



BalticLSC Platform Technical Documentation

The design of the BalticLSC Platform
Version 1.0



EUROPEAN
REGIONAL
DEVELOPMENT
FUND

Priority 1: Innovation

Warsaw University of Technology, Poland
RISE Research Institutes of Sweden AB, Sweden
Institute of Mathematics and Computer Science, University of Latvia, Latvia
EurA AG, Germany
Municipality of Vejle, Denmark
Lithuanian Innovation Center, Lithuania
Machine Technology Center Turku Ltd., Finland
Tartu Science Park Foundation, Estonia

BalticLSC Platform Technical Documentation

Design of the BalticLSC Platform

Work package	WP4
Task id	A4.3
Document number	O4.3
Document type	Technical document
Title	BalticLSC Platform Technical Documentation
Subtitle	Design of the BalticLSC Platform
Author(s)	Daniel Olsson (RISE), Magnus Nilsson-Mäki (RISE)
Reviewer(s)	Agris Šostaks (IMCS)
Accepting	Michał Śmiałek (WUT)
Version	1.0
Status	Final version

History of changes

Date	Ver.	Author(s)	Review	Change description
10.10.2019	0.01	Daniel Olsson (RISE)		Document creation and initial contents
23.10.2019	0.02	Daniel Olsson (RISE)		Saved document to dropbox
25.10.2019	0.1	Daniel Olsson (RISE)	Jonas, Anki and Magnus	Updated with inputs from reviews
12.11.2019	1.0	Daniel Olsson (RISE)	Agris	Updated with inputs from reviews

Executive summary

The overall aim for the Baltic LSC activities is developing and providing a platform for truly affordable and easy to access LSC Environment for end-users to significantly increase capacities to create new innovative data-intense and computation-intense products and services by a vast array of smaller actors in the region.

Current Platform Technical Documentation is an important output of the Baltic LSC project, based on technical workshops held during 2019 with the participation of external experts and led mainly by the project's technology partners from:

- Warsaw University of Technology (WUT)
- RISE Research Institutes of Sweden AB (RISE)
- IMCS University of Latvia (IMCS)

The document is also based on input obtained during Baltic LSC project's international and local workshops as well as individual meetings with experts, potential cooperation partners and end-users from March – June 2019.

It is supporting the Environment Vision Document (Baltic LSC Output 3.1), which is providing an overall vision of the Baltic LSC Environment and is complemented by Baltic LSC Software Architectural Vision (Baltic LSC Output 5.1) describing in detail the hardware visions for the platform.

This document contains an architectural model of the BalticLSC Platform. It contains decisions on which technologies (computing and networking hardware, operating system software, computing languages) should be. Also, it will propose optimal solutions for combining selected hardware units into coherent computation grids and networking solutions that would allow combining such small grids into large computation networks on a transnational level. It contains design details that is to be used as input when developing the BalticLSC Platform.

Table of Contents

History of changes.....	2
Executive summary	3
Table of Contents	4
1. Introduction	6
1.1 Objectives and scope	6
1.2 Relations to Other Documents.....	6
1.3 Intended Audience and Usage Guidelines.....	6
2. Requirements.....	7
2.1 REST API.....	7
2.2 Docker container support	7
2.3 Resource usage tracking.....	7
2.4 User isolation and quota	7
3. System Overview	8
3.1 Kubernetes.....	9
3.2 BalticLSC Platform Manager	9
3.2.1 Platform Manager.....	10
3.2.2 Rancher.....	11
4. Installation and Setup	12
4.1 Production cluster configuration	12
4.1.1 Management cluster.....	12
4.1.2 Compute cluster.....	13
4.1.3 Operating system requirements	13
4.1.4 Development/minimal cluster configuration.....	14
4.2 Installation of Rancher and Kubernetes.....	14
4.3 BalticLSC settings in Rancher.....	15
4.3.1 Custom security policy	15
4.3.2 Create user with associated project and resource quota	16
5. System Design.....	19
5.1 Hardware resources and pricing	19
5.1.1 Central Processing Unit - CPU.....	19
5.1.2 Memory	19
5.1.3 Storage.....	19
5.1.4 Network.....	19
5.1.5 Graphical Processing Unit - GPU.....	20
5.2 Users, projects and quota.....	20
5.2.1 One BalticLSC Software user	21
5.3 Network isolation	21

5.4	Resource monitoring	21
5.4.1	Cluster monitoring settings.....	22
5.5	Billing.....	23
5.5.1	Utilization reports for billing.....	23
5.6	API usage specifics	24
5.6.1	Project id reference when creating namespace.....	24
5.6.2	Resource quotas on namespaces.....	24
5.6.3	Container resource requests and limits	24
5.7	Priority and pre-emption.....	24
5.7.1	Using PriorityClasses	25
5.8	New software components	25
5.8.1	Namespace controller	25
5.8.2	Ingress rule conflict controller.....	26
5.8.3	Extension API-server serving user resources	26
5.8.4	GPU quota	26
6.	Use cases	27

1. Introduction

1.1 Objectives and scope

The BalticLSC Platform is where computation tasks compiled by the BalticLSC Software are to be executed. The scope of this document is to give the reader insights to the workings of the Platform. From hardware procurement and installation to actual deployment and management.

1.2 Relations to Other Documents

The current BalticLSC Platform Technical Documentation Document is supporting the Environment Vision (BalticLSC Output 3.1), which is providing an overall vision of the BalticLSC Environment and is complemented by BalticLSC Software Architectural Vision (BalticLSC Output 5.1) describing in detail the main concepts, components and software development technologies of the system. The document is supporting all requirements in BalticLSC Platform User Requirements (BalticLSC Output 3.2).

1.3 Intended Audience and Usage Guidelines

This document is the Output 4.3 as described in the BalticLSC Application Form and intended for internal use within BalticLSC consortium as the basis for future software development activities as well as for reporting purposes for local First Level Control and Baltic Sea Region Program.

2. Requirements

2.1 REST API

There must exist a way to programmatically use the compute resources of the Platform. A REST API must therefore be provided which will be used by the users of the Platform. The most obvious user is BalticLSC Software.

2.2 Docker container support

During the workshops we decided that Docker¹ container should be used as the execution platform.

A container is a standard unit of software that packages up the code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, run-time, system tools, system libraries and settings.

Container images become containers at run-time and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker is the de facto standard when it comes to containers and Kubernetes comes with out of the box support for it. These are the reasons why BalticLSC should run Docker containers.

2.3 Resource usage tracking

To enable business models, hardware resource usage tracking is mandatory. This means that the amount of resources (CPU, GPU, RAM, etc.) that a docker container consumes during its lifetime must be tracked and stored.

2.4 User isolation and quota

Users of the platform must be isolated from each other. This means that one user shall not be able to affect another user in any way. The Platform must also support resource quotas and limits. This means that it must be possible to specify how much resources (see above) a user can allocate, as well as controlling that the user does not use more than requested.

¹ <https://docker.com>

3. System Overview

The BalticLSC Platform consists of several different components which are described in this section.

