The future high-level trigger orchestration in ATLAS

EP-ADT-DQ – **MATTEO FERRARI**

ON BEHALF OF ATLAS TDAQ COLLABORATION

CEPC INTERNATIONAL WORKSHOP, OCTOBER 2023

Outline

Introduction: ATLAS Event Filter Farm

- From collision to tapes storage: a video introduction
- LHC / Detector performance
- Level 1 filtering
- High Level Trigger filtering: Event Filter Farm
- Enhancements for High Luminosity LHC (HL-LHC)

EF Farm Orchestrator

- Current Solution and Phase 2 enhancement
- What is Kubernetes? Why?
- Current prototype implementation
- Benchmark and tests

Conclusions

Note about acronyms:

- K8s -> Kubernetes

From collision to data storage

Currently, during operations, LHC delivers collision data at 40 MHz

We can't save all the data. That's why we need filtering

With filtering we discard events we are not interested to analyse (because, for example, they are very common and well-studied)

Level 1 Trigger -> Hardware filtering

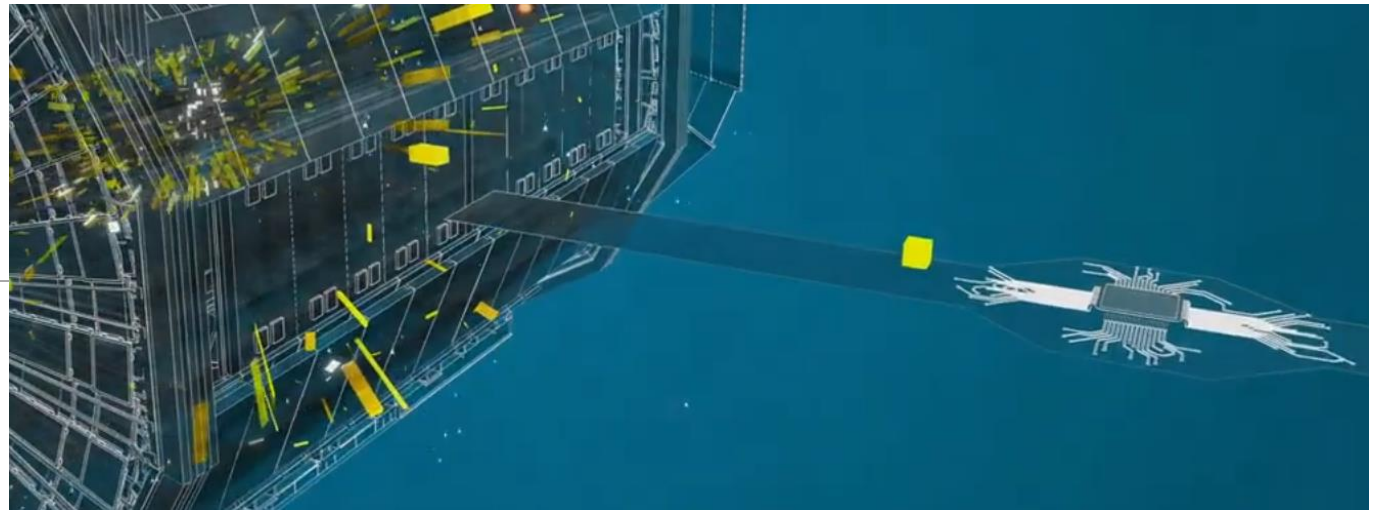
High Level Trigger (HLT) -> Software filtering (2k server farm)

The entire system is real-time.

- Queues and buffers are very limited
- Buffers can't be the solution to performance related issues.

[Link: Video@CERN]

