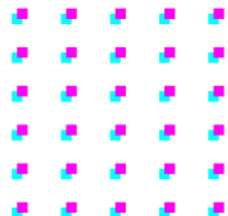




Leading the Big Data
Revolution

Cookbook

Evolve Monitoring,
Scheduling and
Resources allocation



- This document presents the Evolve platform Notebook user guide and documentation for the Frontend UI and the Zeppelin interpreter reference.

Authors	Christian Pinto, Srikumar Venugopal, Christos Kozanitis, George Zervas, Dimos Masouros, Ioannis Baroumas, Jean-Thomas Acquaviva
Description	User Guide for the EVOLVE monitoring and scheduling technologies
Date	June 2021
Revisions	0.1 (June 2021) – Merged all contributions from partners

Contents

■

1. Monitoring of system and application metrics.....	5
1.1 Monitoring of custom applications metrics	5
1.2 IME monitoring through Prometheus	8
2. Skynet Application Porting Tutorial2.1. Notes and notebooks	15
3. Data aware scheduling	19
3.1 Installation	19
3.2 Usage	21
4. Resources balancing	24

1.

Monitoring of system and application metrics

1. Monitoring of system and application metrics

The Evolve consortium supports monitoring of applications and system level metrics by providing an installation of Prometheus and Grafana on the Nova cluster. Both services are allowed at the below addresses:

- Prometheus: <https://prometheus.platform.evolve-h2020.eu/>
- Grafana: <https://grafana.platform.evolve-h2020.eu/>

Most users won't need to access the Prometheus interface as it is fully integrated as a Grafana dashboard. Therefore, Grafana represents the one-stop-shop for most of monitoring needs, especially when pre-defined monitoring metrics are needed (e.g., containers CPU and memory utilization, etc.). Monitoring of custom applications is described in the next section. Please refer to Grafana and Prometheus documentations for details on how to use those tools

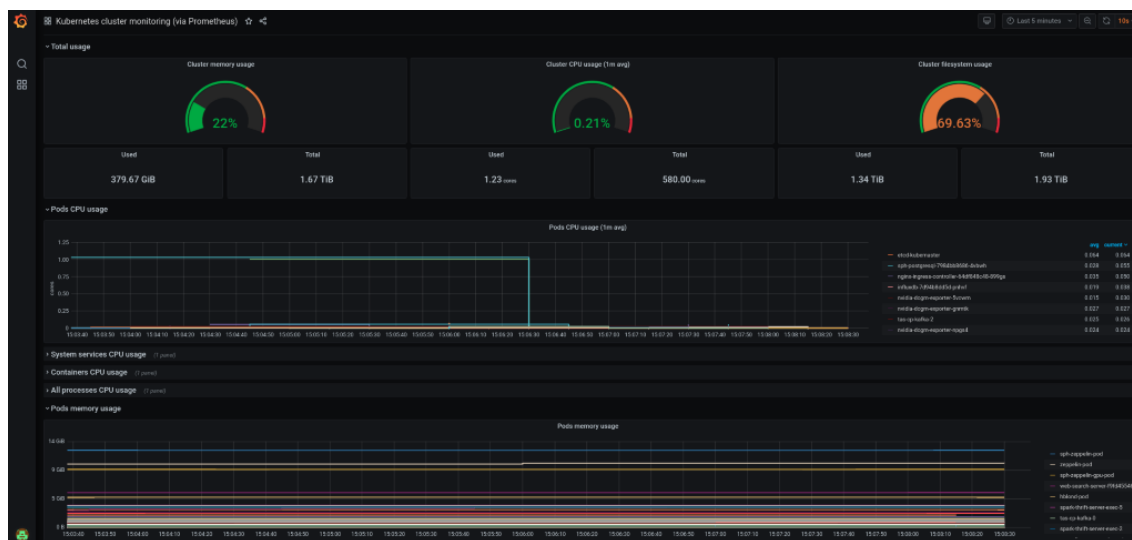


Figure 1: Example Grafana dashboard

1.1 Monitoring of custom applications metrics

Prometheus enables monitoring of custom user application metrics via the Prometheus client libraries, available for many programming languages either as officially supported or developed by third parties in the community. The process through which Prometheus collects metrics is called 'scraping' and exploits a web server exposed by the client library embedded in the application. The snippet below shows an example python application using the official Python Client Library.