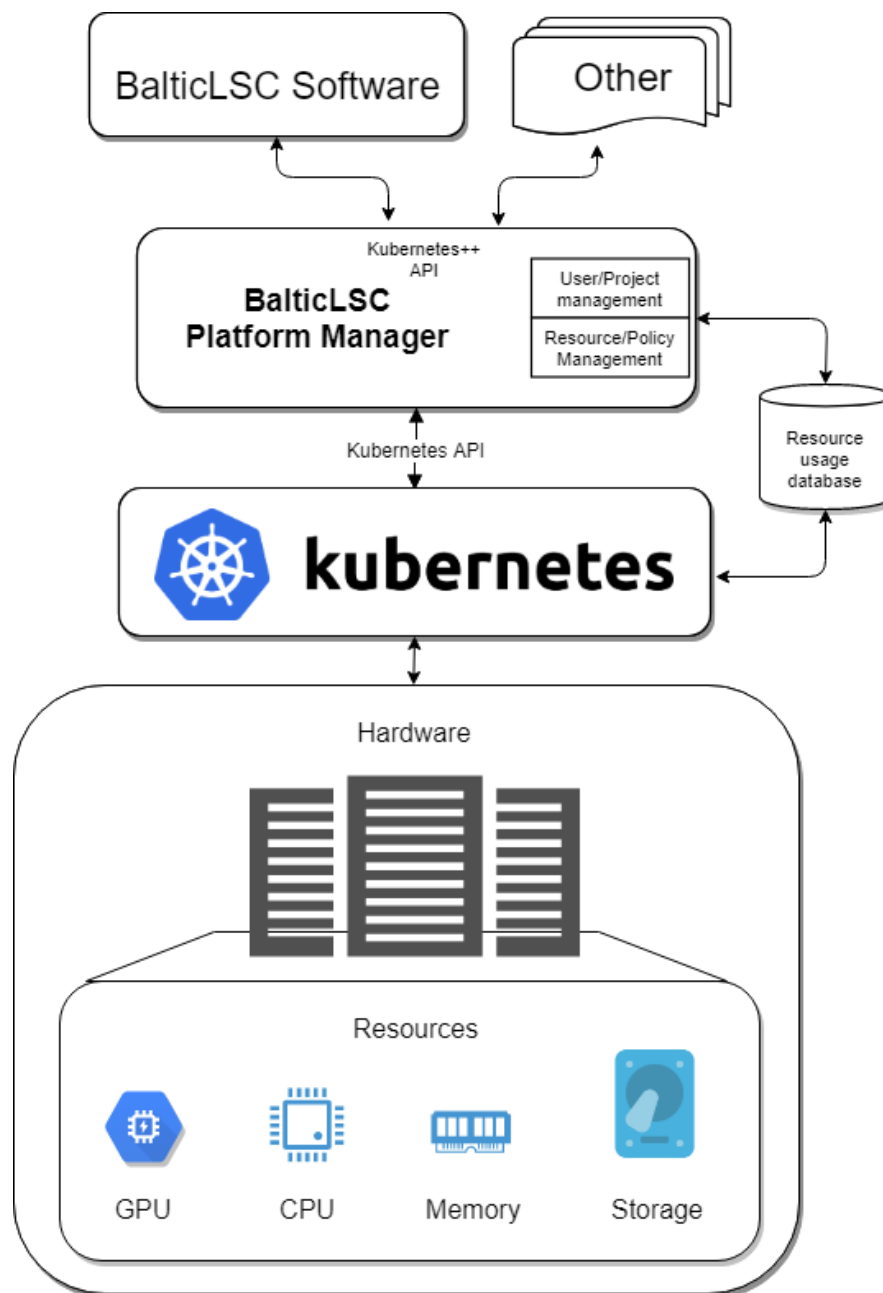


Figure 1: BalticLSC Platform Architecture

3.1 Kubernetes

Kubernetes (K8s)² is an open-source system for automating deployment, scaling, and management of containerized applications. Several different container orchestration platforms were evaluated before Kubernetes was chosen. Following platforms were evaluated:

- Kubernetes
- Docker Swarm
- Apache Mesos
- OpenShift (RedHat)
- DC/OS
- Amazon ECS
- Nomad
- Cattle

This evaluation was done in 2017 and following google trends graph shows that Kubernetes popularity is only growing.

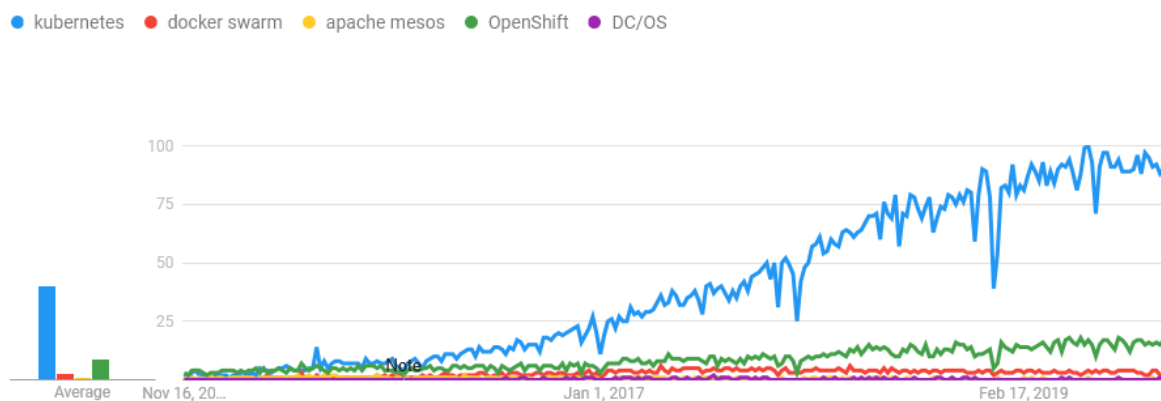


Figure 2: Google trends

One Kubernetes cluster is composed of one or more Nodes. They can be virtual machines or physical servers. Each node has one or several roles: ETCD, control-plane or worker. Nodes assigned the ETCD and control-plane roles become part of the Kubernetes management and control plane. Nodes assigned the worker role will run the workloads. Workloads consist of Pods which are groups of one or more containers with shared storage, network and a specification for how to run the containers.

Kubernetes fulfills following requirements:

- 2.1 REST API – the Kubernetes REST API
- 2.2 Docker Container Support – It is the native execution environment
- 2.3 User isolation and quota – supported by Kubernetes

3.2 BalticLSC Platform Manager

The BalticLSC Platform Manager fulfills requirement 2.3 *Resource Usage Tracking* and makes fulfilling requirement 2.4 *User isolation and quota* easier using a GUI.