Level 1 Trigger



Very fast filtering

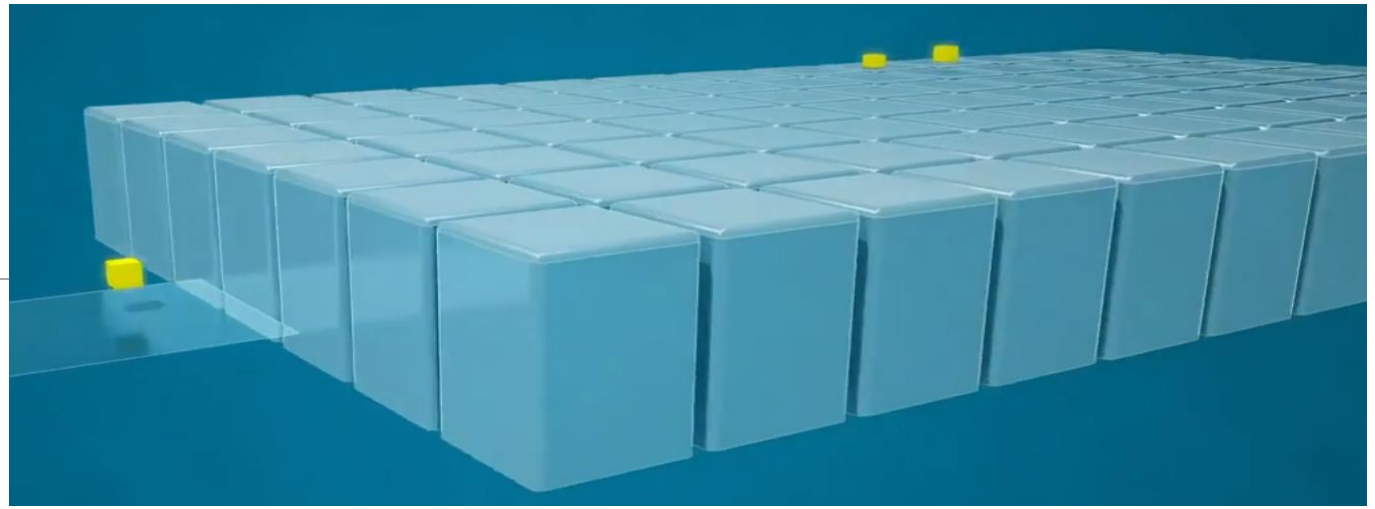
It's hardware based

Current output data: 100 kHz

It receive data at 40 MHz from the Detector

It has less than 5 µs in order to take a decision

HLT filtering



Software Based

Farm of 2000+ servers

More than 60k real CPU cores

Current physics output data: 1-3 kHz

Event Size: ~1,5 MB

~8 GB/s are sent to permanent storage.

HL-LHC: enhancements & challenges

	Current	HL-LHC
Luminosity ($10^{34}cm^{-2}s^{-1}$)	2	7.5
Pile-up	60	200
L1 Output	100 kHz	1 MHz
Physics HLT Output	3 kHz (~4.5 GB/s)	10 kHz (~50 GB/s)
Output Event size	~1.8 MB	~5 MB
48h Permanent Offline Storage	~1.5 PB	10 PB

HL-HLT Challenges

HLT needs to scale x10 more than how is performing during Run 3 and mitigate the rise in algorithm execution times (because of increased pile-up and luminosity)

This goal will be reached by scaling different aspects of HLT:

Horizontal scalability:

- Increase of hardware units (new servers)
- Improve network topology

Vertical scalability:

- Better selection algorithms
- Evolution of the HLT Framework (multi-threading support)
- Support for external accelerators (co-processors or GPU)
- Better resource handling (orchestrator)

EF Orchestrator

EF computing farm is the last step of the event selection component of the TDAQ System.

Current solution is statically configured with no containerised application.

It's working fine, but we can do a lot better:

- Reducing maintenance cost
- Improving resource allocation
- Improve effortless scalability
- Improve monitoring
- Reliability

Kubernetes appears to be the perfect solution for this scenario

Kubernetes

Announced by Google in 2014, it can be described as "*a system for automating deployment, scaling and management of containerised applications*"

Just after 10 years is now one of the preferred deployment solution in the industry

It has a huge open-source community behind with big partners like RedHat, CoreOS and Intel

Some key features:

- Highly customizable at any level (scheduler, API, software)
- Scheduling based on required resources and other constraints
- Built-in support for service discovery and load-balancing
- Easy management of storage back-ends (transparent mounting of both local and network volumes)
- Built-in functionalities for manual or automatic scaling of applications

Extending the Kubernetes API

Kubernetes is highly configurable and extensible at any level (Configuration, API, Plugins, etc..)

Extending API via CRD (Custom Resource Definition)

- Kubernetes works with objects that are accessible via REST API: you can create, modify, delete objects
- With CRD, you can define a new object type
- Once you define a new CRD, you need to teach Kubernetes how to handle those new object via the Operator Pattern

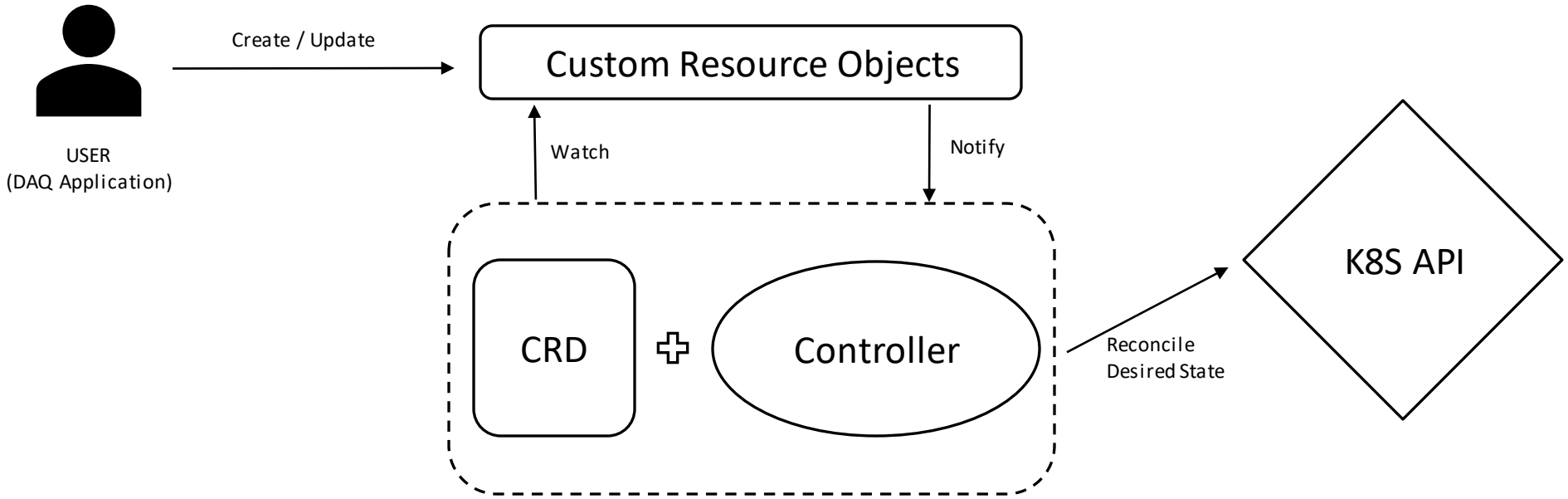
It's like introducing a new recipe (CRD) into a restaurant (k8s) by teaching to a chef (operator) how to cook it. Then, clients (DAQ application) can request this new recipe and the restaurant (k8s) can handle the request, the creation, and delivery.