As shown at line 11 in Figure 2 the client library starts an HTTP server on port 8080 that will be later regularly scraped by Prometheus. In this example we initialize all the four different metrics types supported by the official client library. The difference between the types is on how data is accumulated. As an example, a summary type provides the total number of observations as well as the sum of the observed values, while a gauge monitors a value that can arbitrarily increase/decrease at any point in time. It is up-to the application code to decide when to push a

new value to the various metrics (see lines 14-17 in the code snippet). The code can be then built into a docker container.

In order for prometheus to scrape the user containers we need to set some specific values in the in the Kubernetes pods specification. In the specific each container exporting Prometheus metrics must be bound to a service to enable access to the web server port, as well as annotate the container for scraping.

Figure 3 shows both an example service on port 8080 (lines 3-16) and the prometheus annotations (lines 34-35). The *prometheus.io/scrape: 'true'* informs Prometheus that this pod should be regularly scraped for data. At every scrape event the client library will dump all the data available for any metric defined in the code.

```
1 import prometheus_client as prom
2 import random
3 import time
4
5 if __name__ == '__main__':
6
7     counter = prom.Counter('my_counter', 'This is my counter')
8     gauge = prom.Gauge('my_gauge', 'This is my gauge')
9     histogram = prom.Histogram('my_histogram', 'This is my histogram')
10    summary = prom.Summary('my_summary', 'This is my summary')
11    prom.start_http_server(8080)
12
13    while True:
14        counter.inc(random.random())
15        gauge.set(random.random() * 15 - 5)
16        histogram.observe(random.random() * 10)
17        summary.observe(random.random() * 10)
18
```

Figure 2: Example python application using Prometheus client library.

```

1  ---
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: app-to-monitor
6    labels:
7      app: app-to-monitor
8  spec:
9    type: NodePort
10   ports:
11   - port: 8080
12     targetPort: 8080
13     protocol: TCP
14     name: http
15   selector:
16     app: app-to-monitor
17   ---
18   apiVersion: apps/v1
19   kind: Deployment
20   metadata:
21     name: app-to-monitor
22     labels:
23       app: app-to-monitor
24   spec:
25     replicas: 1
26     selector:
27       matchLabels:
28         app: app-to-monitor
29     template:
30       metadata:
31         labels:
32           app: app-to-monitor
33         annotations:
34           prometheus.io/scrape: 'true'
35           prometheus.io/port: '8080'
36       spec:
37         containers:
38         - name: app-to-monitor
39           image: 172.9.0.240:5000/monitoring/test-custom-metrics:latest
40           imagePullPolicy: Always
41           env:
42             - name: LISTENING_PORT
43               value: "8080"
44           imagePullSecrets:
45             - name: regcred
46
47

```

Figure 3: Example pod definition enabled to Prometheus scraping

After the application is running, users can verify it is being scraped by Prometheus by checking **Status -> Targets** and then search for the application named "app-to-monitor". If the target is up as in Figure 4, Prometheus is correctly scraping it.

http://10.244.6.20:8080/metrics	UP	<div>app="app-to-monitor" instance="10.244.6.20:8080"</div> <div>job="kubernetes-pods"</div> <div>kubernetes_namespace="karvdash-cpinto"</div> <div>kubernetes_pod_name="app-to-monitor-55fb5dfbc7-j29x5"</div> <div>pod_template_hash="55fb5dfbc7"</div>	839ms ago	88.7ms
---------------------------------	----	---	-----------	--------

Figure 4: Prometheus target for custom scraper.

Custom counters can be queried by name as in Figure 5 where the query is reading the value of the my_counter custom metric.