² <https://kubernetes.io>

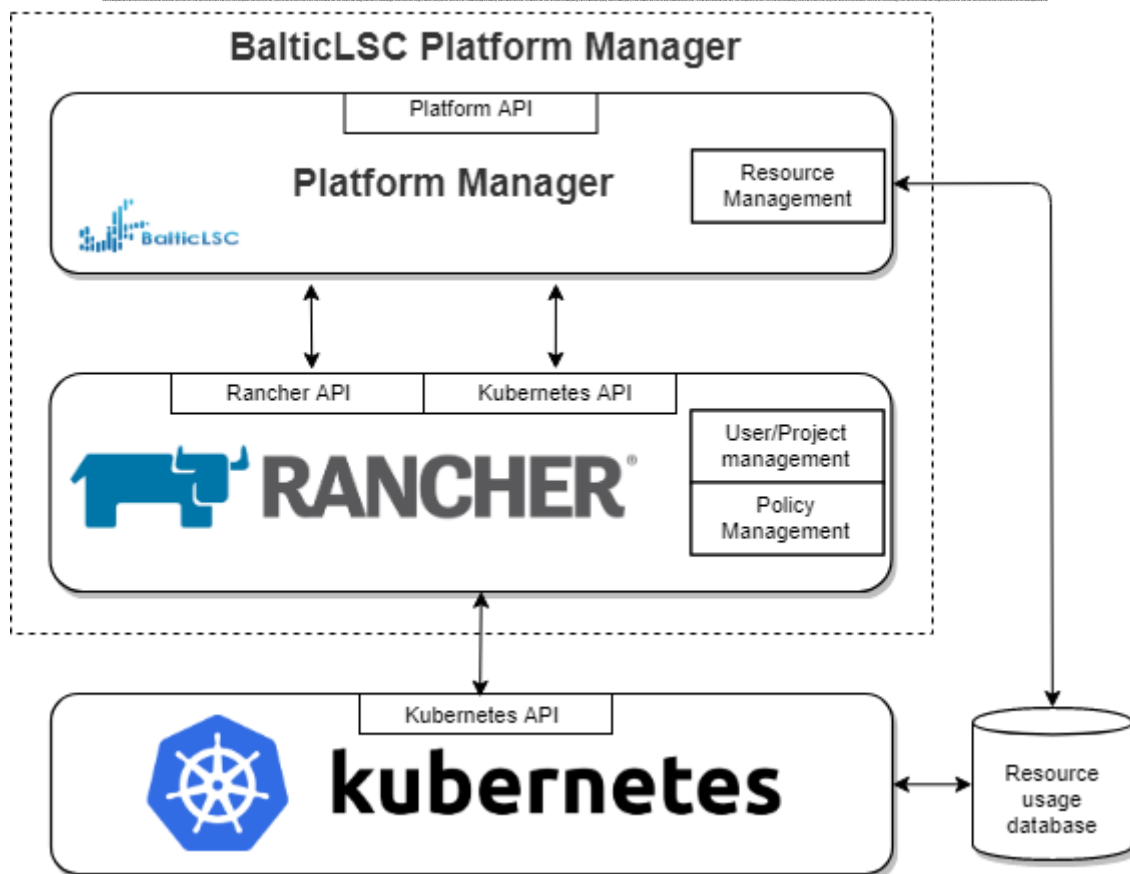


Figure 3: BalticLSC Platform Manager

3.2.1 Platform Manager

The BalticLSC Platform Manager consists of two components.

1. Platform Manager
2. Rancher (see section 3.2.2)

The Platform Manager consists of components implementing all functions not directly supported by Rancher or Kubernetes that is required by The BalticLSC Platform. The custom functions are described in section 5. Its main purpose is to provide API access to the resource usage database. This is done via the Platform API which is the native Kubernetes API with additions for querying resource usage data.

The Platform Manager is a new component that will be developed. In this document we suggest that it should be written as an addition to Rancher in the language Golang. The reason for this is that Kubernetes is built as a framework for building any application. In fact, rancher itself is built upon Kubernetes. Therefore, it is suggested to build also the BalticLSC Platform upon the Kubernetes development framework.

3.2.2 Rancher



Figure 4: Rancher

Rancher is a platform for Kubernetes management. Here is a short description from their website³:

Rancher is a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters while providing DevOps teams with integrated tools for running containerized workloads.

Following list of features is what made us choose rancher:

- Easy Kubernetes deployment
- Easy Cluster Operation & Workload Management
- Very nice web GUI
- User and project management, authentication and policing
- Multi-Cluster support
- 100% Free and Open Source Software with Native Upstream Kubernetes

It comes with support for many of the requirements that BalticLSC needs. It has the project concept, where one project can include one or more namespace. It also comes shipped with security templates for network policies and more.

Rancher also uses annotations to add new namespaces to a project. So, these need to be injected. This is discussed later in this document. It also comes with two templates for pod security policies, restricted and unrestricted. When deploying a new cluster with Rancher pod security policy support can be enabled under advanced options in the "Deploy Cluster" wizard.

³ <https://rancher.com>

4. Installation and Setup

This section describes two different configurations. One for small-scale and development purposes, and one for large-scale production environments.

4.1 Production cluster configuration

The production cluster consists of two Kubernetes clusters. One running Rancher and BalticLSC Platform components, called the Management Cluster. The other cluster is where all workloads are executed, called the Compute Cluster.

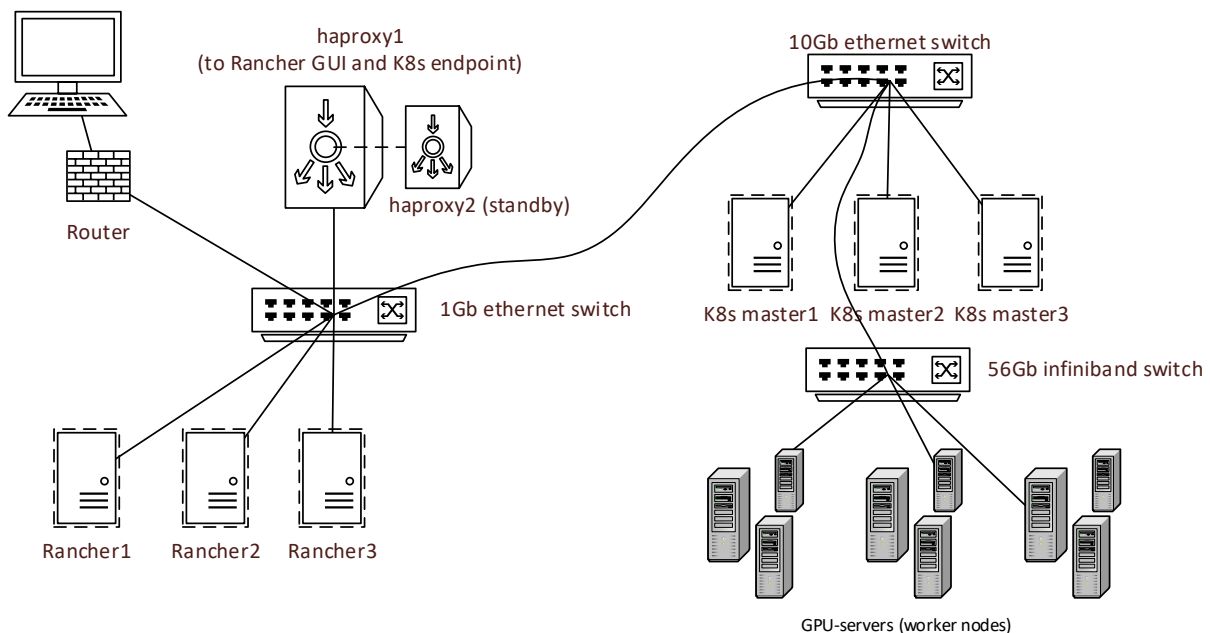


Figure 5: Production cluster topology

Imagine that the figure above is splitted vertically in the middle. Then the left side represents the Management Cluster with Rancher and BalticLSC Platform components. Everything to the right is then the Compute Cluster.

4.1.1 Management cluster

Rancher is running in its own Kubernetes cluster of three nodes. To provide load-balancing to the rancher nodes, a HA proxy can be used. This should also be configured in a highly available configuration. Details of this is omitted in this document. BalticLSC Platform specific components is also running in this cluster.

Hardware requirements on management nodes scale based on the total size of the clusters managed by Rancher. Provision each individual node according to the requirements in following table (taken from Rancher website⁴). If nodes are VMs, they need to run on separate physical hosts.

