KRATEO PLATFORMOPS

Krateo Composable FinOps

Abstract

Krateo's FinOps module addresses challenges in cost management and performance optimization for cloud-based services. It introduces a standardization layer using FOCUS and Prometheus, employs CrateDB for data storage, and leverages scrapers for data collection. The module analyzes data using optimization algorithms and machine learning, transforming insights into Kubernetes Custom Resources for automated action, offering a comprehensive solution for cloud cost optimization.

Contents

This document has been revised on November 21, 2024 and refers to Krateo Composable FinOps rel. 0.3.x. Previous document revisions:

- July 11, 2024
- September 7, 2024
- October 14, 2024 (rel. 0.2.x)

Introduction

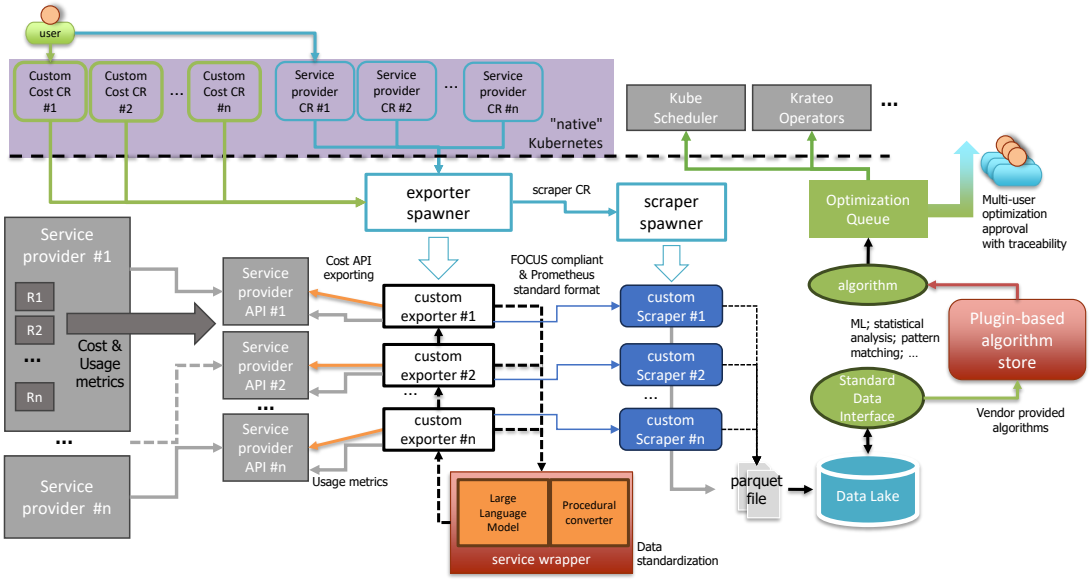

Figure 1: Overview of the Krateo Composable FinOps module architecture.

The FinOps module wants to help manage costs and optimize performance in cloud-based services, applications, and on-premises systems. The need for such a platform arises due to the increasing complexity of modern architectures, especially in distributed cloud applications.

To achieve such a platform, there are several complex challenges in the cost-collection process that must be addressed. The first challenge is the collection problem since gathering the required data from multiple sources takes time and effort. The second challenge is the capacity to store and analyze data because different sources might present their schema, making storing everything in a single, unified database challenging. The third challenge is handling and analyzing a large amount of data. Lastly, automation is needed to enable automatic cost collection, storage, and data processing, as well as planning for corrective actions to be taken when detecting wasted resources. Fortunately, Krateo's FinOps module addresses these problems pragmatically and modularly, allowing for ease of use, efficient performance, and customization.

The module introduces a standardization layer comprising two industry-standard specifications: FOCUS[1], the FinOps Foundation Cost and Usage Specification, and Prometheus. FOCUS is an industry-wide cost-measuring standard that helps standardize cost-related data presentation. We use this standard to achieve a common cost-collection starting point when connecting to the API endpoints. Then, we deploy a set of Prometheus exporters, which are placed as interposers between the cloud providers' APIs and our FinOps module, allowing for the translation and presentation of data in a completely standardized format.

