicate the kind of data that is being stored in that location. Different planes are used for the different categories to isolate the different types of data from each other, but also because they

often require different performance characteristics. A named plane may only store one category of data. The following sections look at the currently supported categories of data used in our containerized deployment.

```
category: data|dali|lz|dll|spill|temp  # What category of data is stored on this plane?
```

The system itself can write out to any data plane. This is how the data category can help to improve performance. For example, if you have an index, Roxie would want rapid access to data, versus other components.

Some components may use only 1 category, some can use several. The values file can contain more than one storage plane definition for each category. The first storage plane in the list for each category is used as the default location to store that category of data. These categories minimize the exposure of plane data to components that don't need them. For example the ECLCC Server component does not need to know about landing zones, or where Dali stores its data, so it only mounts the plane categories it needs.

Storage Defaults

As of version 9.14, HPCC Systems supports a new `defaults` section under storage configuration that provides fallback settings for storage planes. This feature allows administrators to define common configuration values that can be inherited by individual storage planes, reducing duplication and simplifying configuration management.

Storage Defaults Configuration

The storage defaults section is added as a sibling to the `planes` section in your Helm values configuration.

How Storage Defaults Work

The defaults mechanism works as follows:

1. **Priority:** Individual plane settings take precedence over defaults

2. **Inheritance:** Storage planes automatically inherit any properties not explicitly defined at the plane level

Ephemeral Storage

Ephemeral storage is allocated when the HPCC Systems cluster is installed and deleted when the chart is uninstalled. This is helpful in keeping cloud costs down but may not be appropriate for your data.

In your system, you would want to override the delivered stock value(s) with storage appropriate for your specific needs. The supplied values create ephemeral or temporary persistent volumes that get automatically deleted when the chart is uninstalled. You probably want the storage to be persistent. You should customize the storage to a more suitable configuration for your needs.

Persistent Storage

Kubernetes uses persistent volume claims (pvcs) to provide access to data storage. HPCC Systems supports cloud storage through the cloud provider that can be exposed through these persistent volume claims.

Persistent Volume Claims can be created by overriding the storage values in the delivered Helm chart. The values in the *examples/local/values-localfile.yaml* provided override the corresponding entries in the original delivered stock HPCC Systems helm chart. The localfile chart creates persistent storage volumes. You can use the *values-localfile.yaml* directly (as demonstrated in separate docs/tutorials) or you can use it as a basis for creating your own override chart.

To define a storage plane that utilizes a PVC, you must decide on where that data will reside. You create the storage directories, with the appropriate names and then you can install the localfiles Helm chart to create the volumes to use the local storage option, such as in the following example:

```
helm install mycluster hpcc/hpcc -f examples/local/values-localfile.yaml
```

Note: The settings for the PVC's must be ReadWriteMany, except for Dali which can be ReadWriteOnce.

There are a number of resources, blogs, tutorials, even developer videos that provide step-by-step detail for creating persistent storage volumes.

Bare Metal Storage

There are two aspects to using bare-metal storage in the Kubernetes system. The first is the *hostGroups* entry in the storage section which provides named lists of hosts. The *hostGroups* entries can take one of two forms. This is the most common form, and directly associates a list of host names with a name:

```
storage:
  hostGroups:
  - name:  <name> "The name of the host group"
    hosts: [ "a list of host names" ]
```

The second form allows one host group to be derived from another:

```
storage:
  hostGroups:
  - name: "The name of the host group process"
    hostGroup: "Name of the hostgroup to create a subset of"
    count: <Number of hosts in the subset>
    offset: <the first host to include in the subset>
    delta: <Cycle offset to apply to the hosts>
```

Some typical examples with bare-metal clusters are smaller subsets of the host, or the same hosts, but storing different parts on different nodes, for example:

```
storage:
  hostGroups:
  - name: groupABCDE # Explicit list of hosts
    hosts: [A, B, C, D, E]
  - name groupCDE # Subset of the group last 3 hosts
    hostGroup: groupABCDE
    count: 3
    offset: 2
  - name groupDEC # Same set of hosts, but different part->host mapping
    hostGroup: groupCDE
    delta: 1
```

The second aspect is to add a property to the storage plane definition to indicate which hosts are associated with it. There are two options:

- **hostGroup: <name>** The name of the host group for bare-metal. The name of the hostGroup must match the name of the storage plane.
- **hosts: <list-of-namesname>** An inline list of hosts. Primarily useful for defining one-off external landing zones.

For Example:

```
storage:
  planes:
  - name: demoOne
    category: data
    prefix: "/home/demo/temp"
    hostGroup: groupABCD # The name of the hostGroup
  - name: myDropZone
    category: lz
```

```
    prefix: "/home/demo/mydropzone"
    hosts: [ 'mylandingzone.com' ] # Inline reference to an external host.
```

Remote Storage

You can configure your HPCC Systems cloud deployment to access logical files from other remote environments. You configure this remote storage by adding a "remote" section to your helm chart.

The *storage.remote* section is a list of named remote environments that define the remote service url and a section that maps the remote plane names to local plane names. The local planes referenced are special planes with the category 'remote'. They are read-only and only exposed to the engines which can read from them.

For Example:

```
storage:
  planes:
...
  - name: hpcc2-stddata
    pvc: hpcc2-stddata-pv
    prefix: "/var/lib/HPCCSystems/hpcc2/hpcc-stddata"
    category: remote
  - name: hpcc2-fastdata
    pvc: hpcc2-fastdata-pv
    prefix: "/var/lib/HPCCSystems/hpcc2/hpcc-fastdata"
    category: remote
  remote:
  - name: hpcc2
    service: http://20.102.22.31:8010
    planes:
    - remote: data
      local: hpcc2-stddata
    - remote: fastdata
      local: hpcc2-fastdata
```

This example defines a remote target called "hpcc2" whose DFS service url is http://20.102.22.31:8010 and whose local plane is "hpcc2data". The local plane must be defined such that it shares the same storage as the remote environment. This is expected to be done via a PVC that has been pre-configured to use the same storage.

To access the logical file in ECL use the following format:

```
ds := DATASET('~remote::hpcc2::somescope1::somelfn1', rec);
```

Azure Managed Identity Authentication

HPCC Systems platform supports Azure managed identities as an authentication method for API file access, allowing the platform to authenticate with Azure storage without requiring explicit storage account keys.

Enable Azure managed identity authentication using the `managed` configuration property option in the `values.schema.json` file which implements the necessary logic to use that managed identity when enabled.

Azure's managed identities improves security by eliminating the need to store sensitive storage account keys in configuration files, instead leveraging Azure's managed identity service for authentication.

Preferred Storage

The *preferredReadPlanes* option is available for each type of cluster--hThor, Thor, and Roxie.

This option is only significant for logical files which reside on multiple storage planes. When specified, the HPCC Systems platform will seek to read files from the preferred plane(s) if a file resides on them. These preferred planes must exist and be defined in *storage.planes*

The following is an example of a Thor cluster configured with the preferredDataReadPlanes option.

```
thor:
- name: thor
  prefix: thor
  numWorkers: 2
  maxJobs: 4
  maxGraphs: 3
  preferredDataReadPlanes:
  - data-copy
  - indexdata-copy
```

In the above example, running a query that reads a file that resides on both "data" and "data-copy" (in that order) normally would read the first copy on "data". With that *preferredDataReadPlanes* specified, if that file also resides on "data-copy", Thor will read that copy.

This can be useful when there are multiple copies of files on different planes with different characteristics that can impact performance.

Storage Items for HPCC Systems Components

HPCC Systems organizes its data into several specialized storage categories, each serving a distinct role within the platform's architecture. These categories—such as *data*, *lz*, *Dali*, *sasha*, *dll*, *spill*, and *temp*—are specified in the plane definition's *category* field. A plane's category determines its intended use (for example, "data" for logical files, "lz" for landing zones, etc.), but does not dictate the underlying storage type or performance characteristics.

Planes can be backed by a variety of storage options—including Persistent Volume Claims (PVCs), Kubernetes storage classes, hosted storage (backed by *daflesrv*), or external services such as Azure Blob or Azure File (via storage APIs). The underlying storage should be selected based on the requirements of each category (e.g., fast random access for indexes, high throughput for bulk data, etc.). Using a Kubernetes storageClass is typically reserved for ephemeral storage and is less common for persistent data.

Data Storage Categories

HPCC Systems uses several general-purpose storage categories, each tailored to specific types of data and performance requirements:

- **data** (*hpccdata*): Primary storage location for general data files. Where the physical data of logical files will be stored.
- **lz** (*Landing Zone*): Landing zones allow external users to read and write files, and HPCC Systems can import from or export to these zones. Performance requirements are typically lower; object storage (e.g., blob/S3 bucket) or NFS mounts may be used.
- **dali:** Location of the Dali metadata store, which requires fast random access.
- **dll:** Stores compiled ECL queries. Storage must support efficient loading of shared objects.

  Note: If you want Dali and dll data on the same plane, they can share the same storage prefix but must use different subpaths.
- **sasha:** Stores archived workunits and similar data. Performance requirements are typically lower, allowing use of lower-cost storage.
- **spill:** (*Optional*) Stores spill files. Local NVMe disks are potentially a good choice for this.
- **temp:** (*Optional*) Stores internal temporary spill files created by the engines. If it is undefined, default to the "spill" category plane. Like the spill plane this should generally be backed by fast storage (like local NVMe). It is also worth noting that the "spill" and "temp" planes are never read remotely, e.g. from other nodes. Unlike the others which need to be accessible globally. This is why local NVMe storage is suitable.

Multiple Device Planes

HPCC Systems storage planes are typically backed by a single Persistent Volume Claim (PVC). If you wanted to increase the overall bandwidth, you can do so using multiple "devices". You can configure a storage plane to use multiple PVCs. It's also possible to achieve the same results with *storageapi.containers* (see *values.schema.json* in the HPCC-Platform repository ~/helm/hpcc).

To configure an HPCC Storage Plane to be backed by multiple PVCs:

- Set the *numDevices* property to the number of PVCs you want to use.

```
- name: data
  pvc: data-mystorage-hpcc-localfile-pvc
```

```
prefix: "/var/lib/HPCCSystems/hpcc-data"
category: data
numDevices: 2
```

The Helm chart will expect to find PVCs named using the base name with numeric suffixes indexed starting with the number 1 (e.g., data-mystorage-hpcc-localfile-pvc-1, data-mystorage-hpcc-localfile-pvc-2). The Helm chart will automatically create multiple mount points, one for each PVC.

- data-mystorage-hpcc-localfile-pvc-1
- data-mystorage-hpcc-localfile-pvc-2

Each PVC will be mounted as a separate device.

Note The Helm chart expects indexing to start at 1. In this example the first PVC must be named data-mystorage-hpcc-localfile-pvc-1 and not data-mystorage-hpcc-localfile-pvc-0.

- Create the necessary storage accounts, Persistent Volumes (PVs), and Persistent Volume Claims (PVCs).
- Define a data plane that references the base name of your PVCs. The base name refers to the common prefix shared by all PVCs in the storage plane. For example, if your PVCs are named data-mystorage-hpcc-localfile-pvc-1 and data-mystorage-hpcc-localfile-pvc-2, the base name would be data-mystorage-hpcc-localfile-pvc.
- Create the PVCs with a common prefix, typically matching the pvc field in your HPCC Systems plane definition, followed by a numeric suffix.