Deployment size	Clusters	Nodes	vCPUs	RAM
Small	Up to 5	Up to 50	2	8 GB
Medium	Up to 15	Up to 200	4	16 GB

⁴ <https://rancher.com/docs/rancher/v2.x/en/installation/requirements/>

Large	Up to 50	Up to 500	8	32 GB
X-Large	Up to 100	Up to 1000	32	128 GB

Table 1: Rancher node requirements in high-availability configuration

4.1.2 Compute cluster

It is recommended to use Rancher to install the compute cluster. The compute cluster consists of at least three master nodes and one or more worker nodes. Installation instructions is found later in this document.

Following server specifications can be used as inspiration when writing procurements.

Master node / Low performance compute node

- 1U server
- 2x CPU
- 32GB RAM
- SSD
- 10Gb Ethernet

Medium performance compute node

- 2U server
- 2 x CPU
- 128 GB RAM
- 1 x Nvidia GPU
- SSD
- 10Gbps Ethernet

High-performance GPU compute node

- Supermicro 4029GP-TRT3 server chassis
- Single root PCIe architecture
- 2 x Intel Xeon Gold 6130
- 256 GB RAM
- 4TB SSD
- 8 x Nvidia GTX 2080ti
- Infiniband 56 Gbps

4.1.3 Operating system requirements

Rancher is tested on the following operating systems and their subsequent non-major releases with a supported version of Docker⁵.

- Ubuntu 16.04 (64-bit x86)
- Docker 17.03.x, 18.06.x, 18.09.x
- Ubuntu 18.04 (64-bit x86)
- Docker 18.06.x, 18.09.x
- Red Hat Enterprise Linux (RHEL)/CentOS 7.6 (64-bit x86)
- RHEL Docker 1.13
- Docker 17.03.x, 18.06.x, 18.09.x

⁵ <https://docker.com>

- RancherOS 1.5.1 (64-bit x86)
- Docker 17.03.x, 18.06.x, 18.09.x

The recommendation is to run Ubuntu or CentOS because that is what we are experienced with. This concerns both Rancher and Compute nodes.

4.1.4 Development/minimal cluster configuration

For development and test there are almost no requirements on the hardware. It is possible to run everything on a standard consumer laptop. When running on laptop, and a cluster of several nodes is wanted, then each node should run in its own virtual machine. Choose your favorite hypervisor for running VMs, like KVM, VirtualBox or VMWare.

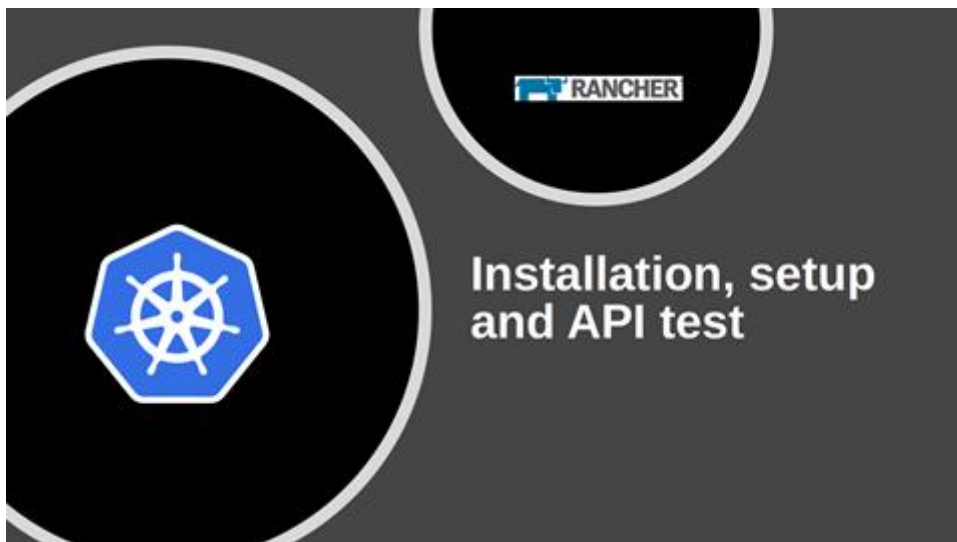
There are other more lightweight Kubernetes alternatives like Lightweight Kubernetes⁶ which has very low system requirements. It has following minimum system requirements which means that it can even run on a Raspberry Pi.

- 512 MB of RAM per server
- 75 MB of RAM per node
- 200 MB of disk space

4.2 Installation of Rancher and Kubernetes

Because Rancher is an external component we recommend using the official Rancher installation documentation that can be found here <https://rancher.com/docs/rancher/v2.x/en/installation/>.

But we have created an step-by-step video that goes through installation and setup that can be used when installing BalticLSC Platform in an early stage. The installation will be improved after the Platform is fully implemented and new instructions will be provided.



Video link: https://youtu.be/e_ss4SA4hY

Content:

1. Deployment of nodes
2. Preparing nodes
3. Installing Rancher
4. Installing Kubernetes using rancher
5. Configuring monitoring

⁶ <https://k3s.io/>

6. Configuration and setup of a restricted user with custom security profile and resource quotas
7. Installation and setup of kubectl. Access via API for admin and restricted user

4.3 BalticLSC settings in Rancher

This section describes how to manually setup Rancher to support BalticLSC. It is also documenting how Rancher and Kubernetes is setup to support the requirements that BalticLSC Platform has. Not all requirements are fulfilled by just Kubernetes and Rancher, therefore there are some new software components that needs to be implemented. These will be covered later in this document. The setup involves creating users, projects, security profiles and their associations, as well as how to setup resource quotas and limits.

4.3.1 Custom security policy

To restrict users from being able to do harm, a custom **pod security policy** is created which is bound to the user's project (shown in 4.3.2). The policy should only deny access to critical capabilities and resources. Thus, the user should not be able to do harm to the cluster and should not be restricting in normal operation. It is not an easy task to define an unrestricting policy that is still protecting the cluster. Following is the recommended policy settings:

- Name: rise-custom
- Basic Policies: No on all
- Capability Policies:
 - Allowed Capabilities: CHOWN , NET_BIND_SERVICE , NET_ADMIN, NET_BROADCAST , NET_RAW , SETGID , SETUID
 - Default Add Capabilities: CHOWN , NET_BIND_SERVICE , NET_BROADCAST , NET_RAW , SETGID , SETUID
- Required Drop Capabilities: None
- Volume Policy: emptyDir, secret, persistentVolumeClaim, downwardAPI, configMap, projected
- Allowed Host Paths Policy: None
- FS Group Policy: RunAsAny
- Host Ports Policy: None
- Run As User Policy: RunAsAny
- SELinux Policy: RunAsAny
- Supplemental Groups Policy: RunAsAny