The data storage problem is simplified thanks to the standardization layer, and it is solved using a data lake like database: CrateDB. This storage solution allows the use of a standardized communication format, the Parquet file, which helps with the quick transfer and storage of the standardized data coming from the exporting layer. To move the data from the export layer to the data lake, we use generic and configurable scrapers that can quickly move the data from the exporters to our data lake.

Once the data is stored in the data lake, we have another challenge: more than cost data is needed to optimize a cloud-based service. For example, a simple web server hosted on the cloud and a cost-optimization algorithm. The algorithm's goal is to reduce the cost as much as possible. With only the cost data available, this translates into shutting down the web server. This example shows how there is a need for additional information, namely, performance metrics from the virtual machine running the web server and the key performance metrics (e.g., average response time), along with a performance goal. We can use the information in the FOCUS report to obtain these data to deploy a new set of exporters targeting usage and performance metrics. Since our scrapers are generic and configurable, we can reuse them to upload the usage data to our data lake.

[1]The Krateo Composable FinOps currently supports the version 1.1 of the FOCUS.

We can analyze all the data collected in the data lake to find avenues for cost optimization. This can be achieved with multivariable optimization functions and statistical analysis with pattern recognition. Finally, we transform all optimizations into a set of Kubernetes Custom Resources (CRs) to act upon newly found cost-related deficiencies. This allows us to use Kubernetes operators (explicitly coded to interact with cloud services) to monitor these metrics and act automatically to apply changes to remote resources.

Krateo's FinOps module covers the entire spectrum of cost-related optimization, considering all the challenges and complexities involved in managing costs and optimizing performance in cloud-based services and applications. In the following chapters, each module will be analyzed in greater detail.

1 Cost Collection

The cost collection begins with the creation of one of two possible CRs from the user that represent two different types of costs:

- API-based costs;
- custom costs.

1.1 API-based Cost collection

In the first case (top of Fig. 1, in blue), we create a CR for the finops-operator-exporter containing the details required to connect to an API endpoint, including HTTP port, login information, etc. An exporter is automatically generated: finops-prometheus-exporter-generic. This exporter will connect to the specified API and download cost data. If the data already uses the FOCUS standard (e.g., from Azure, AWS, or GCP), it is exposed in the Prometheus format. Otherwise, if it is not in the FOCUS standard, it is converted with standard algorithms and artificial intelligence and then exposed in the Prometheus format. Additional information is also included in the CR to deploy a scraper, which allows the data to be stored in a database (it will be analyzed in greater detail in section 1.3. The CR is structured as follows:

```yaml
apiVersion: finops.krateo.io/v1
kind: ExporterScraperConfig
metadata:
  name: # ExporterScraperConfig name
  namespace: # ExporterScraperConfig namespace
spec:
  exporterConfig: # same as krateoplatformops/finops-prometheus-exporter-generic
    provider:
      name: # name of the provider config
      namespace: # namespace of the provider config
    url: # url including http/https of the CSV-based API to export, parts with <varName> are taken from additionalVariables: http://<varName> -> http://sample
    requireAuthentication: # true/false
    authenticationMethod: # one of: bearer-token, cert-file
    # bearerToken: # optional, if "authenticationMethod: bearer-token", objectRef to a standard Kubernetes secret with specified key
    #  name: # secret name
    #  namespace: # secret namespace
    #  key: # key of the secret
    # metricType: # optional, one of: cost, resource; default value: resource
    pollingIntervalHours: # int
    additionalVariables:
      varName: sample
      # Variables whose value only contains uppercase letters are taken from environment variables
      # FROM_THE_ENVIRONMENT must be the name of an environment variable inside the target exporter container
      envExample: FROM_THE_ENVIRONMENT
  scraperConfig:
    # ...
```

Figure 2: Example configuration to bootstrap the data collection for API-based costs.