CRD and Operator Pattern

Operators are software extensions to Kubernetes in order to manage CRD objects.

- It follows Kubernetes principles, in particular the **control loop**

Operators are clients of the Kubernetes API that act as controllers for a resource being deployed or configured.



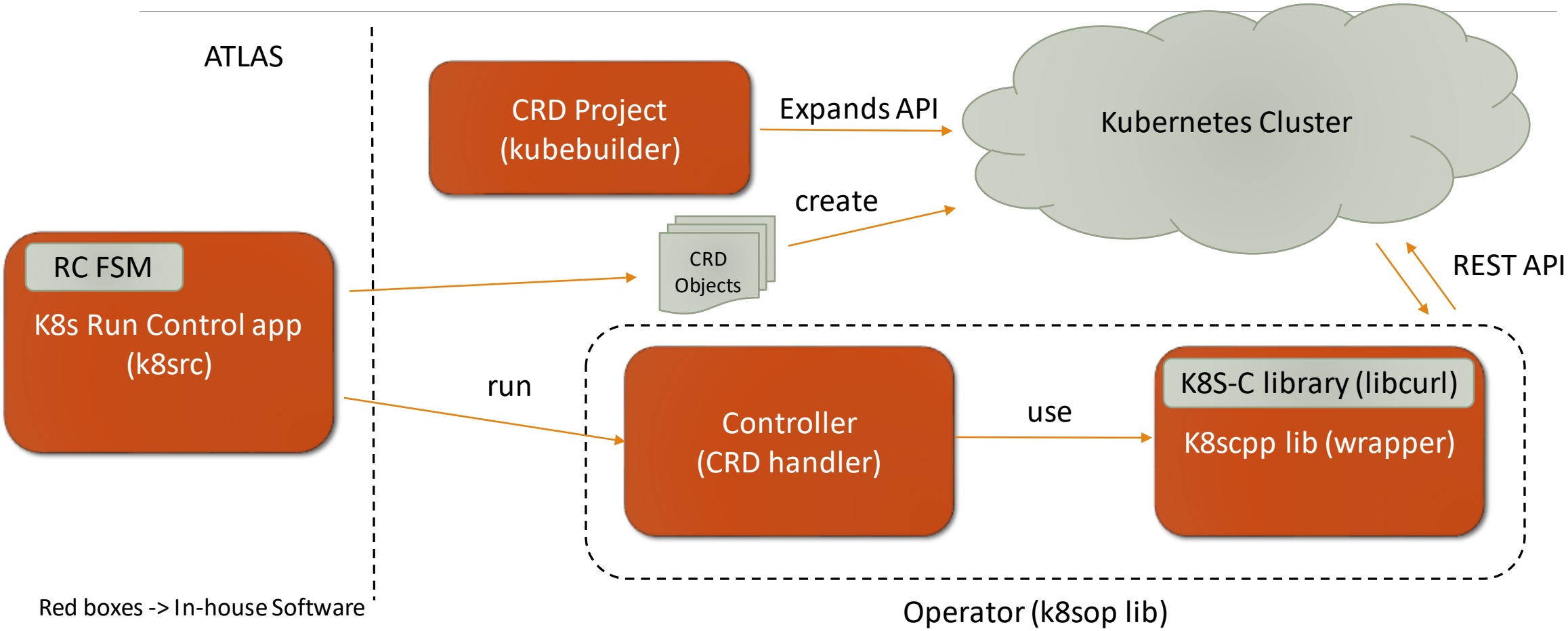
Kubernetes and DAQ sessions

How?

When a data taking sessions starts (it means there will be collisions and data coming from the detector), we use a software bridge in order to deploy all the filtering algorithms in the EF farm orchestrated by k8s.