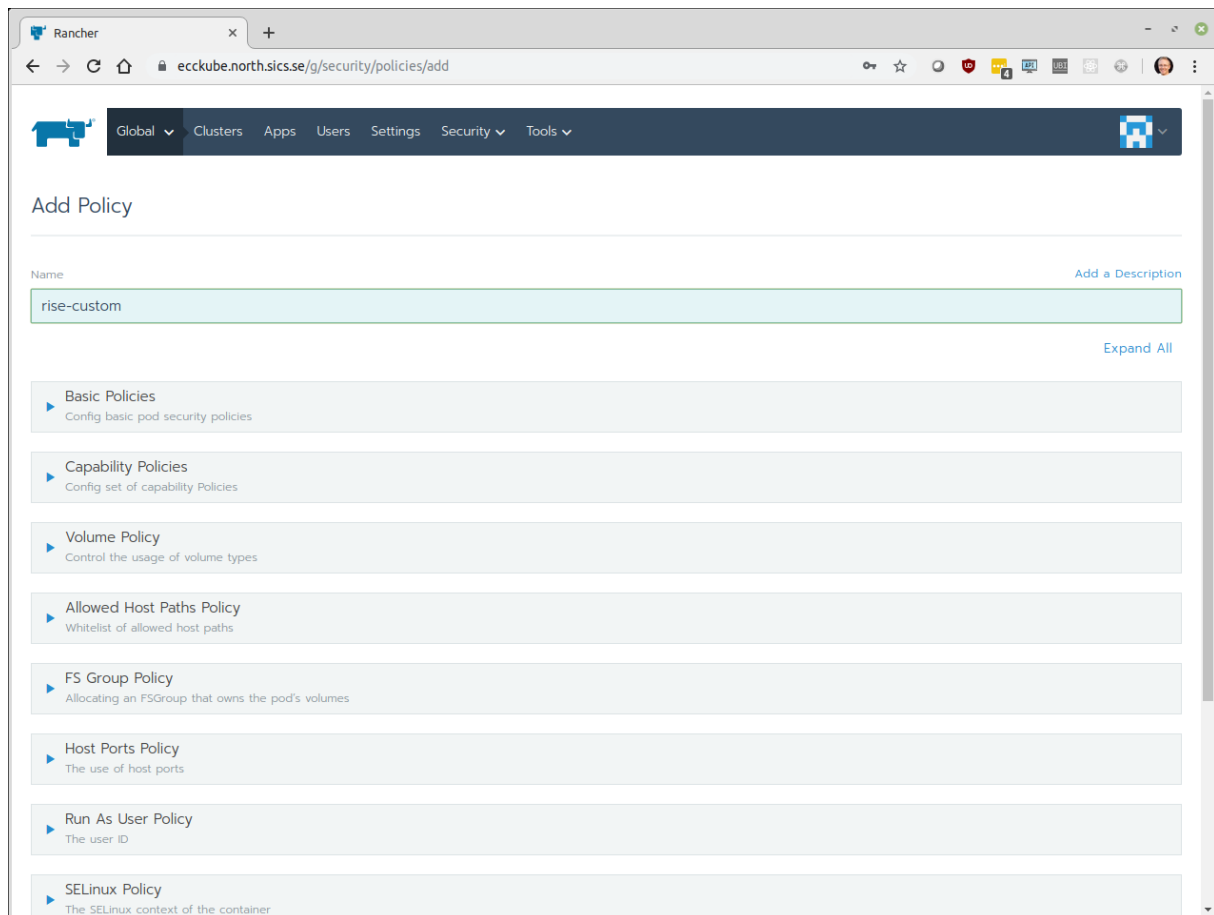


Figure 6: Adding custom security policy

4.3.2 Create user with associated project and resource quota

In this example we add an account for user bob@mail.com in Rancher. First, we create the user and fill in following:

Username: balticlsc

Password: *****

Display Name: BalticLSC Software User

Global Permissions:

- Custom
 - Use Catalog Templates
 - Login Access

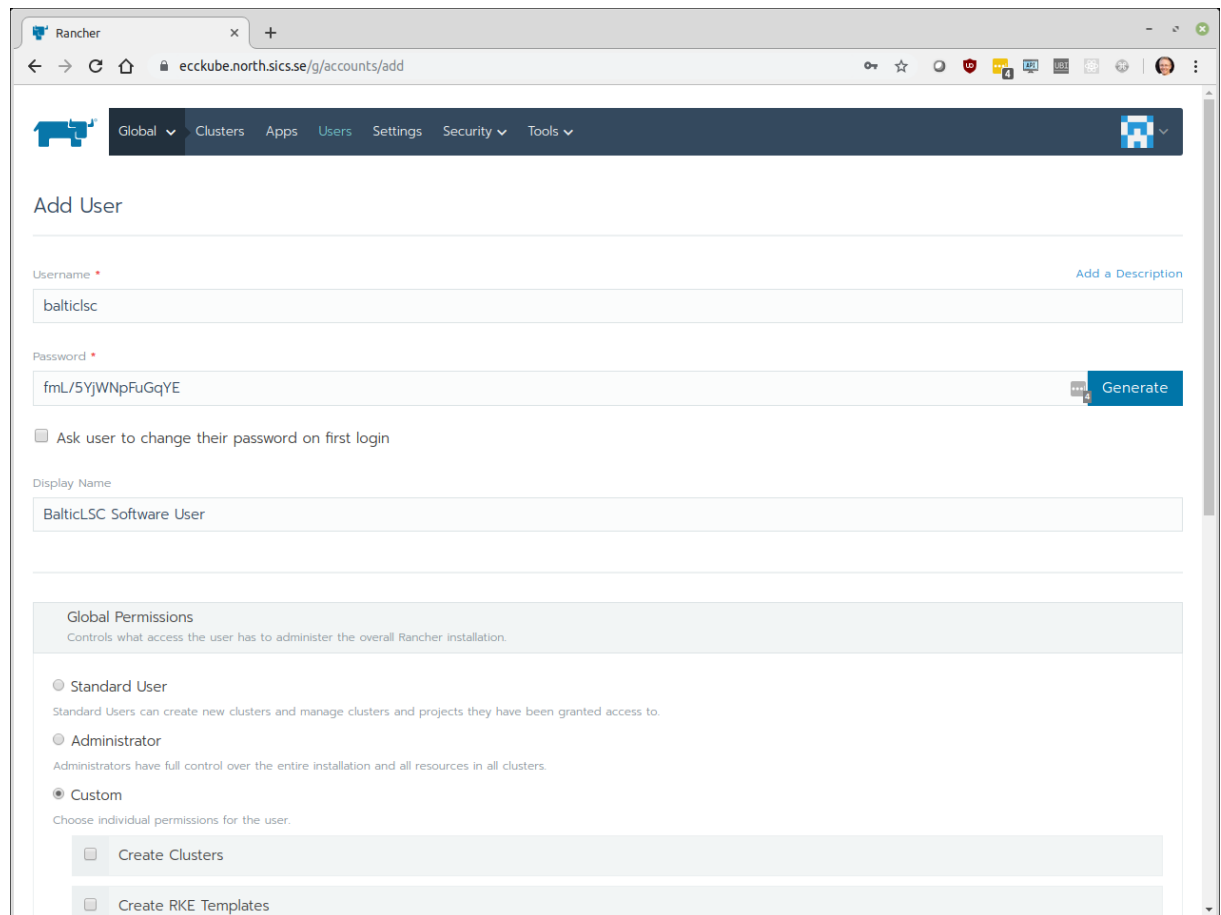


Figure 7: Screenshot adding user in Rancher

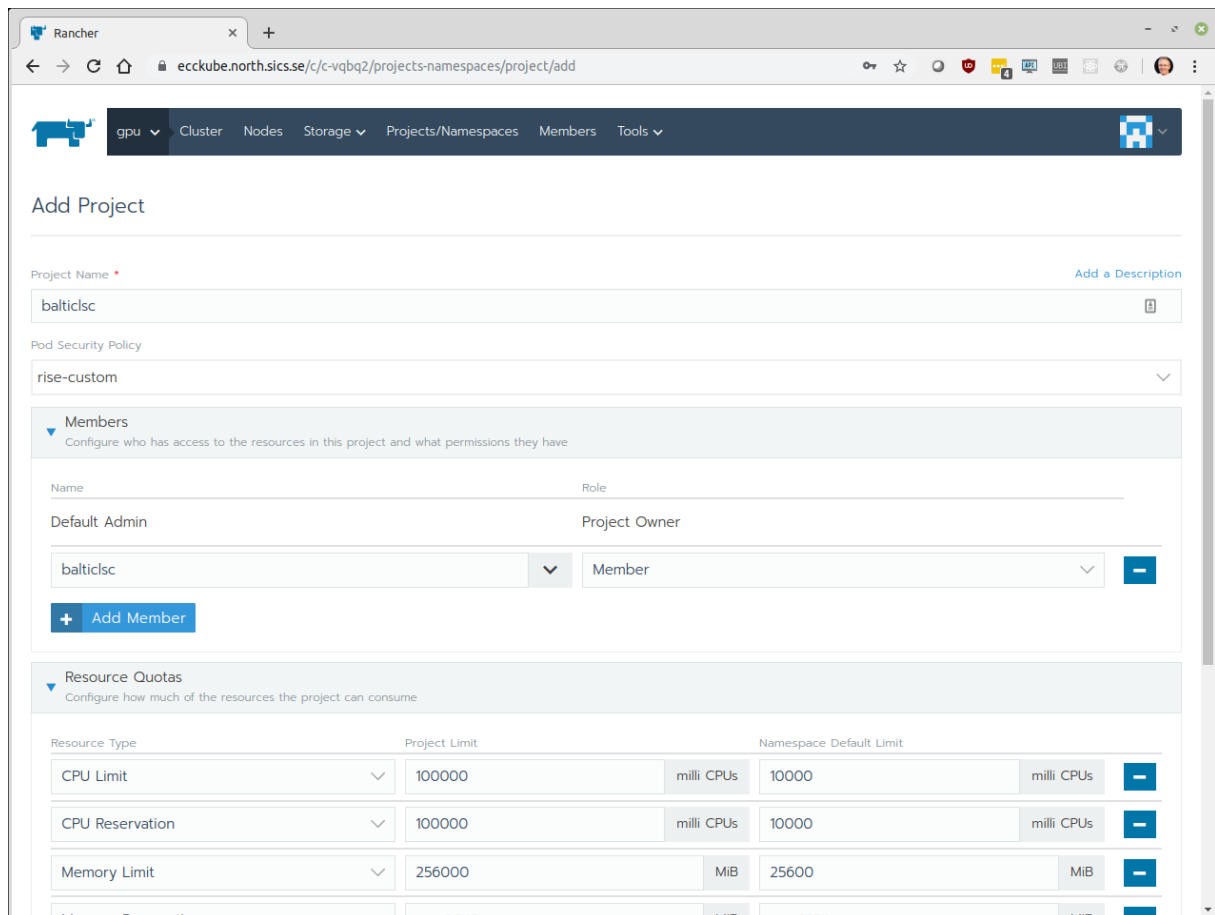
Then we add a new project with the user above as a project member:

- Project Name: balticlsc
- Pod Security Policy: rise-custom
- Add user balticlsc as project member
- Resource Quotas

Resource type	Project	Default namespace
CPU Limit	100000 mCPUs	10000 mCPUs
CPU Reservation	100000 mCPUs	10000 mCPUs
Memory Limit	256000 MiB	25600 MiB
Memory Reservation	256000 MiB	25600 MiB
Persistent Volume Claims	50	5
Storage Reservation	10000 GB	1000 GB

- Container Default Resource Limit

Resource type	
CPU Limit	2000 mCPUs
CPU Reservation	2000 mCPUs
Memory Limit	256 MiB
Memory Reservation	256 MiB



Add Project

Project Name * Add a Description

balticlsc

Pod Security Policy

rise-custom

Members
Configure who has access to the resources in this project and what permissions they have

Name	Role
Default Admin	Project Owner
balticlsc	Member

[+ Add Member](#)

Resource Quotas
Configure how much of the resources the project can consume

Resource Type	Project Limit	Namespace Default Limit
CPU Limit	100000	10000
CPU Reservation	100000	10000
Memory Limit	256000	25600

Figure 8: Adding project for balticlsc user

5. System Design

This section contains more detailed information about sub-components mentioned earlier in the system architecture.

5.1 Hardware resources and pricing

The hardware consists of computation servers, hereafter called nodes, interconnected using network switches.

There are multiple types of hardware resources that can be configured differently on different Nodes. All resources can be priced differently, and the same type of resource can be priced depending on the model and version. A factor that also can make the price vary is the geographical location of the resource because the cost of electricity can vary. Electricity cost variations and network switch resource usage costs are not taken into consideration by the platform. Also note that the platform provides resource usage information that another system can use for billing. Thus pricing in set in another system, however this section gives some pricing suggestions.

5.1.1 Central Processing Unit - CPU

One thing to consider in the cluster regarding CPUs is the type of CPU running on the nodes. This information is needed both in terms of how much computation power is available and for the pricing of the resource. One CPU core is divided into millicores, 1/1000 of a core, in Kubernetes. This means that computation tasks don't need to claim full cores.

Pricing will be set per CPU/h. I.e. the price for one CPU-core for one hour which corresponds to 1000 mCPUs for one hour in Kubernetes. There will not be any differentiation between types of CPUs, which means that a CPU with high performance will have the same price as a CPU with low performance.

5.1.2 Memory

Much of what is being said about the CPU can be said about memory. Kubernetes provides information about the total amount of memory on one node. Detailed information about memory could be interesting for the business model.

Pricing should be set per MiB/h.

5.1.3 Storage

Storage should be located as near as possible to the computation resources so that large amounts of data doesn't need to be moved long distances between computation cycles. The storage can be located on the Nodes where the computation should be done or on a storage cluster nearby. In the BalticLSC Platform storage clusters will be mandatory because the computation is never guaranteed to be run on a specific Node if more than one Node fulfill the requirements for the resources. There can be different classes of storage, priced differently, that have different performance. To prevent data loss the storage must replicate the data.

Pricing should be set per TB/h.

5.1.4 Network

Network traffic can be divided into two categories, inbound and outbound network traffic. But the inbound and outbound traffic can occur on different layers in the network, it can be inside the cluster but also from/to the cluster from/to Internet. There isn't any built-in functionality to distinguish between traffic going outside the cluster and internally. Most of the traffic is most likely only traveling locally because the tasks are computation oriented on data already in the cluster. This, of course, assumes data has already been moved to the cluster.

Pricing would preferably be set per GB of outbound traffic. However, in current monitoring solution there is no differentiation between internal and outbound traffic. Therefore, network resource usage reporting will be omitted in the first version.

5.1.5 Graphical Processing Unit - GPU

Servers usually don't contain a GPU by default except for dedicated GPU server in which case it can contain one or more GPUs. If the software that's being deployed needs GPUs it needs to be run on Nodes with one or more installed. There are some restrictions with regards to GPUs and software libraries, software development kits (SDKs) and the manufacturer of the GPU. The problem with libraries and SDKs is that they are designed for hardware from different vendors. There are tools that try to unify the hardware under common abstraction layers. Today these abstraction layers are not well supported because of the difficulty to port the different platforms.

The most established vendor in the field of using GPUs as computation power is Nvidia, that is why our current recommendation is to only support Nvidia GPUs.

Pricing should be set per GPU type per hour. For example, Nvidia GTX 1080 ti or Nvidia RTX 2080 ti.