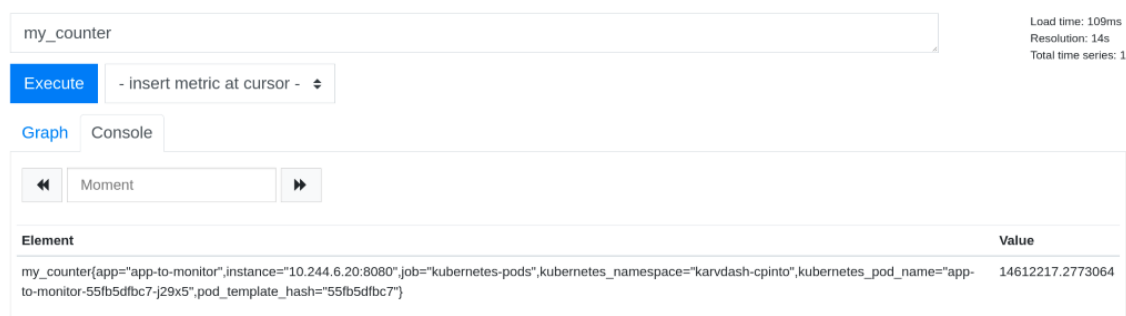


Figure 5: Querying a user defined metric from Prometheus

If you have access to a Grafana installation connected with Prometheus all the custom metrics are immediately accessible and can be integrated in any dashboard. The full sources of this example are available in the [Evolve Consortium Area](#).

1.2 IME monitoring through Prometheus

In order to deliver a comprehensive performance dashboard for all the components of EVOLVE, an important effect has been made towards the integration of IME monitoring through Prometheus.

The following sections present directives on how to access and find the right IME metrics for most end-users monitoring needs. All the metrics are accessible either from the [Prometheus UI](#) or through [Grafana](#) (details on those are given in Section 1 of the document)

CPU metrics

Different CPU metrics are available to monitor IME through Prometheus UI. Notice that a search bar with auto-completion is implemented (Figure 6).

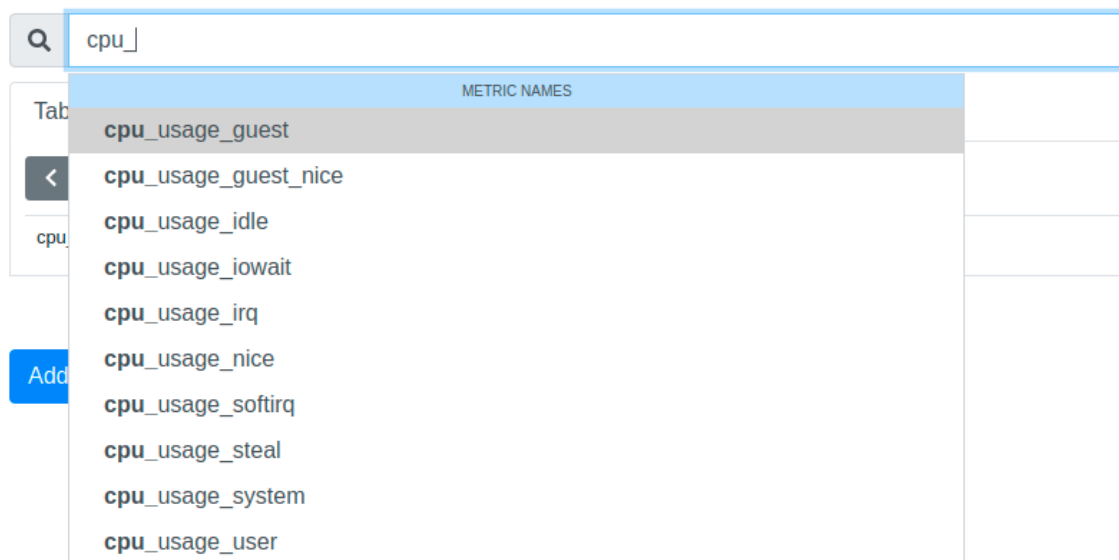


Figure 6: Different CPU metrics offered for IME. Autocompletion proposes a range of options for a query starting with the "cpu_" prefix.

The metrics above offer a variety of ways to monitor the CPU usage in IME.

More information on the different metrics can be found on Telegraf CPU Input Plugin official [Github page](#)

Once one of the metrics above is selected, a list of the last scrapped values or a comparative graph for all the IME servers (Table or Graph option respectively) is displayed.

It is then possible to narrow down the value to a specific IME server by specifying the host name in brackets as shown below (Figure 7).