If the field `metricType` is set to `cost`, then the API in the URL must expose a FOCUS report in a CSV file. Otherwise, if set to `resource`, it must expose usage metrics according to the JSON/OPENAPI schemas

available here and the field `additionalVariables` must contain a field `ResourceId` with the identifier of the resources to be used in the database as external key to reference the cost metric from the usage metric (i.e., the same as the field `resourceId` of the focusConfig CR).

1.2 Custom Costs collection

In the second case (top of Fig. 1, in green), in those instances where an API endpoint is unavailable, for example, to collect costs regarding the purchase of hardware or the power bill, we create a CR for the finops-operator-focus. This CR will contain all the information FOCUS requires regarding cost and some additional details related to the scraper configuration. The CR is then stored in the Kubernetes database. The finops-operator-focus then creates a configuration CR for the finops-operator-exporter, listing the Kubernetes API server as the API server to use to obtain cost data. The CR is structured as follows (some fields are omitted for clarity, see FOCUS for the complete list):

```
1  apiVersion: finops.krateo.io/v1
2  kind: FocusConfig
3  metadata:
4    name: focusconfig-sample
5  spec:
6    focusSpec: # See FOCUS for field details
7      availabilityZone:
8      billedCost:
9      billingAccountId:
10     billingAccountName:
11     billingCurrency:
12     billingPeriodEnd:
13     billingPeriodStart:
14     chargeCategory:
15     chargeClass:
16     chargeDescription:
17     chargeFrequency:
18     chargePeriodEnd:
19     ...
20     resourceId:
21     resourceName:
22     resourceType:
23     serviceCategory:
24     serviceName:
25     skuId:
26     skuPriceId:
27     subAccountId:
28     subAccountName:
29     tags:
30       - key:
31         value:
```

Figure 3: Example configuration for the collection of custom costs.

Info: You can also configure custom cost exports by creating exporters for an on-premises cluster, or use other services (e.g., OpenCost).

1.3 Scraper Configuration

Fig. 4 shows the CR used for the configuration of the scraper. This custom resource is included in either the CR from Fig. 2 or Fig. 3, inside the object `spec` (i.e., lines 6 and 5 respectively).

The finops-operator-scraper manages this configuration. It includes data regarding the database, the polling rate, etc. A scraper is automatically generated for each exporter: finops-prometheus-scraper-generic. Data is scraped from the exporters and stored in a data lake. This allows all the cost data to be stored in the same format and ready to be analyzed.

```
scraperConfig: # same fields as krateoplatformops/finops-prometheus-scraper-generic
  tableName: # tableName in the database to upload the data to
  # url: # path to the exporter, optional (if missing, its taken from the exporter service)
  pollingIntervalHours: # int
  scraperDatabaseConfigRef: # See above kind DatabaseConfig
    name: # name of the databaseConfigRef CR
    namespace: # namespace of the databaseConfigRef CR
```

Figure 4: Scraper Configuration Custom Resource.

1.4 Database Connection Configuration

To configure the scraper's access to the database (see `scraperConfig` in Fig. 4), we use an additional custom resource, the `databaseConfig` (see Fig. 5). This custom resource must include the username of the CrateDB instance and the reference to the secret with the CrateDB password.

```
apiVersion: finops.krateo.io/v1
kind: DatabaseConfig
metadata:
  name: # DatabaseConfig name
  namespace: # DatabaseConfig namespace
spec:
  username: # username string
  passwordSecretRef: # object reference to secret with password
    name: # secret name
    namespace: # secret namespace
    key: # secret key
```

Figure 5: Database Configuration Custom Resource.