  For example, data-mystorage-hpcc-localfile-pvc-*1*, data-mystorage-hpcc-localfile-pvc-*2*

  The Helm chart will automatically create multiple mount points, one for each PVC.

This configuration enables you to scale your HPCC storage plane by leveraging multiple PVCs, thus improving both capacity and performance.

Logical Partition Striping

HPCC engines, such as Thor, will utilize multiple devices by striping logical file partitions across them. For example, Thor workers will write partitions to different PVCs, enabling parallel data access and improved throughput.

Egress

In order to allow clusters to be securely locked down but still allow access to the services they need there is an egress mechanism. Egress provides a similar mechanism to Ingress, by being able to define which endpoints and ports components are permitted to connect to.

Most HPCC Systems components have their own auto-generated network policies. The generated network policies typically work to limit ingress and egress to inter-component communication, or expected external service ingress only.

For instance, in a default deployed system with network policies enforced, a query running (on hThor, Thor, or Roxie), will not be able to connect with a 3rd party service, such as an LDAP service or log stack.

In the default configuration, any pod with Kube API access will also have access to any external endpoint. This is because a *NetworkPolicy* is generated for egress access for components that need access to the Kube API. However, *global.egress.kubeApiCidr* and *global.egress.kubeApiPort* in the values.yaml should be configured in a secure system to lock this egress access down so that it only exposes egress access to the Kube API endpoint.

We have added a mechanism similar to the visibilities definitions, which allows named egress sections, which can then be referenced per component.

For example:

```
global:
  egress:
    engineEgress:
    - to:
        - ipBlock:
            cidr: 201.13.21.0/24
        - ipBlock:
            cidr: 142.250.187.0/24
      ports:
        - protocol: TCP
          port: 6789
        - protocol: TCP
          port: 7890
...

thor:
...
  egress: engineEgress
```

Note that the name 'engineEgress' is an arbitrary name, any name can be chosen, and any number of these named egress sections can be defined.

For more information, please see the *egress:* section in the default stock/delivered HPCC Systems YAML file. The *values.yaml* file can be found under the helm/hpcc/ directory on the HPCC Systems github repository:

https://github.com/hpcc-systems/HPCC-Platform

Security Values

This section outlines the contents of the *values.yaml* file pertaining to system security components.

TLS Configuration for Platform Components

Configuring TLS for secure communication between platform components in containerized deployments differs significantly from bare-metal installations. Kubernetes environments allow flexibility when configuring for TLS. This section addresses the methods most directly supported by the HPCC Systems Helm chart:

- Cluster mTLS: Mutual TLS for trust between components within a cluster
- Remote mTLS: Establishing a zone of trust with mTLS between components across clusters
- TLS Using Public CA: Use a public Certificate Authority to establish trust between external clients and a publicly-exposed component such as ECL Watch

The cert-manager is used in all these configurations to manage the Kubernetes certificate-related resources. Vault is used as the PKI (Public Key Infrastructure) when establishing trust between components in separate HPCC clusters.

To enable TLS for component make these configuration changes:

- Set the *certificates.enabled* property to *true*
- Set the *enabled* property to *true* for the specific issuer you want to use: local, public, or remote
- Set the component's *visibility* property to the value that assigns it the issuer you want to use: *cluster* visibility for *local* issuer, *local* or *global* visibility for the *public* issuer.
- To use the *remote* issuer, set the component's *visibility* property to *local* or *global*, and its *trustClients* array to the list of remote trusted clients.

  Note: Any SSL references should be taken as colloquial or legacy nomenclature that actually refers to a TLS implementation.

The following examples are to exhibit basic functionality in a testing environment- production deployments may require different settings. The instructions reference paths that are relative to the root of the HPCC Systems Platform repository. Adjust as needed for your environment.

Install cert-manager

A prerequisite to all these examples is installing *cert-manager* to your Kubernetes cluster. To install the cert-manager:

```
helm repo add jetstack https://charts.jetstack.io

helm repo update

helm install cert-manager jetstack/cert-manager --set crds.enabled=true --namespace cert-manager --create-nam
```

Cluster mTLS

This approach requires creating a root certificate and private key for a local cluster certificate authority, then mounting them as Kubernetes secrets for cert-manager to use.

Create a root certificate and private key for our local cluster certificate authority with a single OpenSSL call. This call uses a sample OpenSSL config file found in the HPCC-Platform repository.

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout ca.key -sha256 -days 1825 \
            -out ca.crt -config helm/examples/certmanager/ca-req.cfg
```

The root certificate needs to be added as a Kubernetes secret in order to be accessible to cert-manager. The secret name matches the default name used in the local issuer configuration in values.yaml.

```
kubectl create secret tls hpcc-local-issuer-key-pair --cert=ca.crt --key=ca.key
```

Repeat the root certificate creation process to create the CA secret for the code signing issuer. This is needed to run the ECL testing script at the end to show the example is working.

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout signingca.key -sha256 \
            -days 1825 -out signingca.crt -config helm/examples/certmanager/ca-req.cfg
```

Create a Kubernetes TLS secret from the generated signing root certificate and private key

```
kubectl create secret tls hpcc-signing-issuer-key-pair --cert=signingca.crt --key=signingca.key
```

Install the HPCC Helm chart with the "--set certificates.enabled" option set to true.

```
helm install myhpcc hpcc/hpcc --set certificates.enabled=true
```

The cluster ESPs are now using TLS both locally and publicly. Verify it is working by running an ECL job that requires using mutual TLS (using local client certificate):

```
ecl run --ssl hthor helm/examples/certmanager/localhttpcall.ecl
```

Note that for the HTTPCALL in our ECL example the URL now starts with `mtls:` this tells HTTPCALL/SOAPCALL to use mutual TLS, using the local client certificate, and to verify the server using the local certificate authority certificate.

You should see a result similar to this:

```
          <Result>
            <Dataset name='localHttpEchoResult'>
              <Row>
                <Method>GET</Method>
                <UrlPath>/WsSmc/HttpEcho</UrlPath>
                <UrlParameters>name=doe,joe&number=1</UrlParameters>
                 <Headers>
                          ...
                 </Headers>
              </Row>
            </Dataset>
          </Result>
```

Remote Mutual TLS

This walkthrough demonstrates using a single Hashicorp Vault PKI Certificate authority to establish a zone of trust between two separate HPCC Systems environments. The cert-manager continues to handle the backing Kubernetes resources. For this example each HPCC Systems environment is in a separate Kubernetes namespace: hpcc1 and hpcc2, but in production they could be in different Kubernetes clusters, or even in different cloud subscriptions.

In this example we have hpcc2.roxie making a call to hpcc1.roxie in another environment, and we want them to trust each other. There are two main configuration customizations needed to set up this zone of trust. Each of the two HPCC Systems environments have these changes.

The remote issuer is enabled and its `spec` property is set to use the Vault. Both HPCC Systems environments share the same Vault server, which allows them to use the same underlying certificates to establish trust. Note also that the `ca` property must be set to `null` to override the default.

```
            spec:
              ca: null
              vault:
                path: pki/sign/hpccremote
                server: http://vault.vaultns:8200
                auth:
                  tokenSecretRef:
                      name: cert-manager-vault-token
                      key: token
```

Each Roxie must be told to trust the other. In the hpcc2 values file the Roxie service has its `trustClients` array set to include hpcc1.roxie, and likewise the hpcc1 values file has its Roxie service `trustClients` array set to include hpcc2.roxie. These files can be found in the HPCC-Platform repository at helm/examples/vault-pki-remote/values-hpcc2.yaml and helm/examples/vault-pki-remote/values-hpcc1.yaml. Here are the relevant parts of the hpcc2 values file:

```
            roxie:
            - name: roxie2
              disabled: false
              services:
              - name: roxie2
                servicePort: 29876
                visibility: local
                trustClients:
                - commonName: roxie1.hpcc1
```

Install Hashicorp Vault Service

This scenario is for development only, you would never want to deploy this way in a real world production environment. Deploying in dev mode sets up an in memory kv store that won't persist secret values across restart, and the vault will automatically be unsealed.

In dev mode the default root token is simply the string "root".

Add Hashicorp Helm repository:

```
helm repo add hashicorp https://helm.releases.hashicorp.com

            helm repo update
```

Install Vault server.

When using developer mode Vault you have to set the `VAULT_DEV_LISTEN_ADDRESS` environment variable as shown in order to access the Vault service from an external pod.

```
helm install vault hashicorp/vault  --set "injector.enabled=false" \
              --set "server.dev.enabled=true" \
              --set 'server.extraEnvironmentVars.VAULT_DEV_LISTEN_ADDRESS=0.0.0.0:8200' \
              --namespace vaultns --create-namespace
```

Check the pods:

```
kubectl get pods -n vaultns
```

Vault pods should now be running and ready.

Setting Up Vault

Tell the Vault command line application the server location (dev mode is http, default location is https)

```
export VAULT_ADDR=http://127.0.0.1:8200
```

Export an environment variable for the Vault CLI to authenticate with the Vault server. As mentioned before since we installed dev mode, the Vault token is the string 'root'.

```
export VAULT_TOKEN=root
```

In a separate terminal window start Vault port forwarding.

```
kubectl port-forward vault-0 8200:8200 -n vaultns
```

Login to the Vault command line using the Vault root token (development mode defaults to "root"):

```
vault login root
```

Enable the PKI secrets engine at its default path.

```
vault secrets enable pki
```

Configure the max lease time-to-live (TTL) to 8760h.

```
vault secrets tune -max-lease-ttl=87600h pki
```

Generate the HPCC Systems remote issuer CA, giving it the name *hpcc-remote-issuer*. This name will be used to identify it in the platform configuration.

```
vault write -field=certificate pki/root/generate/internal \
             common_name="hpcc-issuer" issuer_name="hpcc-remote-issuer" ttl=87600h
```

Configure the PKI secrets engine certificate issuing and certificate revocation list (CRL) endpoints to use the Vault service in the "vaultns" namespace.

```
vault write pki/config/urls \
             issuing_certificates="http://vault.vaultns:8200/v1/pki/ca" \
             crl_distribution_points="http://vault.vaultns:8200/v1/pki/crl"
```

For our local mTLS certificates we will use our Kubernetes namespace as our domain name. This will allow us to recognize where these components reside. For our public TLS certificates for this demo we will use example.com as our domain.

Configure a role named hpccremote that enables the creation of certificates in the *hpcc1* and *hpcc2* domains with any subdomains.

```
vault write pki/roles/hpccremote key_type=any allowed_domains="hpcc1,hpcc2" \
             allow_subdomains=true allowed_uri_sans="spiffe://*" max_ttl=72
```

Create a policy named pki that enables read access to the PKI secrets engine paths.

```
vault policy write hpcc-remote-pki - <<EOF
     path "pki*"                   { capabilities = ["read", "list"] }
     path "pki/roles/hpccremote"   { capabilities = ["create", "update"] }
     path "pki/sign/hpccremote"    { capabilities = ["create", "update"] }
     path "pki/issue/hpccremote"   { capabilities = ["create"] }
     EOF
```

Install The First Platform Cluster

Create the *hpcc1* namespace.

```
kubectl create namespace hpcc1
```

The local and signing issuers are isolated and won't be using Vault. Create the unique secrets for both issuers in the *hpcc1* namespace.