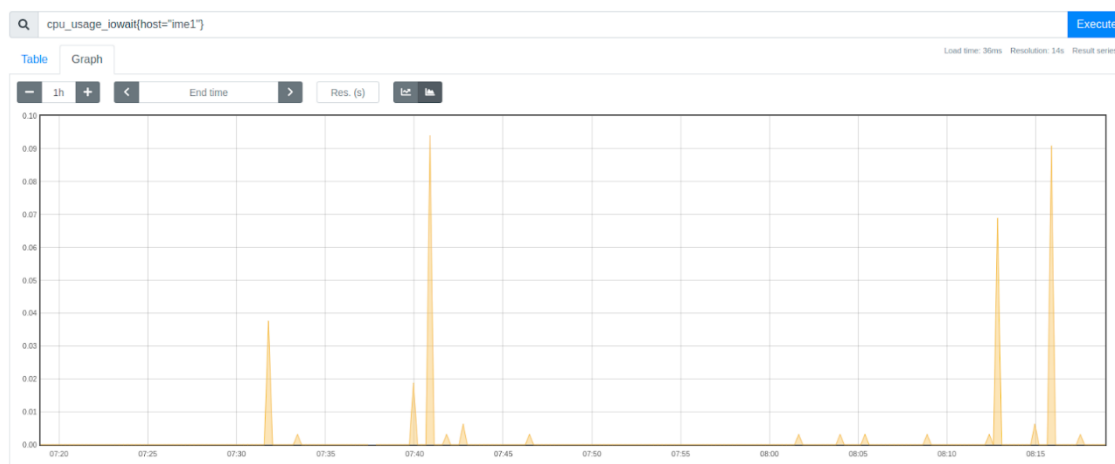


Figure 7: Graph of CPU I/O wait for ime1 for the last hour

As presented in the graph above, the interface offers a variety of options to explore and to filter performance data to suit at best end-user needs. One have the following options:

- Table / Graph: Get the last scrapped value or get a graph of the values through time
- For the Graph itself multiple display formats are supported:
 - Time range option
 - Time for beginning of monitoring
 - Resolution (Granularity of metrics in the graph)
 - Unstacked vs Stacked graph

Disk metrics

Disk usage in IME can also be monitored (Figure 8).

As for the previously discussed metrics, it is possible to explore the different metrics for Disk usage of IME through the Prometheus UI (e.g. starting a query with the “disk_” prefix) and the official Plugin [Github Page](#).

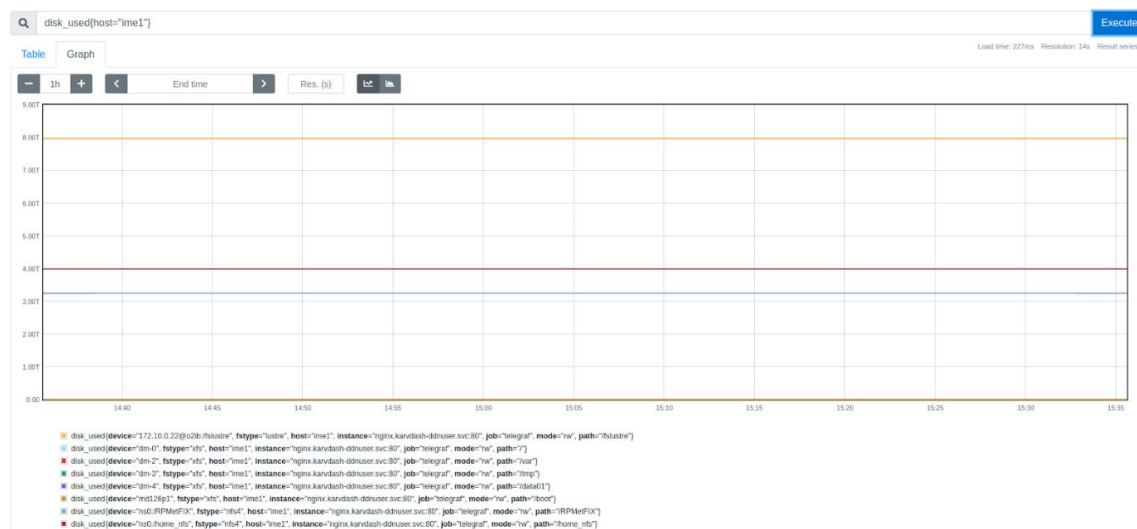


Figure 8: Comparative graph of the disk usage of the various file systems in host ime1

Memory metrics

As for the previously presented metric, it is possible to explore the different metrics for Memory usage of IME through the Prometheus UI (e.g. starting a query with the “mem_” prefix, Figure 9) and the official Plugin [Github Page](#).