We need a DAQ Bridge to the Kubernetes Cluster

Architecture of the prototype



CRD Project with Kubebuilder

Kubebuilder

- It's an official tool/sdk (in GO) - Documentation
- It expands API in a declarative way with CRD (Custom Resource Definition) with strict grammar
- It helps to deploy easily this expansion on a k8s cluster

```go
type RunControlApp struct {

    // Specifies the type of deployment.
    // Valid values are:
    // - "Deployment" (default);
    // - "Daemonset";
    // +optional
    Type DeploymentPolicy `json:"type,omitempty"`

    Name  string `json:"name"`
    Image string `json:"image"`

    // The number of replicas. Default is 1
    // This is a pointer to distinguish between explicit zero and not specified.
    // +optional
    Replicas *int32 `json:"replicas,omitempty"`
}

// RunControlSpec defines the desired state of RunControl
type RunControlSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    Namespace string          `json:"namespace,omitempty"`
    Apps      []RunControlApp `json:"apps"`
}
```

```json
"spec": {
    "namespace": "default",
    "apps": [
    {
        "type": "Deployment",
        "name": "test-1",
        "image": "k8s.gcr.io/pause:latest",
        "replicas": 1
    },
    {
        "type": "Daemonset",
        "name": "test-2",
        "image": "k8s.gcr.io/pause:latest",
        "replicas": 1
    }
    ]
}
```

Kubernetes-c library

Official Kubernetes client library in order to interact with HTTP REST API

K8s is based on OpenApi Specification

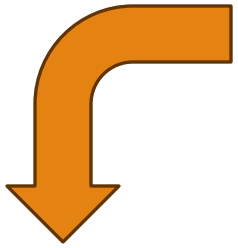
- C client generated with openapi-generator
- We found some critical bug on the C generator. It's going to be fixed

It's using **libcurl** for HTTP communications

k8scpp

C++ Wrapper of the kubernetes-c library

- Make it easy to use
- In-house made
- RAII approach for resource handling
- Handle exceptions
- Convert results in (boost) JSON



```cpp
try
{
  k8scpp::DeploymentsAPI dApi;
  boost::json::value resList = dApi.list("default");
}
catch(k8scpp::K8SException &e) {
    printf(e.what());
}
```

```cpp
char *basePath = NULL;
sslConfig_t *sslConfig = NULL;
list_t *apiKeys = NULL;
/* NULL means loading configuration from $HOME/.kube/config */
int rc = load_kube_config(&basePath, &sslConfig, &apiKeys, NULL);
if (rc != 0) {
    printf("Cannot load kubernetes configuration.\n");
    return -1;
}
apiClient_t *apiClient = apiClient_create_with_base_path(basePath, sslConfig, apiKeys);
if (!apiClient) {
    printf("Cannot create a kubernetes client.\n");
    return -1;
}
```

READ K8S Configuration

```cpp
v1_pod_list_t *pod_list = NULL;
pod_list = CoreV1API_listNamespacedPod(apiClient, "default",    /*namespace */
                                       NULL,    /* pretty */
                                       0,   /* allowWatchBookmarks */
                                       NULL,    /* continue */
                                       NULL,    /* fieldSelector */
                                       NULL,    /* labelSelector */
                                       0,   /* limit */
                                       NULL,    /* resourceVersion */
                                       NULL,    /* resourceVersionMatch */
                                       0,   /* sendInitialEvents */
                                       0,   /* timeoutSeconds */
                                       0    /* watch */
    );
printf("The return code of HTTP request=%ld\n", apiClient->response_code);
```

CALL API

```cpp
if (pod_list) {
    printf("Get pod list:\n");
    listEntry_t *listEntry = NULL;
    v1_pod_t *pod = NULL;
    list_ForEach(listEntry, pod_list->items) {
        pod = listEntry->data;
        printf("\tThe pod name: %s\n", pod->metadata->name);
    }
    v1_pod_list_free(pod_list);
    pod_list = NULL;
} else {
    printf("Cannot get any pod.\n");
}
```

RESULT Handling

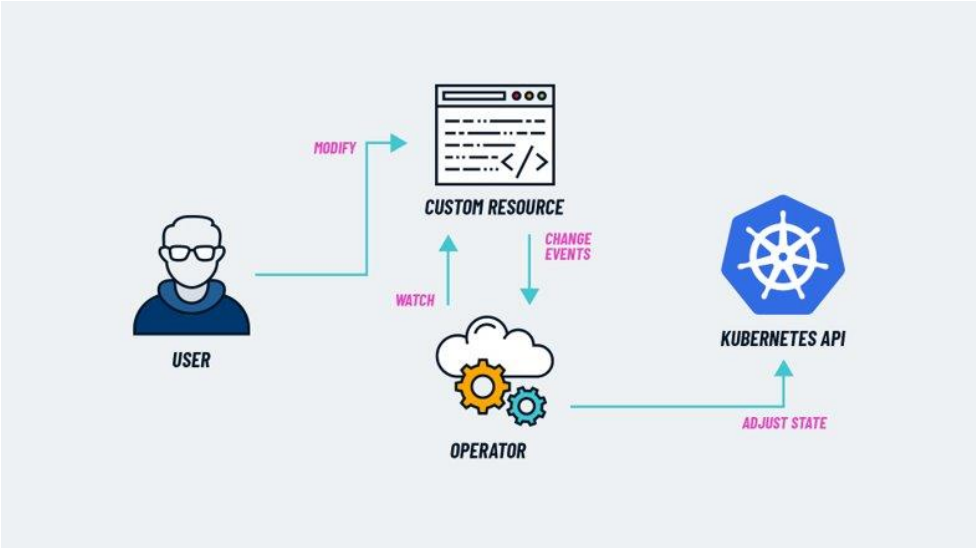
```cpp
apiClient_free(apiClient);
apiClient = NULL;
free_client_config(basePath, sslConfig, apiKeys);
basePath = NULL;
sslConfig = NULL;
apiKeys = NULL;
apiClient_unsetupGlobalEnv();
```