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout hpcc1local.key -sha256 -days 1825 -out hpcc1local.crt -conf
```

```
kubectl create secret tls hpcc-local-issuer-key-pair --cert=hpcc1local.crt --key=hpcc1local.key -n hpcc1

openssl req -x509 -newkey rsa:2048 -nodes -keyout hpcc1signing.key -sha256 -days 1825 -out hpcc1signing.crt -
kubectl create secret tls hpcc-signing-issuer-key-pair --cert=hpcc1signing.crt --key=hpcc1signing.key -n hpcc
```

The remote issuer does use Vault. Create the secret the remote issuer that hpcc1 will use to access the Vault pki engine

```
kubectl create secret generic cert-manager-vault-token --from-literal=token=root -n hpcc1

            helm install myhpcc hpcc/hpcc --values helm/examples/vault-pki-remote/values-hpcc1.yaml -n hpcc1
```

Use kubectl to check the status of the deployed pods. Wait until all pods are running before continuing.

```
kubectl get pods -n hpcc1
```

Check and see if the certificate issuers have been successfully created:

```
kubectl get issuers -o wide
```

Install The Second Platform Cluster

Create the *hpcc2* namespace

```
kubectl create namespace hpcc2
```

The local and signing issuers are isolated and won't be using Vault. Create the unique secrets for both issuers in the *hpcc2* namespace.

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout hpcc2local.key -sha256 -days 1825 -out hpcc2local.crt -conf
kubectl create secret tls hpcc-local-issuer-key-pair --cert=hpcc2local.crt --key=hpcc2local.key -n hpcc2

openssl req -x509 -newkey rsa:2048 -nodes -keyout hpcc2signing.key -sha256 -days 1825 -out hpcc2signing.crt -
kubectl create secret tls hpcc-signing-issuer-key-pair --cert=hpcc2signing.crt --key=hpcc2signing.key -n hpcc
```

The remote issuer does use Vault. Create the secret the remote issuer that hpcc1 will use to access the Vault pki engine

```
kubectl create secret generic cert-manager-vault-token --from-literal=token=root -n hpcc2

            helm install myhpcc hpcc/hpcc --values helm/examples/vault-pki-remote/values-hpcc2.yaml -n hpcc2
```

Use `kubectl` to check the status of the deployed pods. Wait until all pods are running before continuing.

```
kubectl get pods -n hpcc2
```

Check and see if the certificate issuers have been successfully created:

```
kubectl get issuers -o wide
```

ECL Example Demonstrating Trust

Now publish ECL to each environment that will communicate securely across the environments.

roxie_echo.ecl which returns a dataset passed into it. remote_echo.ecl which calls roxie_echo.ecl.

For this example we will:

1. publish roxie_echo.ecl to the hpcc1 namespace. It returns the dataset passed into it.
2. Publish remote_echo.ecl to the hpcc2 namespace. It calls roxie_echo.ecl in the hpcc1 namespace using remote mTLS. This is accomplished by using the `remote-mtls:` prefix in the URL to signal that the call should be made using the certificate associated with the `remote` issuer.

3. Use hpcc2::remote_echo.ecl to call hpcc1::roxie_echo.ecl.

Publish the queries:

```
ecl publish roxie1 --ssl --port 18010 roxie_echo.ecl
ecl publish roxie2 --ssl --port 28010 remote_echo.ecl
```

Call the query and demonstrate trust

You can navigate to EclQueries/WsEcl on port 28002 in your browser and run the `remote_echo` query from there. You should see results similar to the following:

```
{"remote_echoResponse":  {"sequence": 0, "Results":  {"remoteResult": {"Row": [{"Dataset": {"Row": [{"name":
```

TLS Using Public CA

The HPCC Systems Helm chart templates support using a Public CA issuer such as Let's Encrypt or ZeroSSL through cert-manager, but most of the work is external to configuring the platform itself and is out of scope for this document. This section outlines the configuration changes needed in the platform to make use of a Public CA issuer setup outside the platform deployment process.

Assume you're using Let's Encrypt as your Public CA issuer, and you've named it *letsencrypt-prod*. Then set the platform's public issuer to use that name:

```
            certificates:
              issuers:
                public:
                  name: letsencrypt-prod
                  kind: ClusterIssuer
```

Note: `ClusterIssuer` is usually preferred for Public CA configurations to centralize certificate management across all namespaces in a cluster and help avoid hitting Public CA usage limits.

Troubleshooting

Common issues and solutions when configuring SSL/TLS for ESP in containerized deployments:

Certificate Not Found — Verify the certificate secret exists in the correct namespace and contains both 'tls.crt' and 'tls.key' keys. Use **kubectl describe secret <secret-name>** to inspect.

Permission Denied — Ensure certificate files are readable by the hpcc user in the container. Consider using an init container to set proper ownership and permissions.

cert-manager Issues — Check cert-manager logs and Certificate resource status. Ensure DNS is properly configured for ACME challenges when using Let's Encrypt.

Certificates

HPCC Systems containerized deployments support comprehensive certificate management through Helm chart configuration. This includes support for multiple certificate issuers, domain management, and advanced certificate features for secure communication between components.

The certificate system integrates with cert-manager to provide automated certificate generation and renewal for both internal (local, i.e., communication between HPCC components within the same cluster or namespace) and external (public/remote) communications.

Enable Certificates

Use the certificates section to enable cert-manager to generate TLS certificates for each component in the HPCC Systems deployment.

```
certificates:
  enabled: false
  issuers:
    local:
      name: hpcc-local-issuer
```

In the delivered YAML file certificates are not enabled, as illustrated above. You must first install the cert-manager to use this feature.

Certificate Issuers Configuration

Certificate issuers are configured in the `certificates.issuers` section of the Helm values. The system supports multiple issuer types:

- **local**: For internal component communication
- **public**: For external-facing services
- **remote**: For remote client connections
- **signing**: For code signing certificates

Alternative Domains Support

The `certificates.issuers.remote` and `certificates.issuers.public`sections support an `alternativeDomains` array configuration option. This feature allows certificates to be valid for multiple domains in addition to the primary domain specified in the issuer configuration.

Configuration Syntax

Each entry in the `alternativeDomains` array represents an additional domain that will be included in the `dnsNames` property of the generated Certificate resource manifest.

Example 1. Remote Issuer with Alternative Domains

```
certificates:
  enabled: true
  issuers:
    remote:
      name: remote-issuer
      kind: ClusterIssuer
      domain: example.com
      alternativeDomains:
        - subdomain1.example.com
        - subdomain2.example.com
        - api.example.org
        - secure.mycompany.net
```

In this configuration:

- The primary domain is `example.com`
- Certificates will also be valid for all domains listed in `alternativeDomains`
- The generated Certificate resource will include all domains in its `dnsNames` section

Certificate Generation Process

When the Helm chart is deployed with certificate generation enabled, the system automatically creates Certificate resources based on the issuer configuration. For remote issuers with alternative domains:

1. **Domain Collection**: The system collects the primary domain and all alternative domains from the issuer configuration
2. **DNS Names Generation**: Service names are combined with each domain to create comprehensive DNS name lists
3. **Certificate Creation**: cert-manager generates certificates valid for all specified domain combinations
4. **Secret Storage**: Generated certificates are stored in Kubernetes secrets for use by HPCC components

Generated Certificate Properties

Certificates generated with alternative domains will contain:

Subject Alternative Names (SAN)	All combinations of service names with the primary domain and alternative domains
Common Name	Typically set to the service name combined with the primary domain
Validity Period	Default 90-day validity with automatic renewal before expiration
Usage	Configured for both server authentication and client authentication

Use Cases and Best Practices

Multi-Domain Certificate Management

The `alternativeDomains` feature is the preferred method for configuring certificates that need to be valid across multiple domains and subdomains. Common use cases include:

- **Multi-tenant Deployments**: Supporting multiple customer domains from a single HPCC deployment
- **Load Balancer Integration**: Certificates valid for both direct service access and load balancer endpoints
- **Development and Production**: Single certificates valid across multiple environment domains
- **Legacy Migration**: Supporting both old and new domain names during migration periods

Configuration Best Practices

1. **Domain Validation**: Ensure all domains in `alternativeDomains` are properly configured in DNS before certificate generation
2. **Certificate Authority Limits**: Be aware of rate limits imposed by certificate authorities when using multiple domains
3. **Security Considerations**: Only include domains that should legitimately be covered by the same certificate
4. **Renewal Planning**: Monitor certificate renewal processes when using external certificate authorities

Complete Configuration Example

Below is a comprehensive example showing how to configure multiple issuers with alternative domain support:

Example 2. Complete Certificate Configuration

```
certificates:
  enabled: true
  issuers:
    local:
      name: local-ca-issuer
      kind: Issuer
      domain: hpcc.local

    public:
      name: letsencrypt-prod
      kind: ClusterIssuer
      domain: hpcc.example.com

    remote:
      name: remote-ca-issuer
      kind: ClusterIssuer
      domain: api.example.com
      alternativeDomains:
        - client.example.com
        - secure.example.com
        - legacy.oldcompany.net
        - backup.example.org

    signing:
      name: code-signing-issuer
      kind: Issuer
      domain: signing.internal
```

This configuration enables:

- Local inter-component communication with internal certificates

- Public-facing services with Let's Encrypt certificates
- Remote client certificates valid across multiple domains
- Code signing capabilities for ECL applications

Troubleshooting Certificates

Certificate Generation Issues

If certificates are not generating correctly with alternative domains:

1. Verify that cert-manager is properly installed and running
2. Check that all domains resolve properly via DNS
3. Examine Certificate and CertificateRequest resources for error messages
4. Validate issuer configuration and permissions

Common Error Messages

Watch for these common issues when using alternative domains:

DNS validation failures	Ensure all alternative domains have proper DNS records and are accessible from the certificate authority
Rate limit exceeded	Some certificate authorities impose limits on the number of domains per certificate or certificates per timeframe
Invalid domain format	Verify that all domains in alternativeDomains follow proper DNS naming conventions

Migration from Legacy Configuration

If you are currently using individual domain configurations or multiple certificate resources, consider migrating to the `alternativeDomains` approach for simplified management:

1. Identify all domains currently covered by separate certificates
2. Update the remote issuer configuration to include all domains in the `alternativeDomains` array
3. Deploy the updated configuration
4. Verify that the new certificate covers all required domains
5. Remove old certificate configurations once the new setup is validated

Note

The `alternativeDomains` configuration is now the preferred method for multi-domain certificate management in HPCC Systems containerized deployments. This approach simplifies certificate management and reduces the overhead of maintaining multiple certificate resources.

Secrets

The secrets section makes sensitive information published as Kubernetes secrets available to the platform in a structured way without exposing it in code or configuration files. At the top level is a set of categories,

where the category determines what the secret is and which components can use it. Under each category is a list of secrets of the format: `platform-name: k8s-published-name` where `platform-name` is the name of the secret as it is referred to in the platform, and `k8s-published-name` is the name of the secret as it is published to the Kubernetes cluster.