5.2 Users, projects and quota

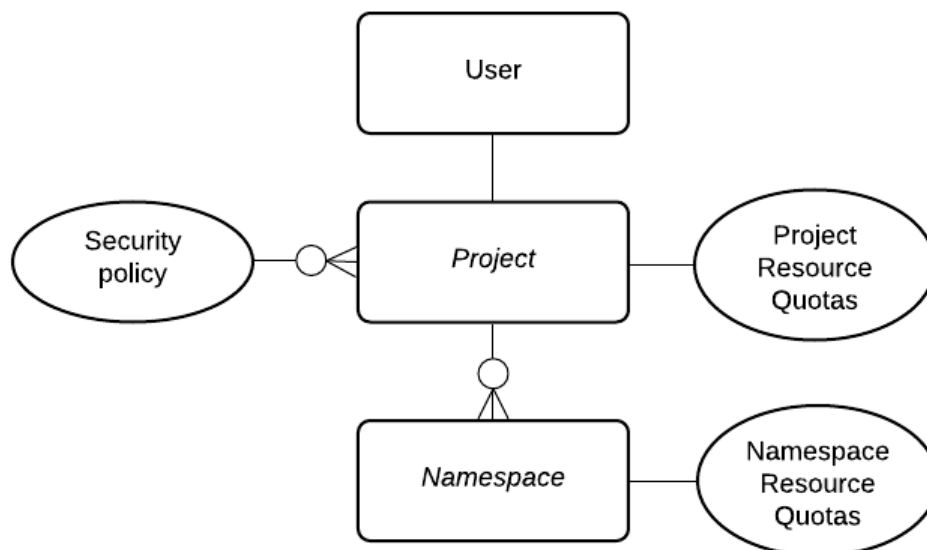


Figure 9: Relations

For easier management of resources and resource quotas, the concept of *projects* is used. A project is a collection of namespaces. A resource quota can then be applied on the project and a user assigned to the project. So, the BalticLSC Software user would be assigned a project with the appropriate resource quota and permissions.

The cluster owner set restrictions on how much resources BalticLSC Platform user can use. This is done on the project assigned to the user. Kubernetes implements resource quotas on namespace level, so the user needs to assign part of the project quota to namespaces upon creation. If no quotas are assigned, defaults (set on a project) should be applied.

Permissions are controlled in Kubernetes with Role-based access control (RBAC). The recommended permissions for the BalticLSC user are:

- Only allow namespaced scope, i.e. only allow control over namespaces created by the user
- The user needs a ClusterRole, with permissions to create namespaces

When creating a new namespace through the Kubernetes API there is no information available of who created the namespace. So, the namespace cannot be assigned to a project with the appropriate permissions owned by the user. To be able to do this additional information must be added when the namespace is created. Here is an example how-to assign namespace *mynamespace* to project with project id *myprojectId*.

```
An example
# Create Namespace for Feature
apiVersion: v1
kind: Namespace
metadata:
  name: mynamespace
annotations:
  field.cattle.io/projectId: myprojectId
```

One problem with this solution is that it is not transparent to the BalticLSC Platform user, it may not be a problem for BalticLSC Software to include this annotation but if the cluster would be shared with other users it could become a problem. To solve the problem with the mandatory project annotation, the Platform Manager could insert the correct project annotation for the user. This would be transparent for the user. The Platform Manager would intercept requests to the API resource for namespace creation and look up what project the user belongs to. What project a user is tied to can be stored in a custom resource. This same resource would be used by the controller acting on namespace creation events to know what permissions and roles should be bound to the namespace. The Platform Manager could also prevent users from creating namespaces in someone else projects. This would not pose a security risk because the namespace would be assigned with permissions only allowing the project owner to use the namespace, but it would create confusion and allow for an DoS-like attack.

5.2.1 One BalticLSC Software user

It is important to understand that the mapping between BalticLSC Software user and BalticLSC Platform user is not one-to-one. The correct mapping is many-to-one. This means that the same BalticLSC Platform user account is used to execute computation tasks of many BalticLSC Software users. If a one-to-one mapping would be used, then BalticLSC Software would need administrator privileges to the cluster. We do not think that many resource providers are willing to grant admin privileges, thus this must be avoided.

5.3 Network isolation

All the BalticLSC Platform user applications should be isolated from other platform user applications on a network level running in the cluster. The user applications should not be able to access other user network services. The user should only be able to access services running in the user project. Kubernetes can do this with network policies. To implement network policies, a network plugin compatible with the containers running must be installed. We have evaluated Canal/Calico which works well.

5.4 Resource monitoring

Rancher has “built-in” cluster resource monitoring that can be enabled on a Kubernetes cluster. When enabled a resource monitoring framework is deployed in the cluster. It consists of a Prometheus

database, Grafana with pre-configured dashboards and monitoring agents. This monitoring framework collect enough data to build billing upon.



Figure 10: Grafana screenshot

5.4.1 Cluster monitoring settings

Enable resource monitoring in Rancher with following recommended settings:

Name	Value
Data retention	1080 hours
Enable Node Exporter	True
Enable Persistent Storage for Prometheus	True
Persistent Storage Size	500Gi
Grafana Persistent Storage	10Gi
Prometheus CPU Limit	2000 mCPUs
Prometheus Memory Limit	2000 MiB
Node Exporter CPU Limit	200 mCPUs
Node Exporter Memory Limit	200 MiB

GPU monitoring can be enabled using following Prometheus exporter. However, it is not necessary for the function of the system.

<https://github.com/NVIDIA/gpu-monitoring-tools/tree/master/exporters/prometheus-dcgm>

5.5 Billing

Billing is based on monitoring data in Prometheus database. I.e. resources reserved by a user in a specific time interval. Need to query both requested and actual usage and base billing on the higher value of them both (see Figure 11: Billed utilization).

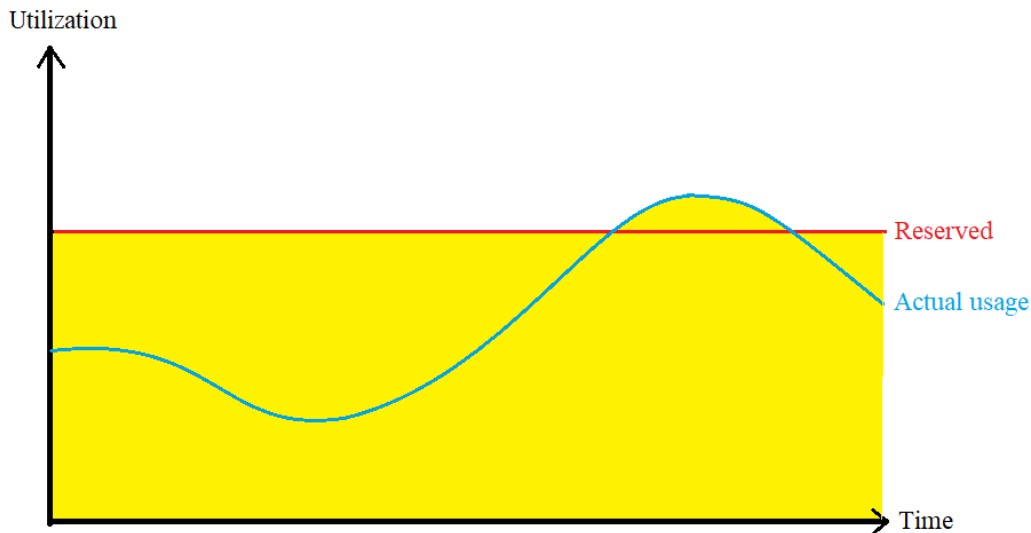


Figure 11: Billed utilization

5.5.1 Utilization reports for billing

The cluster monitoring data in Prometheus is very thorough and is only stored for a set number of hours. For this reason, it is needed to continuously store user utilization for every user. To do this there is a framework called *Operator Metering* that can be used.