1.5 Usage Data Collection

As mentioned in the introduction, more than the cost data is needed to perform any optimization (aside from reverse engineering usage from unit cost and total expenditure). This means that when a FOCUS report is obtained, the list of resources generating costs must also be collected. Thus, each exporter analyzes this list of resources and compares them to known providers/resources stored through CRs, allowing us to specify which additional metrics should be collected for each. The exporter-scraper CR indicates the provider as "provider." No operator manages the Custom Resource Definitions (CRDs) related to Providers, Resources, and Metrics. Figures 6, 7, and 8 show an example of a set of CRs to request the collection of the CPU usage of Virtual Machines on Azure:

```
apiVersion: finops.krateo.io/v1
kind: ProviderConfig
metadata:
  name: azure
  namespace: finops
spec:
  resourcesRef:
  - name: azure-virtual-machines
    namespace: finops
```

Figure 6: Example configuration of the Provider custom resource for additional data collection.

```yaml
apiVersion: finops.krateo.io/v1
kind: ResourceConfig
metadata:
  name: azure-virtual-machines
  namespace: finops
spec:
  resourceFocusName: Virtual machine
  metricsRef:
  - name: azure-vm-cpu-usage
    namespace: finops
```

Figure 7: Example configuration of the Resource custom resource for additional data collection.

```yaml
apiVersion: finops.krateo.io/v1
kind: MetricConfig
metadata:
  name: azure-vm-cpu-usage
  namespace: finops
spec:
  metricName: Percentage CPU
  endpoint:
    resourceSuffix: /providers/microsoft.insights/metrics?api-version=2023-10-01
  timespan: month
  interval: PT15M
```

Figure 8: Example configuration of the Metric custom resource for additional data collection.

While optimizing resource utilization does allow for the reduction of energy consumption and CO2 emissions, these can also be included in the collected data, allowing for the explicit optimizations of various aspects of the infrastructure.

1.6 Database Handler

The finops-database-handler manages all the data uploaded to the database (i.e., CrateDB), acting as a proxy for all requests. It is a webservice that offers various endpoints, offering also the capability of easily computing data directly on the database through a notebook functionality that allows the user to define custom code.

The webservice exposes several endpoints:

- `POST /upload`: the webservice receives the data (divided into chunks) and directly uploads it into the specified table in the database
- `POST /compute/<compute_name>`: calls the specified compute notebook with the POST body data being the parameters required by the given algorithm, encoded in JSON as: $parameter\_name = parameter\_value$
- `POST /compute/<compute_name>/upload`: uploads the specified notebook into the dedicated database table, using as identifier the name <compute_name>
- `GET /compute/list`: lists all available compute notebooks

Which can be used as follows:

- Upload endpoint:

Command Line

```
curl -X POST -u <db-user>:<db-password> \
"http://finops-database-handler.finops:8088/upload?
table=<table_name>&type=<cost|resource>" \
    -d "<metrics>"
```

- Compute endpoint:

Command Line

```
curl -X POST -u <db-user>:<db-password> \
http://finops-database-handler.finops:8088/compute/cyclic \
    --header "Content-Type: application/json" \
    --data '{"table_name":"testfocus_res"}'
```

- Compute endpoint upload:

Command Line

```
curl -X POST -u <db-user>:<db-password> \
http://finops-database-handler.finops:8088/compute/cyclic/upload \
    --data-binary "@cyclic.py"
```

- Compute endpoint list:

Command Line

```
curl -u <db-user>:<db-password> \
http://finops-database-handler.finops:8088/compute/list
```

1.7 Installation of the Data Collection Module

We use Helm charts to install the data collection components of the Krateo Composable FinOps module. This standardizes and streamlines the installation process. To perform the installation:

Command Line

```
$ helm repo add krateo https://charts.krateo.io
$ helm repo update krateo
$ helm install finops-operator-exporter krateo/finops-operator-exporter
$ helm install finops-operator-scraper krateo/finops-operator-scraper
$ helm install finops-operator-focus krateo/finops-operator-focus
```

We recommend adding the "-n <namespace>" to each install instruction for the three operators to avoid installing them in the "default" namespace, i.e.:

Command Line

```
$ helm install -n finops finops-operator-exporter krateo/finops-operator-exporter
$ helm install -n finops finops-operator-scraper krateo/finops-operator-scraper
$ helm install -n finops finops-operator-focus krateo/finops-operator-focus
```

The deployment of the finops-prometheus-exporter-generic and finops-prometheus-scraper-generic is made automatically by the respective operators. A "repository" variable in the charts' value file contains the target registry from which to deploy the container images (and it can be different from the operator registry).