For example, the category named "eclUser" is for placing secrets that you want to be readable directly from ECL code. Other secret categories, including the "ecl" category, are read internally by system components and not exposed directly to ECL code.

Vaults

The Vaults section is an alternative to the Secrets section. It serves the same purpose and shares the same top-level categories, but uses a Hashicorp Vault server to store the secrets instead of Kubernetes.

Cross Origin Resource Handling

Cross-origin resource sharing (CORS) is a mechanism for integrating applications in different domains. CORS defines how client web applications in one domain can interact with resources in another domain. You can configure CORS support settings in the ESP section of the values.yaml file as illustrated below:

```
esp:
- name: eclwatch
  application: eclwatch
  auth: ldap
  replicas: 1
  # The following 'corsAllowed' section is used to configure CORS support
  #   origin - the origin to support CORS requests from
  #   headers - the headers to allow for the given origin via CORS
  #   methods - the HTTP methods to allow for the given origin via CORS
  #
  corsAllowed:
  # origin starting with https will only allow https CORS
  - origin: https://*.example2.com
    headers:
    - "X-Custom-Header"
    methods:
    - "GET"
  # origin starting with http will allow http or https CORS
  - origin: http://www.example.com
    headers:
    - "*"
    methods:
    - "GET"
    - "POST"
```

Visibilities

The visibilities section can be used to set labels, annotations, and service types for any service with the specified visibility.

Replicas and Resources

Other noteworthy values in the charts that have bearing on HPCC Systems set up and configuration.

Replicas

replicas: defines how many replica nodes come up, how many pods run to balance a load. To illustrate, if you have a 1-way Roxie and set replicas to 2 you would have 2, 1-way Roxies.

Resources

Most all components have a resources section which defines how many resources are assigned to that component. In the stock delivered values files, the resources: sections are there for illustration purposes only, and are commented out. Any cloud deployment that will be performing any non-trivial function, these values should be properly defined with adequate resources for each component, in the same way you would allocate adequate physical resources in a data center. Resources should be set up in accordance with your specific system requirements and the environment you would be running them in. Improper resource definition can result in running out of memory and/or Kubernetes eviction, since the system could use unbound amounts of resources, such as memory, and nodes will get overwhelmed, at which point Kubernetes will started evicting pods. Therefore if your deployment is seeing frequent evictions, you may want to adjust your resource allocation.

```
  #resources:
  #  cpu: "1"
  #  memory: "4G"
```

Every component should have a resources entry, but some components such as Thor have multiple resources. The manager, worker, eclagent components all have different resource requirements.

Environment Values

You can define environment variables in a YAML file. The environment values are defined under the *global.env* portion of the provided HPCC Systems values.yaml file. These values are specified as a list of name value pairs as illustrated below.

```
global:
  env:
  - name: SMTPserver
    value: mysmtpserver
```

The global.env section of the supplied values.yaml file adds default environment variables for all components. You can also specify environment variables for the individual components. Refer to the schema for setting this value for individual components.

To add environment values you can insert them into your customization configuration YAML file when you deploy your containerized HPCC Systems.

Environment Variables for Containerized Systems

There are several settings in environment.conf for bare-metal systems. While many environment.conf settings are not valid for containers, some can be useful. In a cloud deployment, these settings are inherited from environment variables. These environment variables are configurable using the values yaml either globally, or at the component level.

Some of those variables are available for container and cloud deployments and can be set using the Helm chart. The following bare-metal environment.conf values have these equivalent values which can be set for containerized instances.

Environment.conf Value	Helm Environment Variable
skipPythonCleanup	SKIP_PYTHON_CLEANUP
jvmlibpath	JAVA_LIBRARY_PATH
jvmoptions	JVM_OPTIONS
classpath	CLASSPATH

The following example sets the environment variable to skip Python cleanup on the Thor component:

```
thor:
  env:
  - name: SKIP_PYTHON_CLEANUP
    value: true
```

Index Build Plane

Define the *indexBuildPlane* value as a helm chart option to allow index files to be written by default to a different data plane. Unlike flat files, index files have different requirements. The index files benefit from quick random access storage. Ordinarily flat files and index files are output to the defined default data plane(s). Using this option you can define that index files are built on a separate data plane from other common files. This chart value can be supplied at a component or global level.

For example, adding the value to a global level under globlal.storage :

```
global:
  storage:
    indexBuildPlane: myindexplane
```

Optionally, you could add it at the component level, as follows:

```
thor:
- name: thor
  prefix: thor
  numWorkers: 2
  maxJobs: 4
  maxGraphs: 2
  indexBuildPlane: myindexplane
```

When this value is set at the component level it would override the value set at the global level.

Pods and Nodes

One of the key features of Kubernetes is its ability to schedule pods on to nodes in the cluster. A pod is the smallest and simplest unit in the Kubernetes environment that you can create or deploy. A node is either a physical or virtual "worker" machine in Kubernetes.

The task of scheduling pods to specific nodes in the cluster is handled by the kube-scheduler. The default behavior of this component is to filter nodes based on the resource requests and limits of each container in the created pod. Feasible nodes are then scored to find the best candidate for the pod placement. The scheduler also takes into account other factors such as pod affinity and anti-affinity, taints and tolerations, pod topology spread constraints, and the node selector labels. The scheduler can be configured to use these different algorithms and policies to optimize the pod placement according to your cluster's needs.

You can deploy these values either using the values.yaml file or you can place into a file and have Kubernetes instead read the values from the supplied file. See the above section *Customization Techniques* for more information about customizing your deployment.

Placements

Placements is a term used by HPCC Systems, which Kubernetes refers to as the scheduler or scheduling/assigning. In order to avoid confusion within the HPCC Systems and ECL specific scheduler terms, refer to Kubernetes scheduling as placements. Placements are a value in an HPCC Systems configuration which is at a level above items, such as the nodeSelector, Toleration, Affinity and Anti-Affinity, and TopologySpreadConstraints.

The placement is responsible for finding the best node for a pod. Most often placement is handled automatically by Kubernetes. You can constrain a Pod so that it can only run on particular set of Nodes.

Placements would then be used to ensure that pods or jobs that want nodes with specific characteristics are placed on those nodes.

For instance a Thor cluster could be targeted for machine learning using nodes with a GPU. Another job may want nodes with a good amount more memory or another for more CPU.

Using placements you can configure the Kubernetes scheduler to use a "pods" list to apply settings to pods.

For example:

```
placements:
  - pods: [list]
    placement:
      <supported configurations>
```

Placement Scope

Use pod patterns to apply the scope for the placements.

The pods: [list] item can contain one of the following:

Type: <component>	Covers all pods/jobs under this type of component. This is commonly used for HPCC Systems components. For example, the *type:thor* which will apply to any of the Thor type components; thoragent, thormanager, thoragent and thorworker, etc.
Target: <name>	The "name" field of each component, typical usage for HPCC Systems components referrs to the cluster name. For example *Roxie: -name: roxie* which

	will be the "Roxie" target (cluster). You can also define multiple targets with each having a unique name such as "roxie", "roxie2", "roxie-web" etc.
Pod: <name>	This is the "Deployment" metadata name from the name of the array item of a type. For example, "eclwatch-", "mydali-", "thor-thoragent" using a regular expression is preferred since Kubernetes will use the metadata name as a prefix and dynamically generate the pod name such as, eclwatch-7f4dd4d-d44cb-c0w3x.
Job name:	The job name is typically a regular expression as well, since the job name is generated dynamically. For example, a compile job compile-54eB67e567e, could use "compile-" or "compile-.*" or "^compile-.*$"
All:	Applies for all HPCC Systems components. The default placements for pods delivered is [all]

Regardless of the order the placements appear in the configuration, they will be processed in the following order: "all", "type", "target", and then "pod"/"job".

Mixed combinations

NodeSelector, taints and tolerations, and other values can all be placed on the same pods: [list] both per zone and per node on Azure

```
placements:
- pods: ["eclwatch","roxie-workunit","^compile-.*$","mydali"]
  placement:
    nodeSelector:
      name: npone
```

Node Selection

In a Kubernetes container environment, there are several ways to schedule your nodes. The recommended approaches all use label selectors to facilitate the selection. Generally, you may not need to set such constraints; as the scheduler usually does reasonably acceptable placement automatically. However, with some deployments you may want more control over specific pods.

Kubernetes uses the following methods to choose where to schedule pods:

- nodeSelector field matching against node labels
- Affinity and anti-affinity
- Taints and Tolerations
- nodeName field
- Pod topology spread constraints
- Scheduler name

Node Labels

Kubernetes nodes have labels. Kubernetes has a standard set of labels used for nodes in a cluster. You can also manually attach labels which is recommended as the value of these labels is cloud-provider specific and not guaranteed to be reliable.

Adding labels to nodes allows you to schedule pods to nodes or groups of nodes. You can then use this functionality to ensure that specific pods only run on nodes with certain properties.

The nodeSelector

The nodeSelector is a field in the Pod specification that allows you to specify a set of node labels that must be present on the target node for the Pod to be scheduled there. It is the simplest form of node selection constraint. It selects nodes based on the labels, but it has some limitations. It only supports one label key and one label value. If you wanted to match multiple labels or use more complex expressions, you need to use node Affinity.

Add the nodeSelector field to your pod specification and specify the node labels you want the target node to have. You must have the node labels defined in the job and pod. Then you need to specify each node group the node label to use. Kubernetes only schedules the pod onto nodes that have the labels you specify.

The following example shows the nodeSelector placed in the pods list. This example schedules "all" HPCC components to use the node pool with the label group: "hpcc".

```
placements:
  - pods: ["all"]
    placement:
      nodeSelector:
        group: "hpcc"
```

Note: The label group:hpcc matches the node pool label:hpcc.

This next example shows how to configure a node pool to prevent scheduling a Dali component onto this node pool labelled with the key spot: via the value false. As this kind of node is not always available and could get revoked therefore you would not want to use the spot node pool for Dali components. This is an example for how to configure a specific type (Dali) of HPCC Systems component not to use a particular node pool.

```
placements:
  - pods: ["type:dali"]
    placement:
      nodeSelector:
        spot: "false"
```

When using nodeSelector, multiple nodeSelectors can be applied. If duplicate keys are defined, only the last one prevails.

Taints and Tolerations

Taints and Tolerations are types of Kubernetes node constraints also referred to by node Affinity. Only one affinity can be applied to a pod. If a pod matches multiple placement 'pods' lists, then only the last affinity definition will apply.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints. Taints are the opposite -- they allow a node to repel a set of pods. One way to deploy using taints, is to set to repel all but a specific node. Then that pod can be scheduled onto another node that is tolerate.

For example, Thor workers should all be on the appropriate type of VM. If a big Thor job comes along – then the taints level repels any pods that attempt to be scheduled onto a node that does not meet the requirements.

For more information and examples of our Taints, Tolerations, and Placements please review our developer documentation:

https://github.com/hpcc-systems/HPCC-Platform/blob/master/helm/hpcc/docs/placements.md

Taints and Tolerations Examples

The following examples illustrate how some taints and tolerations can be applied.