Figure 9: Graph of free memory in host ime1 for the last 2 hours

System metrics

As for the previously presented metric, it is possible to explore the different metrics for (system load, uptime etc.)

In the same manner as above exploration of the different metrics for Memory usage of IME is possible through the Prometheus UI (e.g. starting a query with the “system_” prefix, Figure 10) and the official Plugin [Github Page](#).

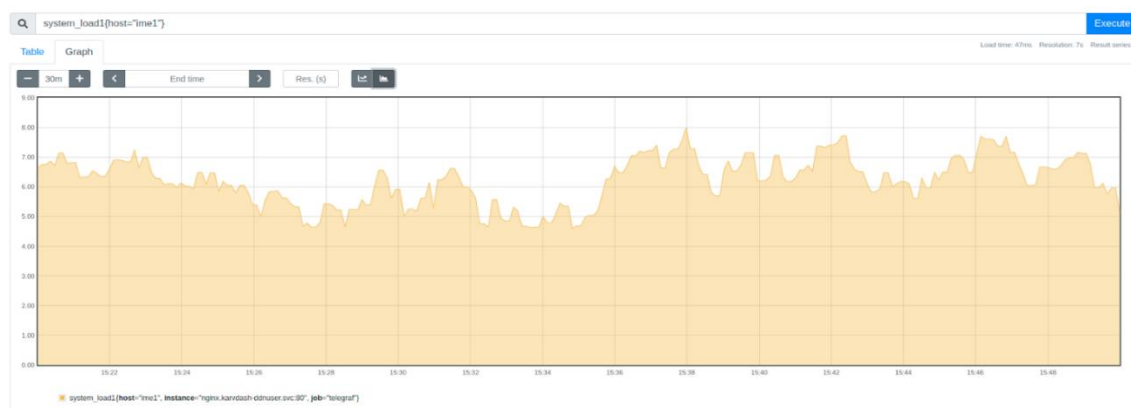


Figure 10: Stacked graph of the 1-minute load average for host ime1

IME native metrics

The purpose of this work is not limited to display already supported metrics by Prometheus, but to extend the scope of available information by integrating the IME native performance statistic. For instance, IME offers a variety of metrics like I/O activity, NVMe health and load, quotas etc.

In a similar way than for Prometheus 'toolbox' metrics, it is possible to explore the different metrics for Memory usage of IME through the Prometheus UI (query starting with the "exec_ime" prefix)

IME native statistics are exported to Prometheus as JSON formatted text. For instance, the excerpt below presents the JSON output of the quotas submodule of ime-monitor:

```
{
  "pid": 3848685,
  "pool_name": "evolve",
  "server_uptime": 1461607
},
{
  "view": "quotas-stats",
  "title": "QUOTAS TABLE",
  "nb_objects": 1,
  "data": [
    {
      "uids": [
        {
"uid": 64030,
          "nblocks": 9024,
          "blk_size": 131072,
          "size_used": 1182793728,
          "total_size": 960193626112,
          ...
        },
        ...
      ],
      "gids": [
        {
          "gid": 64030,
          "nblocks": 9024,
          "blk_size": 131072,
          "size_used": 1182793728,
          "total_size": 960193626112,
          "dirty": 0,
          ...
        }
      ]
    }
  ]
}
```

In Prometheus, in order select the size used by the User with uid 64030 (in **bold green** who has offset 0), the query for ime1 will be the following:

```
exec_ime1_quotas_data_0_uids_0_size_used
```

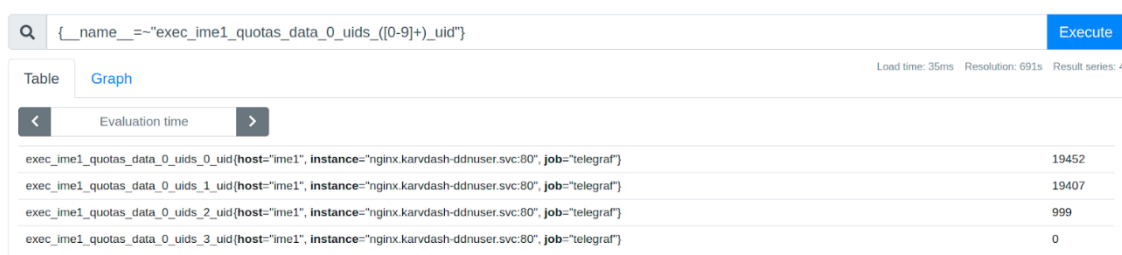
For the second appearing user the query would be

```
exec_ime1_quotas_data_0_uids_1_size_used
```

And so on.

