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.

2022 Version 8.6.22-1

Containerized HPCC Overview

Starting with version 8.0, the HPCC Systems® Platform is focusing 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 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. Future versions may include specifics for other Cloud providers.

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.

Note: The traditional bare-metal version of the HPCC Systems platform is mature and has been heavily used in commercial applications for almost two decades and is fully intended for production use. The containerized version is new and is not yet 100% ready for production. In addition, aspects of that version could change without notice. We encourage you to use it and provide feedback so we can make this version as robust as a bare-metal installation.

Bare-metal vs Containers

If you are familiar with the HPCC Systems platform, 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 helm's 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 eclccserver 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 10	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.

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.

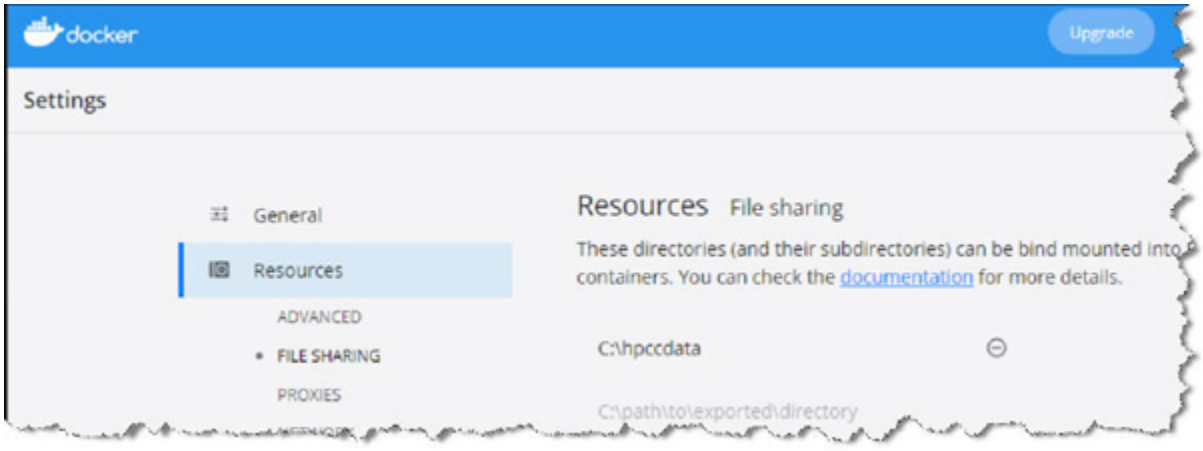
   In this example, we will call the file *mystorage.yaml*. The file should look similar to this:

```
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.

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::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).

Customizing Configurations

Customization Techniques

In this section, we will walk through creating a custom 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.

  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.

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 file.

The *values.yaml* and how it is used

The *values.yaml* file is the delivered configuration values for a Helm chart. The *values.yaml* file is used by the Helm chart to control how HPCC Systems is deployed to the cloud. This values file is one 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 the specific 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, 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.

The schema file typically contains (for a property) a name and a description. It will often include details of the type, and the items it can contain 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 (subpath), 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 of the schema. The schema file should be complete, and contain 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 in 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, 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).

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 effect 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 *maxMemPercentage*. 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.

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.
```

Storage Items for HPCC Systems Components

General Data Storage

General data files generated by HPCC are stored stored in data. For Thor, data storage costs could likely be significant. Sequential access speed is important, but random access is much less so. For ROXIE, speed of random access is likely to be most important.

LZ

LZ or lz, utilized for landing zone data. This is where we would put raw data coming into the system. A landing zone where external users can read and write files. HPCC Systems can import from or export files to a landing zone. Typically performance is less of an issue, it could be blob/s3 bucket storage, accessed either directly or via an NFS mount.

dali

The location of the dali metadata store, which needs to support fast random access.

dll

Where the compiled ECL queries are stored. The storage needs to allow shared objects to be directly loaded from it efficiently.

If you wanted both Dali and dll data on the same plane, it is possible to use the same prefix for both subpath properties. Both would use the same prefix, but should have different subpaths.

sasha

This is the location where archived workunits, etc are stored and it is typically less speed critical, requiring lower storage costs.

spill

An optional category where the spill files are written out to. Local NVMe disks are potentially a good choice for this.

temp

An optional category where temp files can be written to.

The Security Values

This section will look at the *values.yaml* sections dealing with the system security components.

Certificates

The certificates section can be used to enable the 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.