Kubernetes can schedule a pod on to any node pool without a taint. In the following examples Kubernetes can only schedule the two components to the node pools with these exact labels, group and gpu.

```
placements:
  - pods: ["all"]
    tolerations:
      key: "group"
        operator: "Equal"
          value: "hpcc"
            effect: "NoSchedule"
```

```
placements:
   - pods: ["type:thor"]
     tolerations:
        key: "gpu"
        operator: "Equal"
        value: "true"
        effect: "NoSchedule"
```

Multiple tolerations can also be used. The following example has two tolerations, group and gpu.

```
#The settings will be applied to all thor pods/jobs and myeclccserver pod and job
- pods: ["thorworker-", "thor-thoragent", "thormanager-","thor-eclagent","hthor-"]
  placement:
    nodeSelector:
      app: tf-gpu
    tolerations:
    - key: "group"
      operator: "Equal"
      value: "hpcc"
      effect: "NoSchedule"
    - key: "gpu"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
```

In this example the nodeSelector is preventing the Kubernetes scheduler from deploying any/all to this node pool. Without taints the scheduler could deploy to any pods onto the node pool. By utilizing the nodeSelector, the taint will force the pod to deploy only to the pods who match that node label. There are two constraints then, in this example one from the node pool and the other from the pod.

Topology Spread Constraints

You can use topology spread constraints to control how pods are spread across your cluster among failure-domains such as regions, zones, nodes, and other user-defined topology domains. This can help to achieve high availability as well as efficient resource utilization. You can set cluster-level constraints as a default, or configure topology spread constraints for individual workloads. The Topology Spread Constraints **topologySpreadConstraints** requires Kubernetes v1.19+.or better.

For more information see:

https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/ and

https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/

Using the "topologySpreadConstraints" example, there are two node pools which have "hpcc=nodepool1" and "hpcc=nodepool2" respectively. The Roxie pods will be evenly scheduled on the two node pools.

After deployment you can verify by issuing the following command:

```
kubectl get pod -o wide | grep roxie
```

The placements code:

```
- pods: ["type:roxie"]
  placement:
    topologySpreadConstraints:
    - maxSkew: 1
      topologyKey: hpcc
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          roxie-cluster: "roxie"
```

Affinity and Anti-Affinity

Affinity and anti-affinity expands the types of constraints that you can define. The affinity and anti-affinity rules are still based on the labels. In addition to the labels, they provide rules that guide Kubernetes' scheduler where to place pods based on specific criteria. The affinity/anti-affinity language is more expressive than simple labels and gives you more control over the selection logic.

The are two main kinds of affinity, Node Affinity and Pod Affinity.

Node Affinity

Node affinity is similar to the nodeSelector concept that allows you to constrain which nodes your pod can be scheduled onto based on the node labels. These are used to constrain the nodes that can receive a pod by matching labels of those nodes. Node affinity and anti-affinity can only be used to set positive affinities that attract pods to the node. These are used to constrain the nodes that can receive a pod by matching labels to those nodes. Node affinity and anti-affinity can only be used to set positive affinities that attract pods to the node.

There is no schema check for the content of affinity. Only one affinity can be applied to a pod or job. If a pod/job matches multiple placement pods lists, then only the last affinity definition applies.

For more information, see https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

There are two types of node affinity:

requiredDuringSchedulingIgnoredDuringExecution: The scheduler can't schedule the pod unless this rule is met. This function is similar to the nodeSelector, but with a more expressive syntax.

preferredDuringSchedulingIgnoredDuringExecution: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the pod.

You can specify node affinities using the *.spec.affinity.nodeAffinity* field in your pod spec.

Pod Affinity

Pod affinity or Inter-Pod Affinity is used to constrain the nodes that can receive a pod by matching the labels of the existing pods already running on to those nodes. Pod affinity and anti-affinity can be either an attracting affinity or a repelling anti-affinity.

Inter-Pod Affinity works very similarly to Node Affinity but have some important differences. The "hard" and "soft" modes are indicated using the same *requiredDuringSchedulingIgnoredDuringExecution* and *preferredDuringSchedulingIgnoredDuringExecution* fields. However, these should be nested under the

spec.affinity.podAffinity or *spec.affinity.podAntiAffinity* fields depending on whether you want to increase or reduce the Pod's affinity.

Affinity Example

The following code illustrates an example of affinity:

```
- pods: ["thorworker-.*"]
  placement:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/e2e-az-name
              operator: In
              values:
              - e2e-az1
              - e2e-az2
```

In the following schedulerName section the, the "affinity" settings can also be included with that example.

Note: The "affinity" value in the "schedulerName" field is only supported in Kubernetes 1.20.0 beta and later versions.

schedulerName

The **schedulerName** field specifies the name of the scheduler that is responsible for scheduling a pod or a task. In Kubernetes, you can configure multiple schedulers with different names and profiles to run simultaneously in the cluster.

Only one "schedulerName" can be applied to any pod/job.

A schedulerName example:

```
- pods: ["target:roxie"]
  placement:
    schedulerName: "my-scheduler"
#The settings will be applied to all thor pods/jobs and myeclccserver pod and job
- pods: ["target:myeclccserver", "type:thor"]
  placement:
    nodeSelector:
      app: "tf-gpu"
    tolerations:
    - key: "gpu"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
```

Helm and Yaml Basics

This section is intended to provide some basic information to help you in getting started with your HPCC Systems containerized deployment. There are numerous resources available for learning about Kubernetes, Helm, and YAML files. For more information about using these tools, or for cloud or container deployments, please refer to the respective documentation.

In the previous section, we touched on the *values.yaml* file and the *values-schema.json* file. This section expands on some of those concepts and how they might be applied when using the containerized version of the HPCC Systems platform.

The *values.yaml* File Structure

The *values.yaml* file is a YAML file which is a format frequently used for configuration files. The construct that makes up the bulk of a YAML file is the key-value pair, sometimes referred to as a dictionary. The key-value pair construct consists of a key that points to some value(s). These values are defined by the schema.

In these configuration files the indentation used to represent document structure relationship is quite important. Leading spaces are significant and tabs are not allowed.

YAML files are made up mainly of two types of elements: dictionaries and lists.

Dictionary

Dictionaries are collections of key-value mappings. All keys are case-sensitive and the indentation is also crucial. These keys must be followed by a colon (:) and a space. Dictionaries can also be nested.

For example:

```
    logging:
       detail: 80
```

This is an example of a dictionary for logging.

Dictionaries in passed in values files, such as in the *myoverrides.yaml* file in the example below, will be merged into the corresponding dictionaries in the existing values, starting with the default values from the delivered hpcc helm chart.

```
helm install myhpcc hpcc/hpcc -f myoverrides.yaml
```

Any pre-existing values in a dictionary that are not overridden will continue to be present in the merged result. However, you can override the contents of a dictionary by setting it to null.

Lists

Lists are groups of elements beginning at the same indentation level starting with a - (a hyphen and a space). Every element of the list is indented at the same level and starts with a hyphen and a space. Lists can also be nested, and they can even be lists of dictionaries.

An example of a list of dictionaries, with placement.tolerations as a nested list.:

```
placements:
- pods: ["all"]
  placement:
    tolerations:
    - key: "kubernetes.azure.com/scalesetpriority"
```

The list entry here is denoted using the hyphen, which is an entry item in the list, which itself is a dictionary with nested attributes. Then the next hyphen (at that same indentation level) is the next entry in that list. A list can be a list of simple value elements, or the elements can themselves be lists or dictionaries.

Sections of the HPCC Systems Values.yaml

The first section of the *values.yaml* file describes global values. Global applies generally to everything.

```
# Default values for hpcc.
global:
  # Settings in the global section apply to all HPCC components in all subcharts
```

In the delivered HPCC Systems *values.yaml* file excerpt (above) *global:* is the top level dictionary. As noted in the comments, the settings in the global section apply to all HPCC Systems components. Note from the indentation level that the other values are nested in that global dictionary.

Items defined in the global section are shared between all components.

Some examples of global values in the delivered values.yaml file are the storage and security sections.

```
storage:
  planes:
```

and also

```
security:
  eclSecurity:
    # Possible values:
    # allow - functionality is permitted
    # deny - functionality is not permitted
    # allowSigned - functionality permitted only if code signed
    embedded: "allow"
    pipe:  "allow"
    extern: "allow"
    datafile: "allow"
```

In the above examples, *storage:* and *security:* are global chart values.

HPCC Systems values.yaml File Usage

The HPCC Systems *values.yaml* file is used by the Helm chart to control how HPCC Systems is deployed. The stock delivered HPCC Systems *values.yaml* is intended as a quick start type installation guide which is not appropriate for non-trivial practical usage. You should customize your deployment to one more suited towards your specific needs.

Further information about customized deployments is covered in previous sections, as well as the Kubernetes and Helm documentation.

Merging and Overriding

Having multiple YAML files, such as one for logging, another for storage, yet another for secrets and so forth, allows granular configuration. These configuration files can all be under version control. There they can be versioned, checked in, etc. and have the benefit of only defining/changing the specific area required, while ensuring any non-changing areas are left untouched.

The rule here to keep in mind where multiple YAML files are applied, the later ones will always overwrite the values in the earlier ones. They are always merged in sequence of the order they are specified on the helm command line.

Another point to consider, where there is a global dictionary such as root: and its value is redefined in a 2nd file (as a dictionary) it would not be overwritten. You simply cannot overwrite a dictionary. You can redefine a dictionary and set it to null, which will effectively wipe out the first.

WARNING: If you had a global definition (such as storage.planes) and merge it where that becomes redefined it would wipe out every definition in that list.

Another means to wipe out every value in a list is to pass in an empty set denoted by a [ ] such as this example:

```
bundles: []
```

This would wipe out any properties defined for bundles.

Generally Applicable

These items are generally applicable for our HPCC Systems Helm YAML files.

- All names should be unique.
- All prefixes should be unique.
- Services should be unique.
- YAML files are merged in sequence.

Regarding the HPCC Systems components, primarily the components are lists. If you have an empty value list denoted by [ ], it would invalidate that list elsewhere.

Overrides

You can add or modify HPCC Systems components by providing override values. You override the Helm chart values by using either override value files with -f or the --set flag to specify a single value. Helm always merges the override values in the order you specify them on the command line.

For example:

```
helm install myhpcc hpcc/hpcc -f myoverrides.yaml
```

Overrides any values in the delivered *values.yaml* by passing in values defined in *myoverrides.yaml*

You can also use --set as in the following example:

```
helm install myhpcc hpcc/hpcc --set storage.daliStorage.plane=dali-plane
```

To override only that one specified value.

It is even possible to combine file and single value overrides, for instance:

```
helm install myhpcc hpcc/hpcc -f myoverrides.yaml --set storage.daliStorage.plane=dali-plane
```

In the preceding example, the --set flag overrides the value for the storage.daliStorage.plane (if) set in the *myoverrides.yaml*, which would override any *values.yaml* file settings and results in setting its value to *dali-plane*.

If the *--set* flag is used on helm install or helm upgrade, those values are simply converted to YAML on the client side.

You can specify the override flags multiple times. The priority will be given to the last (right-most) file specified.