CLEANING

K8sop library

Implements Kubernetes operator pattern.

- In-house made SDK Operator
- Control Loop mechanism. It's a k8s principle for controlling resources over the cluster
- Take actions on CRDs CREATE, MODIFY, DELETE
- Monitor CRDs Status (check that cluster reach the status specified by the CRD)
- It uses k8scpp to interact with k8s cluster

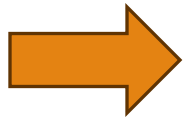


Bridge prototype: K8src executable

Implements the Bridge from DAQ to k8s

- Follow the DAQ Finite State Machine
- It deploys test containers
- Create new CRD Object (list of apps to deploy)
- Check (via k8sop) that k8s cluster is ready

```
"spec": {
    "namespace": "default",
    "apps": [
    {
        "type": "Deployment",
        "name": "test-1",
        "image": "k8s.gcr.io/pause:latest",
        "replicas": 1
    },
    {
        "type": "Daemonset",
        "name": "test-2",
        "image": "k8s.gcr.io/pause:latest",
        "replicas": 1
    }
    ]
}
```



Benchmarks - Test Conditions

Is this solution fast enough? Is it capable to deploy tens of thousands of containers during data taking session initialization?

When a data taking session starts, we need to be quick in the deployment process

If we take too much time, we may lose meaningful data

We did intensive tests in order to find the best configuration of the cluster and minimize start and stop times

Test were executed by deploying a pod (k8s deployment unit) with a single container

Variable number of worker nodes

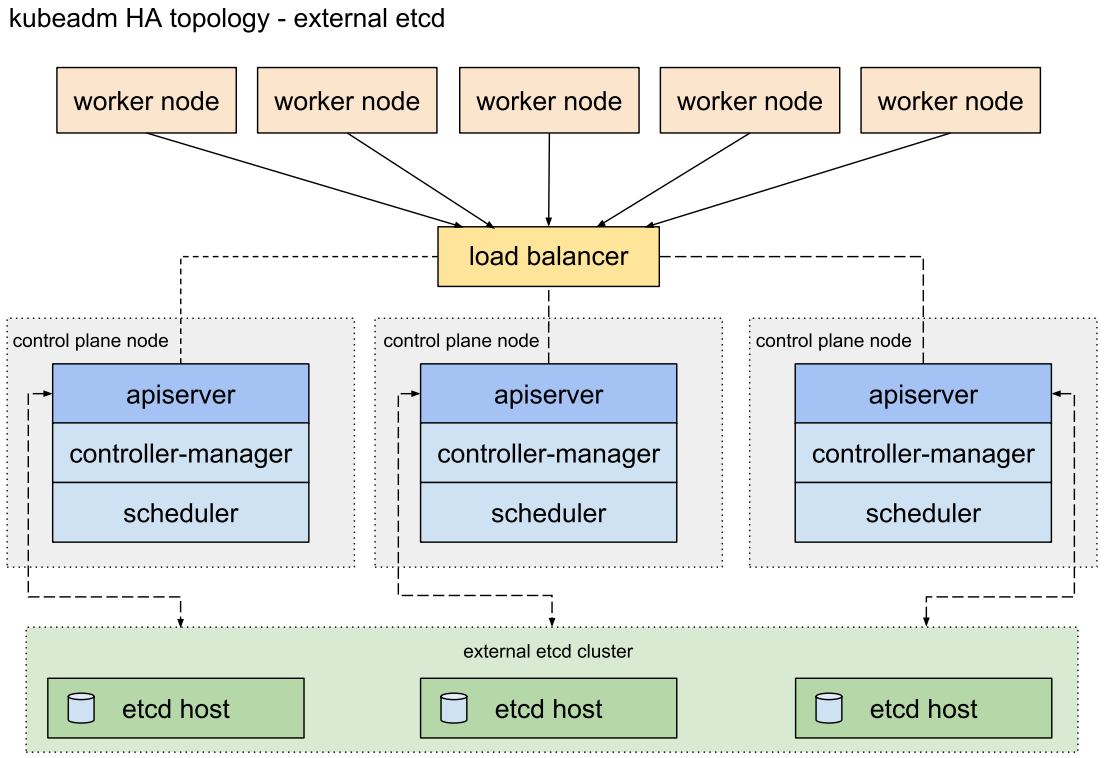
- Different cluster sizes, from 1000 to 2500 worker nodes
- Test with 1, 3, 5 and 10 pods per worker.