Secrets

The Secrets section contains a set of categories, each of which contain a list of secrets. The Secrets section is where to get info into the system if you don't want it in the source. Such as code with embedded code, you can have that defined in the code sign sections. If you have information that you don't want public but need to run it you could use secrets.

Vaults

Vaults is another way to do Secrets. The vaults section mirrors the secret section but leverages *HashiCorp Vault* for the storage of secrets. There is an additional category for vaults named "ecl-user". The intent of the ecl-user vault secrets is to be readable directly from ECL code. Other secret categories are read internally by system components and not exposed directly to ECL code.

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.

Taints, Tolerations, and placements

This is an important consideration for containerized systems. Taints and Tolerations are types of Kubernetes node constraints also referred to by **Node Affinity**. Node affinity is a way to constrain pods to nodes. 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.

For example, Thor workers should all be on the appropriate type of VM. If a big Thor job comes along – then the taints level comes into play.

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

Placements

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. 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>
```

The pods: [list] can contain a variety of items.

1. HPCC Systems component types, using the prefix *type:* this can be: dali, esp, eclagent, eclccserver, roxie, thor. For example "type:esp"

2. Target; the name of an array item from the above types using prefix "target:" For example "target:roxie" or "target:thor".

3. Pod, "Deployment" metadata name from the name of the array item of a type. For example, "eclwatch", "mydali", "thor-thoragent"

4. Job name regular expression: For example "compile-" or "compile-." or exact match "^compile-.$"

5. All: to apply for all HPCC Systems components. The default placements for pods we deliver is [all]

Placements – in Kubernetes the Placement concept allows you to spread your pods across types of nodes with particular characteristics. Placements would be used to ensure that pods or jobs that want nodes with specific characteristics are placed on them.

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. You can use placements to ensure that pods with specific requirements are placed on appropriate nodes.

More Helm and Yaml

This section is intended to provide some helpful information to get started with a containerized deployment. There are numerous resources for using Kubernetes, Helm, and Yaml files. Previously, 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. For more information about using Kubernetes, Helm, or YAML files, or for cloud or container deployments, refer to the respective documentation.

The *values.yaml* file structure

The *values.yaml* file is a yaml file. Yaml is a data serialization language often used as a format for configuration files. The construct that makes up the bulk of a yaml file is the key-value pair, sometimes referred to as a hash or a dictionary. The key-value pair construct consists of a key that points to some value(s). The values could be strings, numbers, booleans, integers, arrays, or dictionaries, and lists. These values are defined by the schema.

In yaml files the indentation is used to represent document structure and nesting. Leading spaces are significant and tabs are not allowed.

Dictionary

Dictionaries are collections of key value mappings. All keys are case-sensitive and as we mentioned earlier the indentation is also crucial. These keys must be followed by a colon (:) and a space. Dictionaries can also be nested.

Dictionary is a key: value, followed by another key: value:, for example:

```
   logging:
      detail: 80
```

This is an example of a dictionary for logging.

Dictionaries in passed in values files, such as the ones 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
```

Note that you can pass in as many yaml files as you like, they will be merged in the order that they appear on the command line.

Any pre-existing values in a dictionary that are not overridden will continue to be present in the merged result. However, you can delete 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 dash and a space). Every element of the list is indented at the same level and starts with a dash and a space. Lists can also be nested, and they can be lists of dictionaries, which may in turn also have list properties.

An example of a list of dictionaries, with placement.tolerations as a nested list.:

```
placements:
- pods: ["all"]
  placement:
    tolerations:
    - key: "kubernetes.azure.com/scalesetpriority"
```

A key is denoted using a minus sign, which is an entry item in the list, which itself is a dictionary with nested attributes. Then the next minus sign (at that same indentation level) is the next entry in that list.

Global

The first section of the *values.yaml* file describes global values. The global.image.root is a string denoting which version to pull. Global applies generally to everything.

```
# Default values for hpcc.

global:
  # Settings in the global section apply to all HPCC components in all subcharts

  image:
    ## It is recommended to name a specific version rather than latest, for any non-trivial
    ## For best results, the helm chart version and platform version should match - default if version
    ## not specified. Do not override without good reason as undefined behavior may result.
    ## version: x.y.z
    root: "hpccsystems"    # change this to pull from somewhere other than DockerHub hpccsystems
    pullPolicy: IfNotPresent

  # logging sets the default logging information for all components. Can be overridden locally
  logging:
    detail: 80
```

In the delivered HPCC Systems *values.yaml* file excerpt (above) *global:* is a top level dictionary. As noted in the comments, the settings in the global section apply to all HPCC Systems components. Note from the indentation that the other values are nested in that global dictionary.