Global/Expert Settings

The 'expert' section under 'global' of the values.yaml should be used to define low-level, testing, or developer settings. This section of the helm chart is intended to be used for custom, low-level or debugging options., therefore in most deployments, it should remain empty.

This is an example of what the global/expert section could look like:

```
global:
  expert:
    numRenameRetries: 3
    maxConnections: 10
    keepalive:
      time: 200
      interval: 75
      probes: 9
```

NOTE: Some components (such as the DfuServer and Thor) also have an 'expert' settings area (see the values schema) that can be used for relevant settings on a per component instance basis, rather than setting them globally.

The following options are currently available:

numRenameRetries (unsigned) If set to a positive number, the platform will re-attempt to perform a rename of a physical file on failure (after a short delay). This should not normally be needed, but on some file systems it may help mitigate issues where the file has just been closed and not exposed correctly at the posix layer.

maxConnections (unsigned) This is a DFU Server setting. If set, it will limit the maximum number of parallel connections and partition streams that will be active at any one time. By default a DFU job will run as many active connection/streams as there are partitions involved in the spray, limited to an absolute maximum of 800. The maxConnections setting can be used to reduce this concurrency. This might be helpful in some scenarios where the concurrency is causing network congestion and degraded performance.

keepalive (time: unsigned, interval: unsigned, probes: unsigned) See keepalive example above. If set, these settings will override the system default socket keepalive settings each time the platform creates a socket. This may be useful in some scenarios if the connections would otherwise be closed prematurely by external factors (e.g., firewalls). An example of this is that Azure instances will close sockets that have been idle for greater than 4 minutes that are connected outside of its networks.

Containerized Logging

Logging Background

Bare-metal HPCC Systems component logs are written to persistent files on local file system, In contrast, containerized HPCC logs are ephemeral, and their location is not always well defined. HPCC Systems components provide informative application level logs for the purpose of debugging problems, auditing actions, and progress monitoring.

Following the most widely accepted containerized methodologies, HPCC Systems component log information is routed to the standard output streams rather than local files. In containerized deployments there aren't any component logs written to files as in previous editions.

These logs are written to the standard error (stderr) stream. At the node level, the contents of the standard error and out streams are redirected to a target location by a container engine. In a Kubernetes environment, the Docker container engine redirects the streams to a logging driver, which Kubernetes configures to write to a file in JSON format. The logs are exposed by Kubernetes via the aptly named "logs" command.

For example:

```
>kubectl logs myesp-6476c6659b-vqckq
>0000CF0F PRG INF 2020-05-12 17:10:34.910 1 10690 "HTTP First Line: GET / HTTP/1.1"
>0000CF10 PRG INF 2020-05-12 17:10:34.911 1 10690 "GET /, from 10.240.0.4"
>0000CF11 PRG INF 2020-05-12 17:10:34.911 1 10690 “TxSummary[activeReqs=22; rcv=5ms;total=6ms;]"
```

It is important to understand that these logs are ephemeral in nature, and may be lost if the pod is evicted, the container crashes, the node dies, etc. Due to the nature of containerized systems, related logs are likely to originate from various locations and need to be collected and processed. It is highly recommended to develop a retention and processing strategy based on your needs.

Many tools are available to help create an appropriate solution based on either a do-it-yourself approach, or managed features available from cloud providers.

For the simplest of environments, it might be acceptable to rely on the standard Kubernetes process which forwards all contents of stdout/stderr to file. However, as the complexity of the cluster grows or the importance of retaining the logs' content grows, a cluster-level logging architecture should be employed.

Cluster-level logging for the containerized HPCC Systems cluster can be accomplished by including a logging agent on each node. The task of each of agent is to expose the logs or push them to a log processing back-end. Logging agents are generally not provided out of the box, but there are several available such as Elasticsearch and Stackdriver Logging. Various cloud providers offer built-in solutions which automatically harvest all stdout/err streams and provide dynamic storage and powerful analytic tools, and the ability to create custom alerts based on log data.

It is your responsibility to determine the appropriate solution to process the streaming log data.

Log Processing Solutions

There are multiple log processing solutions available. You could choose to integrate HPCC Systems logging data with any existing logging solutions, or to implement another one specifically for HPCC Systems data. Starting with HPCC Systems version 8.4, we provide a lightweight, yet complete log-processing solution for convenience. Subsequent sections will look at a couple other possible log-processing solutions.

Log Dependant Applications

Currently there is a utility delivered with a containerized HPCC Systems deployment which is dependant on having a properly configured log-processing solution for optimal results.

The Z.A.P. Utility

The Zipped Analysis Package (Z.A.P.) utility collects system information and encapsulates it into a shareable package. This utility packages up information to send for further analysis. ZAP reports contain several artifacts related to a given workunit, to aid in debugging.

The Component logs are one of most important artifacts expected to be included in the report. In containerized deployments logging is handled differently from bare metal. The log fetching is dependent on a back-end log processor being properly configured and available and the HPCC LogAccess feature configured to bind to the log processor. If those two dependencies are not met, the containerized cluster logs are not included in the ZAP report. Those ZAP reports will then be incomplete. To ensure inclusion of the logs in the ZAP report you must have log access configured properly. See the Containerized Logging sections for more information.

Managed Elastic Stack Solution

HPCC Systems provides a managed Helm chart, *elastic4hpcclogs* which utilizes the Elastic Stack Helm charts for Elastic Search, Filebeats, and Kibana. This chart describes a local, minimal Elastic Stack instance for HPCC Systems component log processing. Once successfully deployed, HPCC component logs produced within the same namespace should be automatically indexed on the Elastic Search end-point. Users can query those logs by issuing Elastic Search RESTful API queries, or via the Kibana UI (after creating a simple index pattern).

Out of the box, the Filebeat forwards the HPCC component log entries to a generically named index: 'hpcc-logs'- <DATE_STAMP> and writes the log data into 'hpcc.log.*' prefixed fields. It also aggregates k8s, Docker, and system metadata to help the user query the log entries of their interest.

A Kibana index pattern is created automatically based on the default filebeat index layout.

Installing the elastic4hpcclogs chart

Installing the provided simple solution is as the name implies, simple and a convenient way to gather and filter log data. It is installed via our helm charts from the HPCC Systems repository. In the HPCC-platform/helm directory, the *elastic4hpcclogs* chart is delivered along with the other HPCC System platform components. The next sections will show you how to install and set up the Elastic stack logging solution for HPCC Systems.

NOTE: The elastic4hpcclogs chart does not enable any security. The responsibility of determining the need for security and enabling security on any deployed Elastic Stack instance or components is up to you and your organization.

Add the HPCC Systems Repository

The delivered Elastic for HPCC Systems chart can be found in the HPCC Systems Helm repository. To fetch and deploy the HPCC Systems managed charts, add the HPCC Systems Helm repository if you haven't done so already:

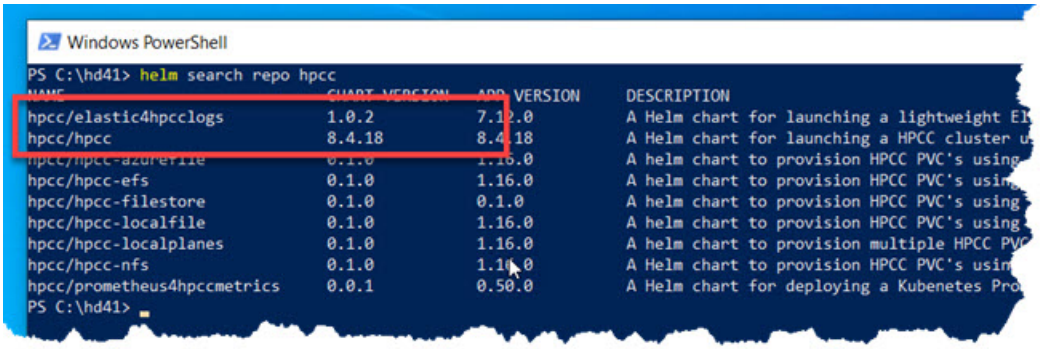
```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

Once this command has completed successfully, the *elastic4hpcclogs* chart will be accessible.

Confirm the appropriate chart was pulled down.

```
helm list
```

Issuing the helm list command will display the available HPCC Systems charts and repositories. The *elastic4hpcclogs* chart is among them.



Install the elastic4hpcc chart

Install the *elastic4hpcclogs* chart using the following command:

```
helm install <Instance_Name> hpcc/elastic4hpcclogs
```

Provide the name you wish to call your Elastic Search instance for the <Instance_Name> parameter. For example, you could call your instance "myelk" in which case you would issue the install command as follows:

```
helm install myelk hpcc/elastic4hpcclogs
```

Upon successful completion, the following message is displayed:

```
Thank you for installing elastic4hpcclogs.
 A lightweight Elastic Search instance for HPCC component log processing.

This deployment varies slightly from defaults set by Elastic, please review the effective values.

PLEASE NOTE: Elastic Search declares PVC(s) which might require explicit manual removal
  when no longer needed.
```

IMPORTANT: PLEASE NOTE: Elastic Search declares PVC(s) which might require explicit manual removal when no longer needed. This can be particularly important for some cloud providers which could accrue costs even after no longer using your instance. You should ensure no components (such as PVCs) persist and continue to accrue costs.

NOTE: Depending on the version of Kubernetes, users might be warned about deprecated APIs in the Elastic charts (ClusterRole and ClusterRoleBinding are deprecated in v1.17+). Deployments based on Kubernetes < v1.22 should not be impacted.

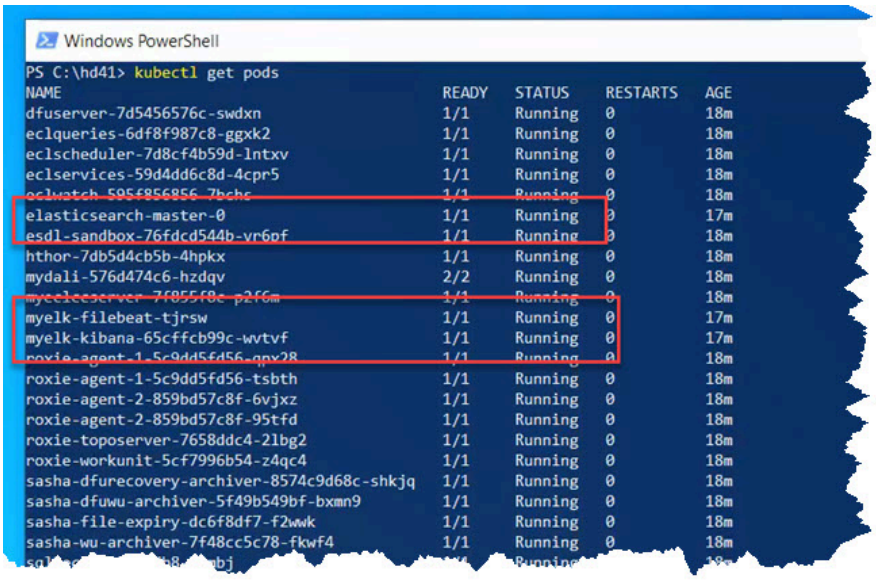
Confirm Your Pods are Ready

Confirm the Elastic pods are ready. Sometimes after installing, pods can take a few seconds to come up. Confirming the pods are in a ready state is a good idea before proceeding. To do this, use the following command:

```
kubectl get pods
```

This command returns the following information, displaying the status of the of the pods.