Finally, install the database handler:

Command Line

```
$ helm install finops-database-handler krateo/finops-database-handler -n finops
```

1.8 Data Lake

We upload the data from the scrapers to a data lake. Specifically, we use an instance of CrateDB (see bottom right of Fig. 1). To allow the scrapers to upload the data, we utilize the REST APIs and the information in the database configuration CR to select the proper database/table. Additionally, we use the notebook functionality, which mimics the behavior of a stored procedure in this instance, allowing for the safe upload of data without exposing requests containing SQL code. The notebook is integrated into the database handler webservice.

1.8.1 CrateDB Installation

CrateDB can be installed through its operator. The operator can be installed through an Helm chart:

Command Line

```
$ helm repo add crate-operator https://crate.github.io/crate-operator
$ helm repo update
$ helm install crate-operator crate-operator/crate-operator \
    --namespace crate-operator \
    --create-namespace \
    --set env.CRATEDB_OPERATOR_DEBUG_VOLUME_STORAGE_CLASS=azurefile-csi-premium
```

Info: Make sure to select the appropriate storage class, especially when installing in a managed environment. In this example, CrateDB Operator is being installed in an Azure managed cluster, with storage class `azurefile-csi-premium`.

To create an instance of the database, you can use the following CR:

```yaml
apiVersion: cloud.crate.io/v1
kind: CrateDB
metadata:
  name: cratedb-cluster
  namespace: finops
spec:
  cluster:
    imageRegistry: crate
    name: crate-dev
    version: 5.9.2
  nodes:
    data:
    - name: hot
      replicas: 2
      resources:
        limits:
          cpu: 2
          memory: 4Gi
        disk:
          count: 1
          size: 32GiB
          storageClass: azurefile-csi-premium
        heapRatio: 0.25
```

Figure 9: Example configuration of the CrateDB operator to create a cluster composed of two nodes.

2 Optimization

2.1 Notebooks on the Data Lake

Currently, we have uploaded all data related to cost and all data related to the usage metrics we care about. Now, it can be analyzed through another set of notebooks, which can be scheduled to run periodically and provide suggestions and optimizations (see top right of Fig. 1). We coded a Python notebook that analyzes the usage patterns and searches for periodic peaks, such as business hours utilization. It then finds when the business hours start and end and proposes an optimization for scaling up/down the involved instances.

The code to calculate the optimization is run by the finops-database-handler (see Section 1.6). Its code is also injected with the following python code to configure the access to the database. The code is injected at line 0:

```
import sys
from crate import client
def eprint(*args, **kwargs):
    print(*args, file=sys.stderr, **kwargs)
host = sys.argv[1]
port = sys.argv[2]
username = sys.argv[3]
password = sys.argv[4]
try:
    connection = client.connect(f"http://{host}:{port}", username=username, password=password)
    cursor = connection.cursor()
except Exception as e:
    eprint('error while connecting to database' + str(e))
    raise
```