Image

In our delivered *values.yaml* file the value immediately following global: is *image:* you should use a specific named version rather than using the "latest", as also indicated in the comments in the values file. The Helm chart version

and platform version should match. Ideally you shouldn't have to set the image.version at all. By default it will match the helm chart version.

The Root value

The global dictionary/definition level entry is root. For example

```
  root: "hpccsystems" # change to pull your images somewhere other than DockerHub hpccsystems
```

In *values.yaml* file this uses our HPCC Systems specific repository. It is possible you may want to pull from some other repository, this then is where to set that value.

root: SomeValue

Other Chart Values

Items defined in the global section are shared between all components.

Examples of global values 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.

Usage

The HPCC Systems *values.yaml* file is used by the Helm chart to control how HPCC Systems is deployed. The values file contains dictionaries and lists, and they can be nested to create more complex structures. The stock HPCC Systems *values.yaml* is intended as a quick start demonstration installation guide which is not appropriate for non-trivial practical usage. You should customize your deployment to one which is more suited towards your specific needs. To customize your deployment you override the stock values in the *values.yaml* file, as in the following example:

```
helm install myhpcc hpcc/hpcc -f myoverrides.yaml
```

The above example uses the *myoverrides.yaml* file via the -f parameter, which overrides any specified values in the HPCC Systems *values.yaml* file. It's important to note that this merges the overrides from myoverrides.yaml. Anything that's in the values in the helm chart itself that is not overwritten by the passed in values will remain active. When there are 2 yaml files such as this example (the stock *values.yaml*, and the myoverrides.yaml), if there is a matching entry (anything other than a dictionary) the value from 2nd file will overwrite the first. Dictionaries however will always be merged.

Further information about customized deployments is covered in other sections, as well as the Kubernetes Helm documentation. Consulting the Helm documentation provides complete detail for every aspect of Helm chart usage, and not only for a few select cases described.

Use Case

For instance, you want to update logging detail. You could have another yaml file to update that value, or any other list value using an override yaml file.

As we will see later, components are defined as lists, so any definition of a component in a user values file will replace all instances of the component in the default chart. You can remove all components defined in a list, by replacing the list with a null list, for example,

```
thor: []
```

This will remove all Thor components.

Other options (for instance configuring the costs for cpu or file access) are implemented as a dictionary, so options can be selectively set in a users values file, and the other options will be retained.

Merging and Overriding

Having multiple yaml files, such as one for logging, another for storage, yet another for secrets and so forth, the files can be in version control. 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 merged in in sequence.

Another point to consider, where there is a global dictionary such as root: and its value is redefined in the 2nd file (as a dictionary) it would not be overwritten. You can't simply overwrite a dictionary. You can redefine a dictionary and set it to null (such as the Thor example in the previous section), which will effectively wipe it out.

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 the 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.

Generally regarding the HPCC Systems components, the components are lists. As stated previously, If you have an empty value list [ ], it would invalidate that list elsewhere.

Additional Usage

Components are added or modified by passing in overrides. Chart values are only overridden, either by passing in override values file using -f, (for override file) or via --set where you can override a single value. Those passed in values are always merged in the order they are given on the helm command line.

For example you can

```
helm install myhpcc hpcc/hpcc -f myoverrides.yaml
```

To override any values in the delivered *values.yaml*. Or you can use --set as in the following example:

```
helm install myhpcc hpcc/hpcc --set storage.daliStorage.plane=dali-plane
```

To override only the global.image.version value. Again, the order the values are merged in is the same in which they are issued on the command line. Now consider:

```
helm install myhpcc hpcc/hpcc -f myoverrides.yaml --set storage.daliStorage.plane=dali-plane
```

In the preceding example, the --set flag in the above command overrides the value for the storage.daliStorage.plane (if) set in the myoverrides.yaml, which overrides any *values.yaml* file settings and results in setting it to dali-plane. So, irrespective of the value in the yaml file for this particular setting, the order specified on the command line overwrites it in the order supplied on the command line.

command line options

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 -f flag multiple times. The priority will be given to the last (right-most) file specified.

```
$ helm install myhpcc hpcc/hpcc -f myvalues.yaml -f override.yaml
```

For the above example, if both myvalues.yaml and override.yaml contained a key called 'Test', the value set in override.yaml would take precedence.

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. Also, due to the nature of containerized solutions, related logs are likely to originate from various locations and might 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 backend. 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 available log processing solutions. You could choose to integrate HPCC Systems logging data with any of your 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 your convenience. As stated there are several possible solutions, you should choose the option that best meets your requirements. The following sections will look at two possible solutions.