Using the ClusterLoader2 toolkit

- Pre-loaded *sleep* container image
- Standard Kubernetes validation tool

Focus on configuration and optimization of the Kubernetes scheduler

Kubernetes Farm



Picture from https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/

4 master nodes

- 2 x Intel Xeon E5-2620 v3 (6/12 cores)
- 32 GB RAM

1 ETCD host

- AMD EPYC 7302P (16/32 cores)
- 64 GB RAM
- 2 SSDs in RAID 0

+2500 worker nodes

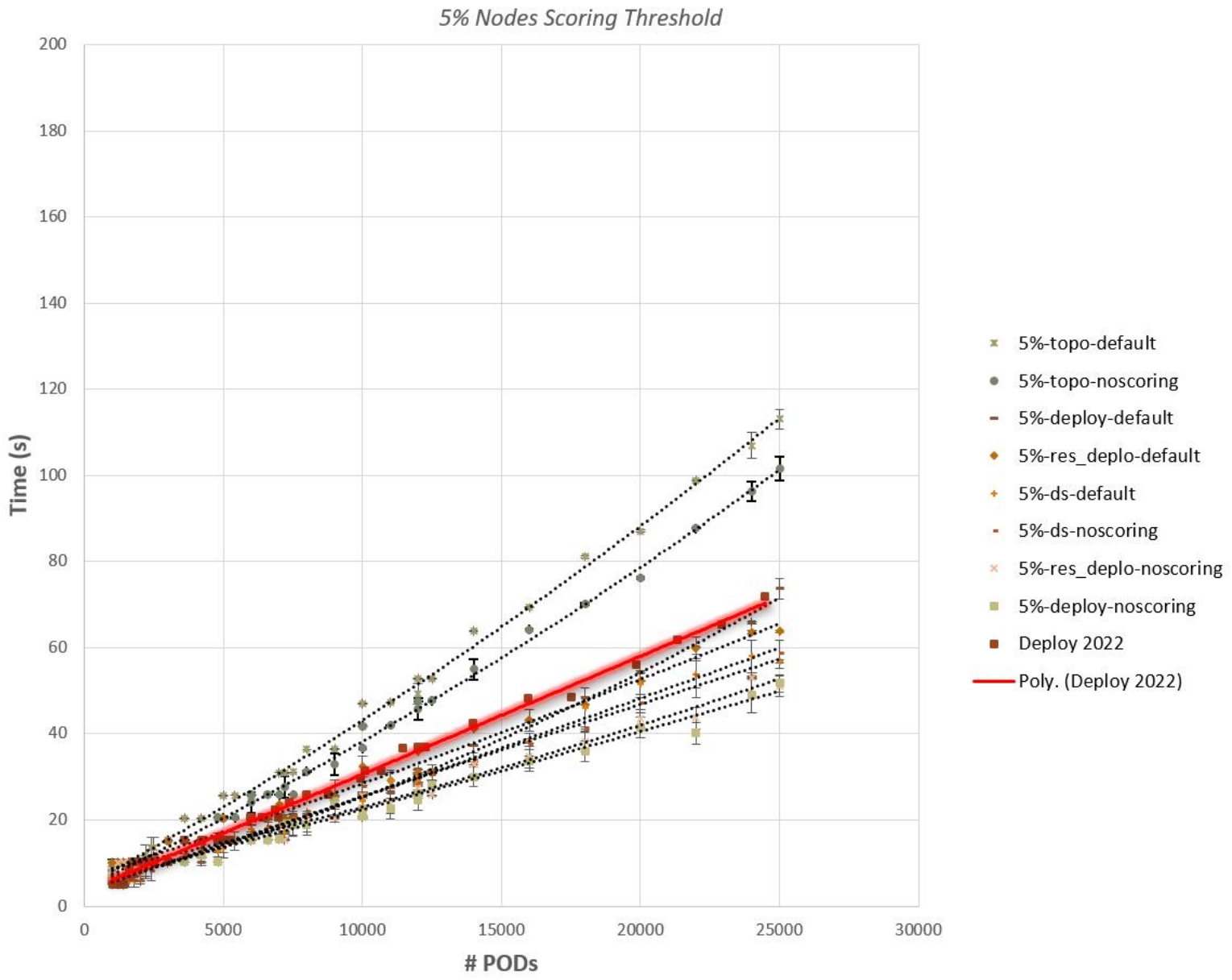
Test Overview

Scheduler Policies	Node Scoring	Deployment Strategies			
		Standard	Standard + Resources Requests	Topological Spread	Daemonsets
Default	Default	✓	✓	✓	✓
	5%	✓			✓
No-Scoring	Default	✓			✓
	5%	✓	✓	✓	✓