Here the issue is that for a user of the Prometheus UI, his/her offset is an internal information not exposed to the end-user. Therefore an additional step is needed in order to find the associated user offset. This can be achieved by searching all the user uids present in the host and then by determining the offset of the user of interest. These operations will be realized by the following query:

```
{__name__=~"exec_ime1_quotas_data_0_uids_([0-9]+)_uid"}
```



Query	Value
exec_ime1_quotas_data_0_uids_0_uid(host="ime1", instance="nginx.karvdash-ddnuser.svc:80", job="telegraf")	19452
exec_ime1_quotas_data_0_uids_1_uid(host="ime1", instance="nginx.karvdash-ddnuser.svc:80", job="telegraf")	19407
exec_ime1_quotas_data_0_uids_2_uid(host="ime1", instance="nginx.karvdash-ddnuser.svc:80", job="telegraf")	999
exec_ime1_quotas_data_0_uids_3_uid(host="ime1", instance="nginx.karvdash-ddnuser.svc:80", job="telegraf")	0

Figure 11: Example of output of the query to output all the UIDS in ime1

Therefore, to get an an overview of the size used by all users, the query will be:

```
{__name__=~"exec_ime1_quotas_data_0_uids_([0-9]+)_size_used"}
```



Figure 12: Example of graph output of the query to determine the disk size used by all users in the host the last two days

2.

Skynet Application Porting Tutorial

2. Skynet Application Porting Tutorial2.1. Notes and notebooks

In order to port a service for Skynet we need to follow the steps below:

Step 1

As a first step we need to extract a performance metric that represents the actual performance of our service and export it periodically to a file inside our service's container. As an application example we are going to use a Tensor flow image processing service.

In the image below we extract the time needed for each batch iteration to finish and extract it in the report.txt file.

```
for i in range(nb_batches):
    tic2 = time.time()
    tiles1 = next(generator_im1)
    tiles2 = next(generator_im2)
    pred11 = encoder.predict_on_batch(tiles1)
    pred12 = encoder.predict_on_batch(tiles2)
    diff_tiles = LA.norm(pred11-pred12, axis=-1) #
    diff_list.append(diff_tiles)
    toc2 = toc2 + time.time() - tic2
    counter = counter + 1
    if counter == 10:
        with open('report.txt', 'w') as f:
            print(format(int(toc2)/10), file=f)
        counter = 0
        print("timer: {}".format(toc2))
        toc2 = 0
```

Figure 13: Example performance metric extraction for skynet

Step 2

As a second step we initialize a simple Node.js server inside the container. This server's listening port is 8080 and each time it receives a request it serves back the application's metric value as well the hostname of the container. Skynet uses Prometheus in order to ping services and gather the performance metrics.

In the image below there is a sample Node.js implementation.

```

var http = require('http');
var os = require('os');

var metric_value = 0;
console.log('count: %s', os.hostname());
http.createServer(function(request, response) {

    var fs = require('fs');
    var contents = fs.readFileSync('report.txt', 'utf8');
    console.log(contents);

    metric_value = Number(contents)
    response.writeHead(200);

    if (request.url == "/metrics") {
        response.end("custom_metric " + metric_value + "\n");
        return;
    }
    response.end(os.hostname() + ". " + metric_value + " \n");

}).listen(8080)

```

Figure 14: Sample Node.js implementation

Step 3

As a third step we have to create a Kubernetes configuration 'yaml' file in order to deploy our service. The table below contains the parameters we need to change inside the configuration in order to port our application for Skynet.

metadata->name	Deployment's name
metadata->annotations>app	Deployment's name
spec>template>metadata->annotations->app	Deployment's name
spec>template>spec->containers->image	The name of the application's Docker image
metric-type	The type of metric our application reports (0 if higher is better - 1 if lower is better)