The Elastic4hpcclogs chart

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.

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:

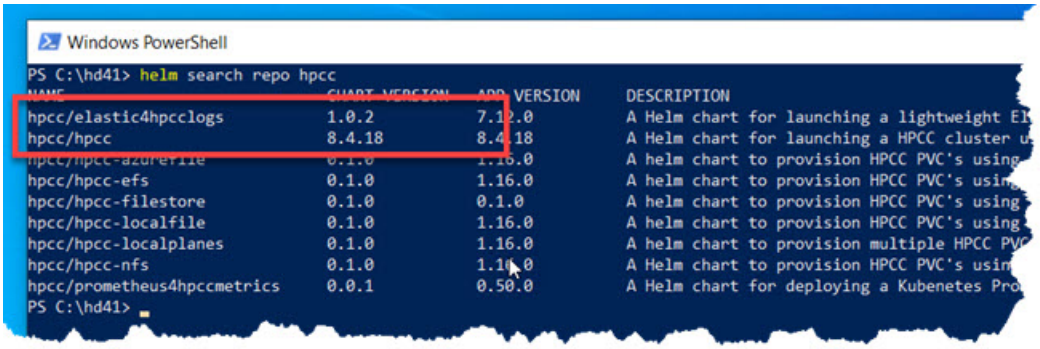
```
helm repo add hpcc https://hpcc-systems.github.io/helm-chart/
```

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.

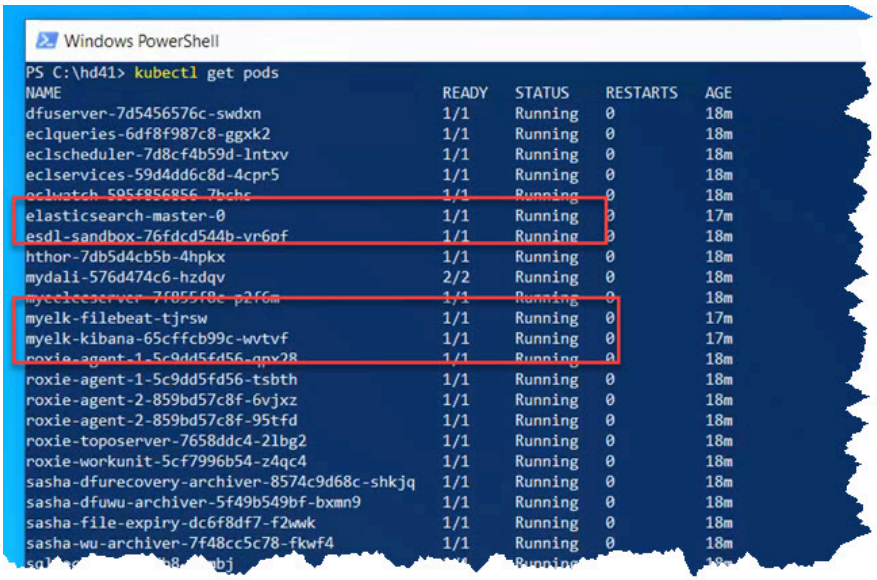
Confirm Your Pods are Ready

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. Using the Kibana interface is well supported and documented. Kibana index patterns are required to explore Elastic Search data from the Kibana user interface. Elastic provides detailed explanations of the information required to understand and effectively utilize the Elastic-Kibana interface. Kibana's robust documentation, should be referred to for more information about using the Kibana interface. Please see:

https://www.elastic.co/

and

https://www.elastic.co/elastic-stack/

Included among the complete documentation are also quick start videos and other helpful resources.

Azure AKS Insights

Azure AKS Insights 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 Insights. 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 Insights

Enabling Azure's Insights on the target AKS cluster can be done from the Azure portal or via CLI. For detailed Azure documentation: Enable Container insights:

https://docs.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-onboard

Azure Portal

To enable the Azure insights on the Azure portal:

1. Select Target AKS cluster
2. Select Monitoring
3. Select Insights
4. Enable - choose default workspace

Command Line

To enable the Azure insights from the command line:

Optionally, create log-analytics workspace [default workspace otherwise]

Enter:

```
az monitor log-analytics workspace create -g myresourcegroup -n myworkspace --query-access Enabled
```

Enable on target AKS cluster (reference the workspace resource id from 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"
```

The AKS Insights 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 Insights 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), 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), 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.

For example, verbosity should be medium for all components:

```
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
```