Over 14 millions pod start/stop in 5 days of testing

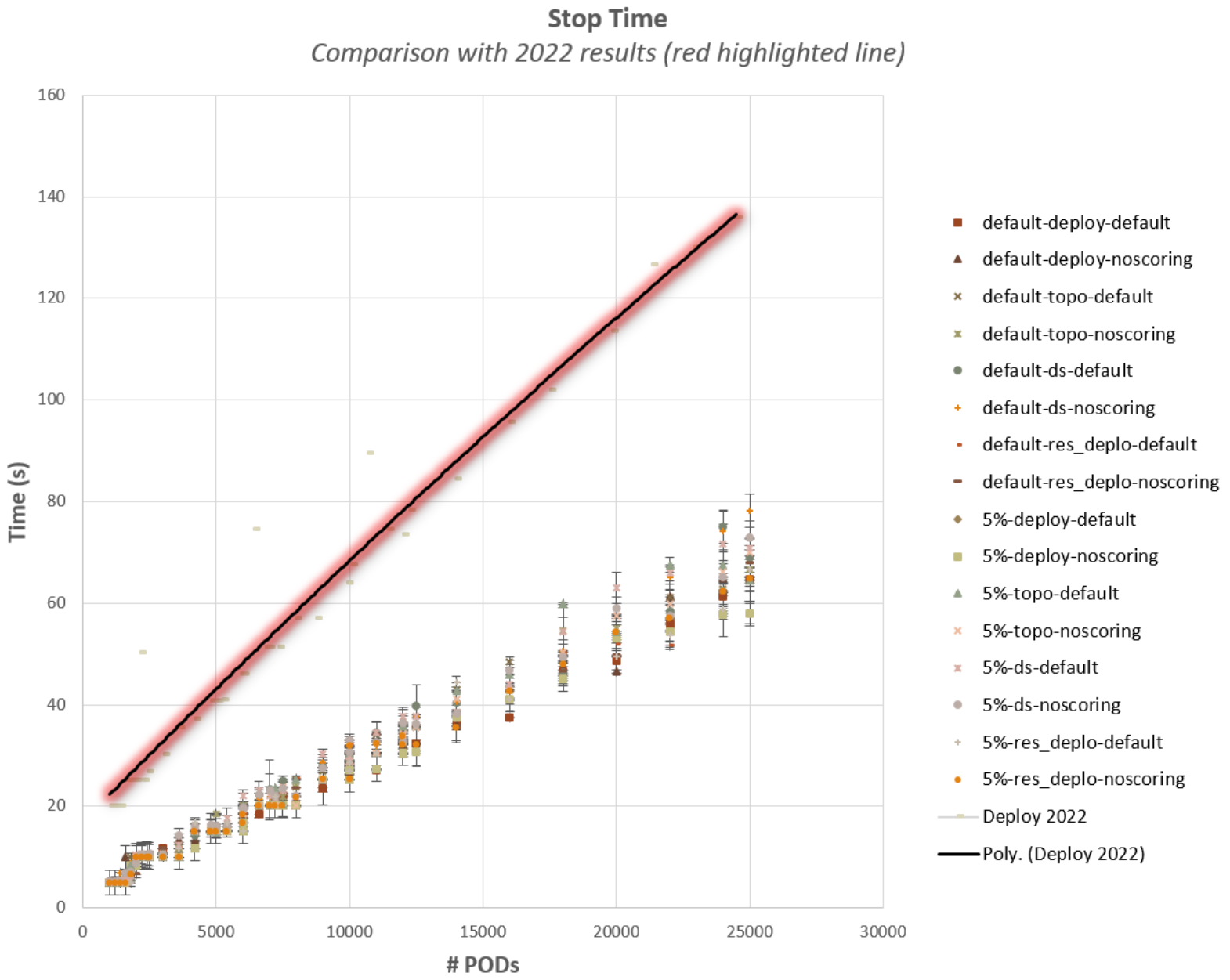
Start Time Results

- In redline test results with an older version of Kubernetes
- We are able to start 25k pods in less than 60 seconds
- We expect to have better results with the latest version of k8s (more optimized scheduling strategies)



Stop Time Results

- In redline test results with an older version of Kubernetes
- There was an huge improvement with a newer version of k8s
- We are able to stop 25k pods in ~60 seconds
- We expect to have even better results with the latest version of k8s



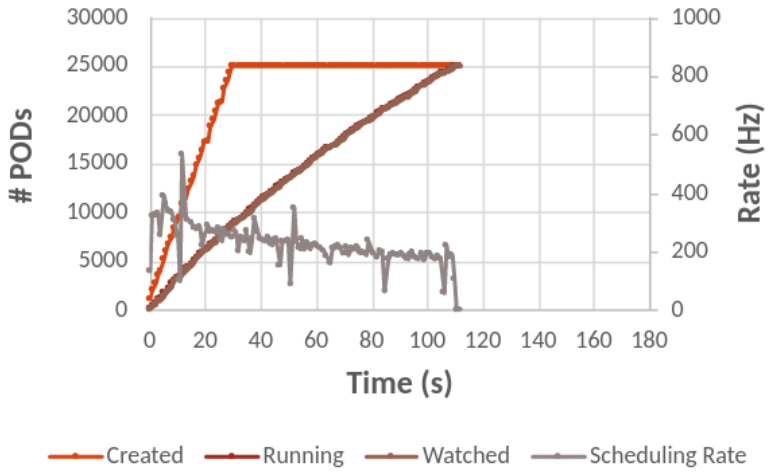
Start Time: Deep Analysis

- In the most optimized configuration, the running rate follows the scheduling rate
- With better hardware this will improve even more