```
elasticsearch-master-0                  1/1     Running           0
myelk-filebeat-6wd2g                    1/1     Running           0
myelk-kibana-68688b4d4d-d489b           1/1     Running           0
```



Once all the pods are indicating a 'ready' state and 'Running', including the three components for filebeats, Elastic Search, and Kibana (highlighted above) you can proceed.

Confirming the Elastic Services

To confirm the Elastic services are running, issue the following command:

```
$ kubectl get svc
```

This displays the following confirmation information:

```
...
elasticsearch-master ClusterIP 10.109.50.54 <none> 9200/TCP,9300/TCP 68m
elasticsearch-master-headless ClusterIP None <none> 9200/TCP,9300/TCP 68m
myelk-kibana LoadBalancer 10.110.129.199 localhost 5601:31465/TCP 68m
...
```

Note: The myelk-kibana service is declared as LoadBalancer for convenience.

Configuring of Elastic Stack Components

You may need or want to customise the Elastic stack components. The Elastic component charts values can be overridden as part of the HPCC System deployment command.

For example:

```
helm install myelk hpcc/elastic4hpcclogs --set elasticsearch.replicas=2
```

Please see the Elastic Stack GitHub repository for the complete list of all Filebeat, Elastic Search, LogStash and Kibana options with descriptions.

Use of HPCC Systems Component Logs in Kibana

Once enabled and running, you can explore and query HPCC Systems component logs from the Kibana user interface. Kibana index patterns are required to explore Elastic Search data from the Kibana user interface. For more information about using the Elastic-Kibana interface please refer to the corresponding documentation:

https://www.elastic.co/

and

https://www.elastic.co/elastic-stack/

Configuring logAccess for Elasticstack

The *logAccess* feature allows HPCC Systems to query and package relevant logs for various features such as the ZAP report, WorkUnit helper logs, ECLWatch log viewer, etc.

Once the logs are migrated or routed to the elastic stack instance. The HPCC Systems platform needs to be able to access those logs. The way you direct HPCC Systems to do so is by providing a values file that includes the log mappings. We have provided a default values file and we provide an example command line that inserts that values file into your deployment. This suggested configuration values file for enabling log access can be found in the HPCC Systems Platform GitHub repository.

https://github.com/hpcc-systems/HPCC-Platform

Then navigating to the *helm/examples/azure/log-analytics/loganalytics-hpcc-logaccess.yaml* file.

You can use the delivered Elastic4HPCCLogs chart provided or you can add the values there to your customized configuration values yaml file.

You can then install it using a command such as:

```
helm install mycluster hpcc/hpcc -f elastic4hpcclogs-hpcc-logaccess.yaml
```

Azure Log Analytics Solution

Azure Kubernetes Services (AKS) Azure Log Analytics (ALA) is an optional feature designed to help monitor performance and health of Kubernetes based clusters. Once enabled and associated a given AKS with an active HPCC System cluster, the HPCC component logs are automatically captured by Log Analytics. All STDERR/STDOUT data is captured and made available for monitoring and/or querying purposes. As is usually the case with cloud provider features, cost is a significant consideration and should be well understood before implementation. Log content is written to the logs store associated with your Log Analytics workspace.

Enabling Azure Log Analytics

Enable Azure's Log Analytics (ALA) on the target AKS cluster using one of the following options: Direct command line, Scripted command line, or from the Azure portal.

For more detailed information refer to the Azure documentation:

https://docs.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-onboard

Direct Command Line

To enable the Azure Log Analytics insights from the command line:

You can create a dedicated log analytics workspace manually, or you can skip this step and utilize the default workspace instead.

To create a dedicated workspace enter this command:

```
az monitor log-analytics workspace create -g myresourcegroup -n myworkspace --query-access Enabled
```

To enable Log Analytics feature on a target AKS cluster, reference the *workspace resource id* created in the previous step:

```
az aks enable-addons -g myresourcegroup -n myaks -a monitoring --workspace-resource-id  \
 "/subscriptions/xyz/resourcegroups/myresourcegroup/providers/ \
  microsoft.operationalinsights/workspaces/myworkspace"
```

Scripted Command Line

As a convenience, HPCC Systems provides a script to enable ALA (with a dedicated log analytics workspace) on the target AKS cluster.

The *enable-loganalytics.sh* script is located at:

https://github.com/hpcc-systems/HPCC-Platform/tree/master/helm/examples/azure/log-analytics

The script requires populating the following values in the env-loganalytics environment file.

Provide these values in the *env-loganalytics* environment file order to create a new Azure Log Analytics workspace, associate it with a target AKS cluster, and enable the processing of logs:

- **LOGANALYTICS_WORKSPACE_NAME** The desired name for the Azure Log Analytics workspace to be associated with target AKS cluster. A new workspace is created if this value does not exist
- **LOGANALYTICS_RESOURCE_GROUP** The Azure resource group associated with the target AKS cluste. A new workspace will be associated with this resource group.

- **AKS_CLUSTER_NAME** The name of the target AKS cluster to associate the log analytics workspace.
- **TAGS** The tags associated with the new workspace.

  For example: "admin=MyName email=my.email@example.com environment=myenv justification=testing"
- **AZURE_SUBSCRIPTION** [Optional] Ensures this subscription is set before creating the new workspace

Once these values are populated, the *enable-loganalyics.sh* script can be executed and it will create the log analytics workspace and associate it with the target AKS cluster.

Azure Portal

To enable the Azure Log Analytics on the Azure portal:

1. Select Target AKS cluster
2. Select Monitoring
3. Select Insights
4. Enable - choose default workspace

Configure HPCC logAccess for Azure

The *logAccess* feature allows HPCC Systems to query and package relevant logs for various features such as the ZAP report, WorkUnit helper logs, ECLWatch log viewer, etc.

Procure Service Principal

Azure requires an Azure Active Directory (AAD) registered application in order to grant Log Analytics API access. Procure an AAD registered application.

For more information about registering an Azure Active Difrectory see the Azure official documentation:

https://docs.microsoft.com/en-us/power-apps/developer/data-platform/walkthrough-register-app-azure-active-directory

Depending on your Azure subscription structure, it might be necessary to request this from a subscription administrator.

Provide AAD Registered Application Information

HPCC Systems logAccess requires access to the AAD tenant, client, token, and target workspace ID via secure secret object. The secret is expected to be in the 'esp' category, and named *'azure-logaccess*'.

The following key value pairs are supported

- aad-tenant-id
- aad-client-id
- aad-client-secret
- ala-workspace-id

The 'create-azure-logaccess-secret.sh' script provided at:

https://github.com/hpcc-systems/HPCC-Platform/tree/master/helm/examples/azure/log-analytics

The script can be used to create the necessary secret.

Example manual secret creation command (assuming ./secrets-templates contains a file named exactly as the above keys):

```
create-azure-logaccess-secret.sh .HPCC-Platform/helm/examples/azure/log-analytics/secrets-templates/
```

Otherwise, create the secret manually.

Example manual secret creation command (assuming ./secrets-templates contains a file named exactly as the above keys):

```
kubectl create secret generic azure-logaccess \
  --from-file=HPCC-Platform/helm/examples/azure/log-analytics/secrets-templates/
```

Configure HPCC logAccess

The target HPCC Systems deployment should be configured to target the above Azure Log Analytics workspace by providing appropriate logAccess values (such as ./loganalytics-hpcc-logaccess.yaml). The previously created azure-logaccess secret must be declared and associated with the esp category, this can be accomplished via secrets value yaml (such as ./loganalytics-logaccess-secrets.yaml).

Example use:

```
helm install myhpcc hpcc/hpcc \
  -f HPCC-Platform/helm/examples/azure/log-analytics/loganalytics-hpcc-logaccess.yaml
```

Accessing HPCC Systems Logs

The AKS Log Analytics interface on Azure provides Kubernetes-centric cluster/node/container-level health metrics visualizations, and direct links to container logs via "log analytics" interfaces. The logs can be queried via “Kusto” query language (KQL).

See the Azure documentation for specifics on how to query the logs.

Example KQL query for fetching "Transaction summary" log entries from an ECLWatch container:

```
let ContainerIdList = KubePodInventory
| where ContainerName =~ 'xyz/myesp'
| where ClusterId =~ '/subscriptions/xyz/resourceGroups/xyz/providers/Microsoft.
                      ContainerService/managedClusters/aks-clusterxyz'
| distinct ContainerID;
ContainerLog
| where LogEntry contains "TxSummary["
| where ContainerID in (ContainerIdList)
| project LogEntrySource, LogEntry, TimeGenerated, Computer, Image, Name, ContainerID
| order by TimeGenerated desc
| render table
```

Sample output

More complex queries can be formulated to fetch specific information provided in any of the log columns including unformatted data in the log message. The Log Analytics interface facilitates creation of alerts based on those queries, which can be used to trigger emails, SMS, Logic App execution, and many other actions.

Controlling HPCC Systems Logging Output

The HPCC Systems logs provide a wealth of information which can be used for benchmarking, auditing, debugging, monitoring, etc. The type of information provided in the logs and its format is trivially controlled via standard Helm configuration. Keep in mind in container mode, every line of logging output is liable to incur a cost depending on the provider and plan you have and the verbosity should be carefully controlled using the following options.

By default, the component logs are not filtered, and contain the following columns:

```
MessageID TargetAudience LogEntryClass JobID DateStamp TimeStamp ProcessId ThreadID QuotedLogMessage
```

The logs can be filtered by TargetAudience, Category, or Detail Level. Further, the output columns can be configured. Logging configuration settings can be applied at the global, or component level.

Target Audience Filtering

The availble target audiences include operator(OPR), user(USR), programmer(PRO), monitor(MON), audit(ADT), or all. The filter is controlled by the <section>.logging.audiences value. The string value is comprised of 3 letter codes delimited by the aggregation operator (+) or the removal operator (-).

For example, all component log output to include Programmer and User messages only:

```
helm install myhpcc ./hpcc --set global.logging.audiences="PRO+USR"
```

Target Category Filtering

The available target categories include disaster(DIS), error(ERR), information(INF), warning(WRN), progress(PRO), event(EVT), metrics(MET). The category (or class) filter is controlled by the <section>.logging.classes value, comprised of 3 letter codes delimited by the aggregation operator (+) or the removal operator (-).

For example, the mydali instance's log output to include all classes except for progress:

```
helm install myhpcc ./hpcc --set dali[0].logging.classes="ALL-PRO" --set dali[0].name="mydali"
```

Log Detail Level Configuration

Log output verbosity can be adjusted from "critical messages only" (1) up to "report all messages" (100). The default log level is rather high (80) and should be adjusted accordingly.

These are the available log levels:

- CriticalMsgThreshold = 1;
- FatalMsgThreshold = 1;
- ErrMsgThreshold = 10;
- WarnMsgThreshold = 20;
- AudMsgThreshold = 30;

- ProgressMsgThreshold = 50;
- InfoMsgThreshold = 60;
- DebugMsgThreshold = 80;
- ExtraneousMsgThreshold = 90;

For example, to show only progress and lower level (more critical) messages set the verbosity to 50:

```
helm install myhpcc ./hpcc --set global.logging.detail="50"
```

Log Data Column Configuration

The available log data columns include messageid(MID), audience(AUD), class(CLS), date(DAT), time(TIM), node(NOD), millitime(MLT), microtime(MCT), nanotime(NNT), processid(PID), threadid(TID), job(JOB), use(USE), session(SES), code(COD), component(COM), quotedmessage(QUO), prefix(PFX), all(ALL), and standard(STD). The log data columns (or fields) configuration is controlled by the <section>.logging.fields value, comprised of 3 letter codes delimited by the aggregation operator (+) or the removal operator (-).

For example, all component log output should include the standard columns except the job ID column:

```
helm install myhpcc ./hpcc --set global.logging.fields="STD-JOB"
```

Adjustment of per-component logging values can require assertion of multiple component specific values, which can be inconvinient to do via the --set command line parameter. In these cases, a custom values file could be used to set all required fields.

For example, the ESP component instance 'eclwatch' should output minimal log:

```
helm install myhpcc ./hpcc --set -f ./examples/logging/esp-eclwatch-low-logging-values.yaml
```

Asychronous Logging Configuration

By default log entries will be created and logged asynchronously, so as not to block the client that is logging. Log entries will be held in a queue and output on a background thread. This queue has a maximum depth, once hit, the client will block waiting for capacity. Alternatively, the behaviour can be be configured such that when this limit is hit, logging entries are dropped and lost to avoid any potential blocking.

NB: normally it is expected that the logging stack will keep up and the default queue limit will be sufficient to avoid any blocking.

The defaults can be configured by setting <section>.logging.queueLen and/or <section>.logging.queueDrop.

Setting <section>.logging.queueLen to 0, will disabled asynchronous logging, i.e. each log will block until completed.

Setting <section>.logging.queueDrop to a non-zero (N) value will cause N logging entries from the queue to be discarded if the queueLen is reached.

Troubleshooting Containerized Deployments

Introduction

Helm is a powerful package manager for Kubernetes, simplifying the deployment and management of complex applications. However, even with Helm, deployment issues can arise. This chapter will guide you through common troubleshooting steps for Helm deployments. Command-line tools, such as *kubectl* and *helm* are available for both local and cloud deployments.

Useful Helm Commands

Here are some useful Helm commands for troubleshooting.

List deployments using this command in a terminal window:

```
helm list
```

This returns all installed Helm releases.

If you have multiple namespaces, use this command:

```
helm list -A
```

Returns all installed Helm releases across all namespaces.

Get the status of a specific release using this command in a terminal window:

```
helm status <release-name>
```

This returns the status of a specific release.

Get the user supplied values for a release using this command in a terminal window:

```
helm get values <release-name>
```

By effectively using these Helm commands, you can quickly identify and resolve issues with your Helm deployments. Remember to consult the official Helm documentation for more detailed information and specific use cases.

Check the Status of Pods

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. Checking the status of pods is a fundamental step in troubleshooting Kubernetes deployments. By monitoring pod status, you can quickly identify and address potential issues, ensuring the health and performance of your applications.

The HPCC Systems platform has one or more pods for each component of a deployed system.

To get a quick overview of pod status, use the following command in a terminal window:

```
kubectl get pods
```

This lists all pods in your cluster, along with their status, restart count, and other details.

If you have deployments to more than one namespace, use this command:

```
kubectl get pods -A
```

This lists all pods in all namespaces.

Each pod should indicate a status of **Running** and have a matching number of pods displayed in the **READY** column.

Check the **RESTARTS** column, a high number of restarts may indicate issues.

Identifying Other Issues and Their Root Cause

Pending Status:

Insufficient Resources	The pod might be waiting for resources like CPU or memory to become available.
Scheduling Failures	There might be scheduling conflicts or node issues preventing the pod from being scheduled.

Running Status:

High Restart Count	Frequent restarts could indicate issues with the pod's configuration, image, or underlying infrastructure.
Resource Constraints	The pod might be experiencing resource limitations, leading to performance degradation or crashes.

Failed Status:

Container Failures	One or more containers within the pod might have failed due to errors or crashes.
Termination Signals	The pod might have been intentionally terminated, potentially due to a deployment or scaling operation.

Describe a Pod

Use the **kubectl** command-line tool to get detailed information about pods. By describing a pod, you can gain valuable insights into its current state, configuration, and resource utilization.

To get detailed information about a pod, use the following command in a terminal window:

```
kubectl describe pod <pod-name>
```

The output provides detailed information about the pod, including:

Events A timeline of events related to the pod's lifecycle. If there are issues with the deployment, they are commonly found in this section.

Containers Information about the containers running within the pod.

Status The current status of the pod.

Conditions The conditions that the pod must meet to be considered running.

If you have deployments to more than one namespace, use this command:

```
kubectl describe pod <pod-name>  -A
```

This describes the pod across all namespaces.

By carefully analyzing this information, you can:

Identify and troubleshoot issues Pinpoint the root cause of problems, such as resource constraints, configuration errors, or network connectivity issues.

Monitor pod health and performance Track the pod's status, resource usage, and event history to ensure it's operating as expected.

Optimize resource allocation Adjust resource requests and limits to improve performance and cost-efficiency.

Gain insights into Kubernetes scheduling and resource management Learn how Kubernetes allocates resources to pods and handles failures.

By mastering the art of describing pods, you can become a more effective Kubernetes administrator and troubleshoot your deployments with confidence.

Check the Status of Services

Services expose applications running on a cluster.

The *kubectl get services* command is a powerful tool for troubleshooting Kubernetes deployments. It provides a concise overview of the services running in your cluster, helping you identify potential issues and their root causes.

To get a quick overview of services status, use the following command in a terminal window:

```
kubectl get services
```

This lists all services in your cluster, along with their type, internal and external IP addresses, port, and uptime (Age).

If you have deployments to more than one namespace, use this command:

```
kubectl get service -A
```

This lists all services in all namespaces.

If a service that should have an external IP listed does not have one displayed, that pod has an issue.

Describe a Service

Use the **kubectl** command-line tool to get detailed information about a service. By describing a service in Kubernetes, you can gain valuable insights into its configuration, health, and how it interacts with pods.

To get detailed information about a service, use the following command in a terminal window:

```
kubectl describe service <service-name>
```

The output provides detailed information about the service, including the service's IP address, port, selectors, and other details.

If you have deployments to more than one namespace, use this command:

```
kubectl describe service <service-name>  -A
```

This describes the service across all namespaces.

Viewing Pod Logs

Viewing pod logs is a crucial step in troubleshooting Helm deployments because it provides real-time insights into the behavior and errors occurring within your application containers. By analyzing these logs, you can quickly identify and address a wide range of issues.

To view the logs of a specific pod, use the following command in a terminal window:

```
kubectl logs <pod-name>
```

This returns the entire log for a pod.

To tail the logs and see real-time output:

```
kubectl logs -f <pod-name>
```

If the pod has more than one container, use this command to get logs for a specific container:

```
kubectl logs <pod-name> -c <container-name>
```

Copying Files To and From a Pod

The **kubectl cp** command allows you to copy files and directories between your local file system and a container running in a pod. This is especially useful for retrieving log files, core dumps, or configuration files from a pod for deeper analysis, or for uploading data files (such as input data) directly into a pod's drop zone.

To copy a file *from* a pod to your local machine, use the following command in a terminal window:

```
kubectl cp <pod-name>:<path/to/remote/file> <path/to/local/destination>
```

For example, to copy a log file from a pod named *myesp-6476c6659b-vqckq*:

```
kubectl cp myesp-6476c6659b-vqckq:/var/log/myapp.log ./myapp.log
```

To copy a file *to* a pod from your local machine, reverse the source and destination arguments:

```
kubectl cp <path/to/local/file> <pod-name>:<path/to/remote/destination>
```

For example, to upload a data file to the drop zone of a pod named *mydropzone-7d9f8b6c4-xkp2t*:

```
kubectl cp mydata.csv mydropzone-7d9f8b6c4-xkp2t:/var/lib/HPCCSystems/dropzone/mydata.csv
```

If the pod runs multiple containers, specify the target container using the *-c* flag:

```
kubectl cp <pod-name>:<path/to/remote/file> <path/to/local/destination> -c <container-name>
```

If you are working across multiple namespaces, prefix the pod name with the namespace:

```
kubectl cp <namespace>/<pod-name>:<path/to/remote/file> <path/to/local/destination>
```

Note

The *kubectl cp* command requires the *tar* utility to be present inside the container. Most HPCC Systems container images include it by default.

Viewing Service Logs

While services themselves don't produce logs, you can view the logs of the pods that are running the service.

To do this, you'll need to identify the pods that are selected by the service's selector. You can use the describe command to see the selector:

```
kubectl describe service <service-name>
```

Once you know the selector, you can list the pods that match it:

```
kubectl get pods -l <selector-label>
```

Then, you can view the logs of those pods using the logs command:

```
kubectl logs <pod-name>
```

Effective Log Analysis

Here are some additional tips for effective log analysis for troubleshooting.

Filter and Search Logs

Use *kubectl logs* options to filter logs by timestamp, container name, or specific keywords to focus on relevant information.

For example:

```
kubectl logs <pod-name> --since=10m
```

Filters logs from a specific time.

Even though *kubectl logs* doesn't have a direct keyword search option, you can use tools like *grep* to filter the output.

For example:

```
kubectl logs <pod-name> | grep "TLS"
```

Correlate Logs with Metrics

Combine log analysis with monitoring metrics to gain a holistic view of application performance.

Leverage Logging Tools

Consider using advanced logging tools like Elasticsearch, Logstash, and Kibana (ELK Stack) to centralize, aggregate, and analyze logs from multiple pods and services.

Set Appropriate Log Levels

Configure your application to log at the appropriate level of detail, balancing the need for informative logs with the risk of excessive log verbosity. You can then use a filter to show only those log entries that meet the criteria.

Monitor logs in real-time.

Monitoring logs in real-time can be useful to debug ongoing issues. Use the following command:

```
kubectl logs -f
```

Additional Troubleshooting Tips

Here are some additional tips for troubleshooting your deployments.

Check your Helm chart configuration.	Ensure that your Helm chart(s) are configured correctly, with accurate values for images, resources, and environment variables.
Verify Image Availability.	Make sure that the images used in your Helm chart are accessible and can be pulled by Kubernetes.
Inspect Resource Limits and Requests.	Review the resource limits and requests defined for your pods and services. Insufficient resources can lead to performance issues or pod failures.
Examine Kubernetes Logs.	Use the *kubectl logs* command to view the logs of specific pods and containers. These logs can provide valuable insights into errors and unexpected behavior.
Review Network Connectivity.	Ensure that your Kubernetes cluster has proper network connectivity, both internally and externally. Network issues can prevent pods from communicating with each other or with external services.
Consider Persistent Volume Claims (PVCs).	If your application requires persistent storage, verify that PVCs are provisioned correctly and that the underlying storage is accessible.

By following these steps and tips, you can effectively troubleshoot your containerized deployments and quickly identify the root cause of issues.