In the image below we define a Kubernetes deployment running a Skynet compatible service.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: Tensor-Flow-Service
  labels:
    app: metrics #tag for metrics server
  annotations:
    metric-target-performance: "0.1"
    metric-type: 1 #use 0 if higher is better # use 1 if lower is better
    app: "Tensor-Flow-Service" #app tag
    type: "DATACENTER-JOB" #job type MPI/DATACENTER job
    resource: "1"
spec:
  replicas: 1
  selector:
    matchLabels:
      app: metrics #tag for metrics server
  template:
    metadata:
      annotations:
        app: "Tensor-Flow-Service"
        schedulerName: Skynet
    spec:
      schedulerName: Skynet
      containers:
        - name: Tensor-Flow-Service
          image: 192.168.1.213:5000/change-detection-tuned:latest
          imagePullPolicy: Always
          ports:
            - name: node-js-metric-server
              containerPort: 8080
          resources:
            limits:
              nvidia.com/gpu: 1

```

Figure 15: Example of Skynet enabled Kubernetes service

3.

Data aware scheduling

3. Data aware scheduling

Volcano is a batch job scheduler and management system for Kubernetes. It provides integrations for many distributed data analytic frameworks, such as Argo, Kubeflow, SparkOperator, PaddlePaddle, and Horovod among others. It provides Kubernetes-native implementation of batch job scheduling capabilities such as queue management, gang scheduling, backfill, job pre-emption and reclamation, and resource reservation. Volcano provides these capabilities through CRDs (Custom Resource Definitions) representing PodGroups, VCJobs, and Queues, and through three components: AdmissionController, Controller Manager, and the Scheduler.

3.1 Installation

The easiest method for installation is to use the Helm charts provided in the Volcano git repository.

```
$ git clone https://github.com/volcano-sh/volcano.git
$ helm install volcano installer/helm/chart/volcano --namespace <target_namespace>
```

Helm installation process can be customised to a specific namespace using the `target_namespace` parameter. An alternative method for installation is to deploy using the YAML file

```
$ kubectl create -f installer/volcano-development.yaml
```

Volcano is running when its three constituent deployments (Admission Controller, Controller, and Scheduler) are up and running. The Admission Controller examines jobs, PodGroups, and pods submitted to Kubernetes API Server for Volcano-specific annotations. The Controller Manager manages the lifecycle of the CRDs. The Scheduler maps the jobs to specific nodes based on resource management algorithms.

```
$ kubectl get deployments -n volcano-system
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
volcano-admission	1/1	1	1	76m
volcano-controllers	1/1	1	1	76m
volcano-scheduler	1/1	1	1	76m

The Volcano installation can be further customised through the configuration file (installer/helm/chart/volcano/config/volcano-scheduler.conf) which lists the plugins that are activated. Each plugin implements a scheduling filter or scoring method to refine the allocation of pods to nodes. The example below includes the plugin for data-aware scoring along with the predicates filter plugin and the priority scoring plugin.

```
$ cat installer/helm/chart/volcano/config/volcano-scheduler.conf
```

```
actions: "enqueue, allocate"
```

```
tiers:
```

- plugins:
- name: predicates
- name: datacache
- name: priority

3.2 Usage

A PodGroup represents a set of pods that are associated with each other. For example, the set of pods belonging to a single application framework (e.g. Spark) can be a PodGroup. PodGroups are the basis for the scheduling decisions in Volcano. VCJobs are Job specifications particular to Volcano, and are composed of Tasks, that can be customised with pod templates, lifecycles, and failure handling. VCJobs can be associated with priority classes. Internally, the tasks get translated to PodGroups and to individual pods. A Queue is a collection of PodGroups and can be limited to deploying pods on specific sets of nodes.

```
apiVersion: v1
kind: Pod
metadata:
  name: spark-dlf-runner
spec:
  ...
  containers:
  - name: spark-tpcds1g
    ...
    args:
    - /opt/spark/bin/spark-submit
    - --master
    - ...
    - --conf
    - spark.kubernetes.driver.podTemplateFile=/opt/spark/tpc-ds-performance-test/podtemplate.yaml
    - --conf
    - spark.kubernetes.driver.label.dataset.0.id=example-dataset
    - --conf
    - spark.kubernetes.driver.label.dataset.0.useas=mount
    - ...
    - --conf
    - spark.kubernetes.executor.podTemplateFile=/opt/spark/tpc-ds-performance-test/podtemplate.yaml
    - --conf
    - spark.kubernetes.executor.label.dataset.0.id=example-dataset
    - --conf
    - spark.kubernetes.executor.label.dataset.0.useas=mount
    - --class
    - SparkRunner
    - ...
```

The above figure shows listing of the Kubernetes YAML file for a Spark application using DLF. Only the relevant lines are shown in the listing. The pod named *spark-dlf-runner* executes *spark-submit* command that interfaces with the Kubernetes API server to create the Spark driver for the job. The Spark driver in turn creates the executors. The dataset information is supplied through the label *dataset.0.id* for both the driver and executor. When the driver and executor pods are created, the admission controller for DLF injects the volume information into the pod specification.