2500 Worker Nodes / 10 Pods per Node

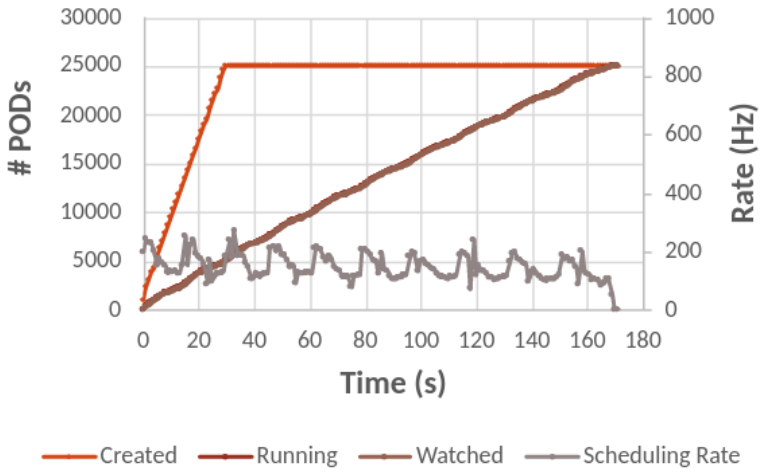
Plain Deployment

Default Scheduler / Default Node Scoring



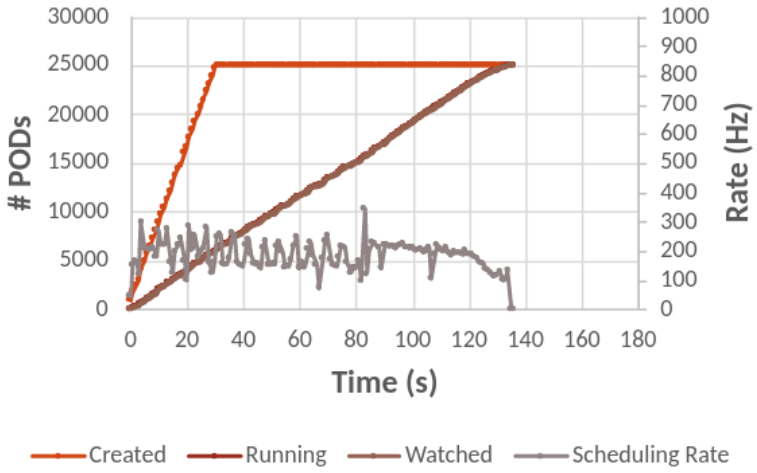
Topological Spread

Default Scheduler / Default Node Scoring



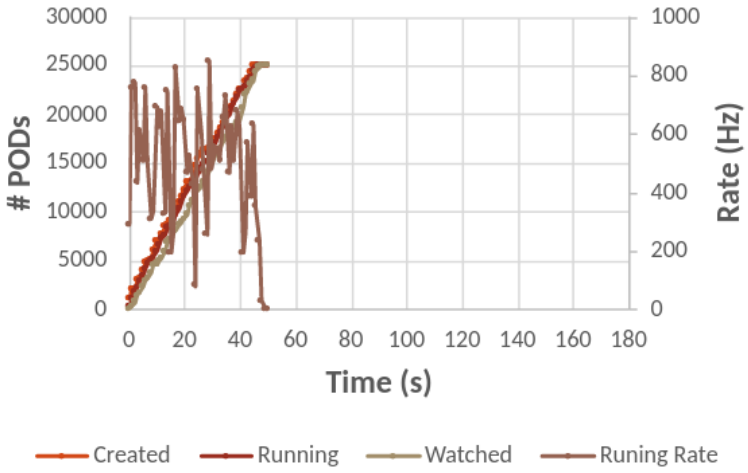
Deploy with Resource

Default Scheduler / Default Node Scoring



Deploy with Resource

NoScoring Scheduler / 5% Node Scoring



Conclusions & Outlook

The full current HLT farm could be orchestrated by Kubernetes

A C++ approach is feasible with basic C++ SDK/Tools implemented.

Start and Stop time benchmark are reasonable for the HLT use-case

Still a Prototype, further investigations and tests needs to be done

Upcoming steps

- Redo benchmark with latest k8s version and better master-nodes hardware
- Continue to implement features to DAQ Bridge software
- Move HLT Filtering Applications to containers and test them on the cluster

Thanks for listening! Any Questions?