Operator Metering is part of the *Operator Framework* from CoreOS. Operator Framework is an open-source toolkit designed to manage Kubernetes Operators in an effective, automated and scalable way. Operator Metering framework enables custom usage reporting derived from monitoring data that can be used for our purposes. It is suggested to generate reports in following intervals:

- Hourly
- Daily
- Monthly

The usage breakdown reports should be by namespace, which can then be used to summarize project/user utilization for a given period. In addition, it would be nice to summarize the namespace reports into user reports. The reports should be created on following metrics:

- CPU request – Amount of reserved CPU
- CPU usage – Real CPU usage (recorded)
- Memory request – Amount of reserved memory
- Memory usage – Real memory usage (recorded)
- Storage request – Amount of reserved storage
- Storage usage – Real storage usage
- GPU request – Number of reserved GPUs of different types

5.6 API usage specifics

5.6.1 Project id reference when creating namespace

Project annotation must be included when creating namespace via Kubernetes API (kubectl) annotations:

```
field.cattle.io/projectId: c-sqc8w:project-2bmmx
```

5.6.2 Resource quotas on namespaces

When a user creates a new namespace, quotas should be set. The quotas are not allowed to exceed the project quotas (that the user is not allowed to change). If namespace default quota is wanted, then resource quota information does not need to be included.

5.6.3 Container resource requests and limits

When deploying containers, the amount of needed resources needs to be communicated. This information is used by Kubernetes scheduler to allocate compute resources and do scheduling. Each container may specify a request and limit value for CPU and memory. It is recommended to set request value equal to limit value. See following example creating pod with two containers with requests and limits set:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
    - name: demo1
      image: demo/demo1
      resources:
        requests:
          memory: "32Mi"
          cpu: "200m"
        limits:
          memory: "32Mi"
          cpu: "200m"
    - name: demo2
      image: demo/demo2
      resources:
        requests:
          memory: "128Mi"
          cpu: "400m"
        limits:
          memory: "128Mi"
          cpu: "400m"
```

If a container's default resource limit is not set on a namespace, then resource limits must be included in specification when deploying containers. If container default resource limit is set on namespace and this limit is wanted, then the resource request and limit can be omitted.

5.7 Priority and pre-emption

Priority classes can be used in Kubernetes to give pods priority over other pods during scheduling. If a pod cannot be scheduled, and the priority is high enough, the scheduler can pre-empt (evict) lower priority pods to make scheduling of the pending pod possible. Pod pre-emption should only be activated

on critical non-workload applications in the cluster. Thus, pre-emption is not available for the user of BalticLSC Platform. However, access to other priority classes may be available to users. To enable pre-emption feature, Kubernetes version must be 1.15+ and NonPreemptingPriority feature needs to be enabled.

Create PriorityClasses that are assigned to users:

- Critical (preemption enabled) – critical-priority
- High – high-priority
- Normal – normal-priority (default)
- Low – low-priority (best effort)

Pod preemption can be set per priority class. {preemptionPolicy: Never} should be set on all PriorityClasses except “Critical” which get default {preemptionPolicy: PreemptLowerPriority}

How billing and quota per PriorityClass will be managed in this case is omitted in this document.

5.7.1 Using PriorityClasses

User needs to specify priorityClassName for pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

<https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/>

5.8 New software components

This section holds information about new components that need to be implemented. There will be one component called *controller* for every function. The controller’s purpose is to monitor a specific resource and making sure it holds a specific state. There should be no dependency or interaction to other controllers in the system. This way it is easy to distribute the implementation in a development team.

5.8.1 Namespace controller

Namespaces are global; thus, conflicts can happen. If someone tries to create a namespace that already exist, he will get an error back. A suggestion would be to require users to add a prefix to all namespaces that are created.

When creating a namespace via kubernetes API (kubectl), it is mandatory to add annotation (field.cattle.io/projectId: [clusterId]:[projectId]) to the spec which binds this new namespace to the users project. The problem is that it is possible to create a namespace without this annotation, but the user is then not allowed to access it. One solution that is to delete all namespaces that are created without any annotation. Another and better solution would be to add this annotation upon namespace creation.

To solve the above, a software component called “Namespace Controller” needs to be implemented. Its task will be:

1. To watch for namespace events and make sure that namespaces gets associated with the user’s project

2. Remove annotation references to wrong projects, or just make sure an error is returned to user
This controller must be installed in the “Rancher cluster” to be able to access all information needed.

Note: When creating namespace via Rancher API or Rancher UI then it is put inside the correct project.

5.8.2 Ingress rule conflict controller

Problem

In Kubernetes it is possible to create two ingresses with the same host rule (ex: service.foo.com). What happens with nginx ingress controller is that the second (conflicting) rule is ignored. However, the second ingress object is created, and it is not possible to see that provisioning failed. And when you are not in control of the whole Kubernetes cluster (shared cluster), it is not possible to see the conflicting rule.

Solution

Implement new controller which adds missing ingress rule conflict handling to Kubernetes API which return error upon conflict.

5.8.3 Extension API-server serving user resources

To give the BalticLSC user access to resource history and billing, an extension API-server with controllers needs to be implemented. The API extensions for resource usage and billing is to be decided later.

5.8.4 GPU quota

Problem

GPU quota can be set on namespace using kubectl (e.g. limits.nvidia/gpu: 4), but not through rancher.

Solution

Investigate if project quota for GPU is supported even though it can't be set via Rancher GUI. If it is not supported a new controller need to be implemented handling this.

6. Use cases

This section is a work in progress. Additional use-cases will be added during the progression of the project.

Action	End-user creates new BalticLSC Software account
BalticLSC Software	User account is created.
BalticLSC Platform	No interaction with BalticLSC Platform

Action	End-user delete BalticLSC Software account
BalticLSC Software	User account is deleted
BalticLSC Platform	No interaction with BalticLSC Platform

Action	Resource provider create BalticLSC Software account
BalticLSC Software	User account is created
BalticLSC Platform	No interaction with BalticLSC Platform

Action	Resource provider register his BalticLSC Platform resources
BalticLSC Software	Registration consists of uploading API configuration file for balticlsc in BalticLSC Platform to BalticLSC Software. This is the Kubeconfig file.
BalticLSC Platform	Resource provider extract kubeconfig file for balticlsc user to be uploaded.

Action	BalticLSC Software User A and BalticLSC Software User B start computation task simultaneously on same BalticLSC Platform
BalticLSC Software	<p>Use BalticLSC Platform API to create two new namespaces. One for User A and one for User B.</p> <p>User A computation task is executed in namespace A.</p> <p>User B computation task is executed in namespace B.</p> <p>When computation task for user A is finished, results are collected and namespace A is deleted.</p> <p>When computation task for user B is finished, results are collected namespace B is deleted.</p> <p>BalticLSC Software will later collect utilization reports for namespace A and namespace B. These are used for billing the separate users.</p>
BalticLSC Platform	<p>Executes directives received from BalticLSC Software via API (scheduling, execution, etc).</p> <p>Record resources used during execution per namespace.</p>