The pod template shown below is applied to both the driver and executor pods. The *schedulerName* property specifies which scheduler will handle the allocation of this pod as Kubernetes allows multiple schedulers in a single cluster. The annotation in 9 associates the driver and executors pod with a Volcano PodGroup (*spark-dlf-pg*). This PodGroup is further defined in the next figure. Volcano associates this PodGroup with a Job and any pods added to the PodGroup become Tasks that are then handled by the scheduler loop.

```
apiVersion: scheduling.volcano.sh/v1beta1
kind: PodGroup
metadata:
  name: spark-dlf-pg
  namespace: dlf
spec:
  minMember: 1
  queue: default
  minResources:
    cpu: 4
    memory: 16384m
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduling.k8s.io/group-name: spark-dlf-pg
spec:
  schedulerName: volcano
```

Currently, the data-aware scoring plugin is activated when three conditions are met:

- A job requires a dataset, and the nodes that host the gateways and data for the dataset are labelled dataset=<dataset_name>
- all the nodes are labelled with the topology keys kubernetes.io/hostname, topology.kubernetes.io/region, topology.kubernetes.io/zone, and rack
- the rack label should group the nodes accordingly

4.

Resources balancing

4. Resources balancing

Evolve platform supports an interference-aware scheduler as an option for application placement in the NOVA cluster. This custom scheduler is able to efficiently place applications on a Kubernetes environment. Using a universal approach for every kind of workload behaviour and duration, this framework aims to decrease application execution delays provoked by interference phenomena.

Evolve's interference-aware scheduler outperforms the default one of Kubernetes, improving the performance of the scheduled workloads, by efficiently equilibrating the usage of low-level system resources between the available machines. The integrated scheduler using socket-level metrics, and a custom scoring function firstly prioritizes the available nodes in respect to their most viable socket, places the application in the winning node and lastly pins the application to the proper socket. With this, interference in low level system resources is detected and avoided by the upcoming applications resulting to a more fair exploitation of the underlying systems of the cluster. As performance we consider the median execution latency in the deployed applications' distribution. You can find an extensive description in the [Evolve Consortium Area](#).

In order to make this tool available to the end user, a mutation webhook is used. Namespaces with the label **custom-scheduler-injector=enabled**, will trigger the mutation webhook by the time of application deployment. Afterwards, all the required configurations are applied, and the incoming workload will be placed in the cluster according to the interference-aware scheduler's policy.

Users that are willing to use this custom scheduler for their workload placement will need to label their namespace (<your-namespace>) with the required key-value pair mentioned above.

```
[user@kubemaster ~]$ kubectl label namespace <your-namespace>
custom-scheduler-injector=enabled
namespace/<your-namespace> labeled
[user@kubemaster ~]$ kubectl get namespaces -L custom-scheduler-injector
```

NAME	STATUS	AGE	CUSTOM-SCHEDULER-INJECTOR
cert-manager	Active	25h	
custom-scheduler-injector	Active	23h	
default	Active	3d21h	
dlf	Active	25h	
iccs-scheduler-apps	Active	2d14h	
ingress-nginx	Active	25h	
kube-node-lease	Active	3d21h	
kube-public	Active	3d21h	
kube-system	Active	3d21h	
<your-namespace>	Active	52s	enabled

evolve



Leading the Big Data
Revolution

in ◦ @evolve-h2020
 ◦ @evolve_h2020

info@evolve-h2020.eu
www.evolve-h2020.eu



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825061

DDN
STORAGE

Bull
data technologies

IBM

FORTH
Research Institute of Computer Science

SUNLIGHT

ETISSEN
ICCS

memoscale

webLizard
technology

LOBA

ThalesAlenia
Space

SPACE

CybeleTech
Technologies innovant pour le monde connecté

MemEx

NEUROCOM

Stiemme
TECHNOLOGIES

virtual vehicle

AVL

AVL

koola