The variables `cursor` and `connection` allow the user to connect to and query the database without explicitly coding this information inside the notebook. The following code shows the aforementioned notebook:

```
import pandas as pd
import numpy as np

# Global variables
json_template = {'resourceId': '', 'optimization': None}
json_template_optimization = {'resourceName':'', 'resourceDelta':0, 'typeChange':None}
json_template_type_change = {'cyclic':'', 'from':'', 'to':''}

TIME_GRANULARITY = 5  # in minutes

def compute(df: pd.DataFrame, resource_id: str, metric_name: str):
    time_durations = [int(60/TIME_GRANULARITY)]  # in minutes
    _, moving_average_values = moving_average(df, np.min(time_durations))
    usage, usage_max = utilization_per_unit(df, moving_average_values, 'd')
    proposed_optimization = optimize(usage, usage_max)
    window_low_utilization = find_start_finish_window_low_utilization(proposed_optimization)

    json_template_type_change['cyclic'] = 'day'
    json_template_type_change['from'] = str(24 + window_low_utilization[2] if window_low_utilization
        [2] < 0 else window_low_utilization[2]) + ':00'
    json_template_type_change['to'] = str(window_low_utilization[1] + window_low_utilization[2]) + '
        :00'
    json_template_optimization['resourceName'] = metric_name
    json_template_optimization['resourceDelta'] = -window_low_utilization[0]
    json_template_optimization['typeChange'] = json_template_type_change.copy()  # Added .copy()
    json_template['resourceId'] = resource_id
    json_template['optimization'] = json_template_optimization.copy()  # Added .copy()

    print(json_template)

def moving_average(df: pd.DataFrame, time_duration: int) -> tuple[list, list]:  # Fixed return type
    hint
    moving_average_values = [0.0 for _ in range(len(df['average']))]
    result = []
    contiguos = False
    for i in range(len(df['average'])):
        partial = 0.0
        cur_index = i
        for _ in range(cur_index, cur_index + time_duration):
            if cur_index < len(df['average']):
                partial += df['average'][cur_index]
                cur_index += 1
        partial = float(partial)/float(time_duration)
        moving_average_values[i] = partial
        if partial > np.average(df['average']) and contiguos:
            result[-1] = (result[-1][0], result[-1][1]+1)
        elif partial > np.average(df['average']):
            result.append((df['timestamp'][i], 1))
            contiguos = True
```

```
47          else:
48              contiguos = False
49      return result, moving_average_values
50
51  def utilization_per_unit(df: pd.DataFrame, moving_average_values: list, unit: str) -> tuple[list, list
        ]:
52      if unit == 'd':
53          fields = 24
54      usage = [0.0 for _ in range(fields)]
55      usage_max = [-1.0 for _ in range(fields)]
56      usage_counter = [0 for _ in range(fields)]
57
58      for index, value in enumerate(moving_average_values):
59          if value > np.average(df['average']):
60              usage[df['timestamp'][index].hour] += value
61              usage_counter[df['timestamp'][index].hour] += 1
62          if value > usage_max[df['timestamp'][index].hour]:
63              usage_max[df['timestamp'][index].hour] = value
64
65      for index, _ in enumerate(usage):
66          if usage_counter[index] != 0:
67              usage[index] = usage[index] / float(usage_counter[index])
68
69      return usage, usage_max
70
71  def optimize(usage: list, usage_max: list) -> list:
72      result = [0 for _ in range(len(usage))]
73      for index, value in enumerate(usage):
74          left_over_average = 100 - value
75          left_over_max = 100 - usage_max[index]
76          if left_over_average > 0 and left_over_max > 0:
77              smaller_left_over = min(left_over_average, left_over_max)
78              result[index] = int(smaller_left_over)
79      return result
80
81  def find_start_finish_window_low_utilization(proposed_optimization: list) -> tuple:
82      lowest_length = (proposed_optimization[0], 0, 0)
83      lowest_length_temp = (proposed_optimization[0], 0, 0)
84      for index, value in enumerate(proposed_optimization):
85          if value != lowest_length_temp[0] and value != lowest_length_temp[0] + 1 and value !=
                lowest_length_temp[0] - 1:
86              if lowest_length_temp[1] > lowest_length[1]:
87                  lowest_length = (lowest_length_temp[0], lowest_length_temp[1], lowest_length_temp[2])
88              lowest_length_temp = (proposed_optimization[index], 0, index)  # Fixed: Added index
89          else:
90              lowest_length_temp = (lowest_length_temp[0], lowest_length_temp[1]+1, lowest_length_temp
                    [2])
91
92      # Check backwards if index is 0
93      if lowest_length[2] == 0:
94          for index in range(1, len(proposed_optimization)):
95              if proposed_optimization[-index] == lowest_length[0] or proposed_optimization[-index] ==
                    lowest_length[0] + 1 or proposed_optimization[-index] == lowest_length[0] - 1:
96                  lowest_length = (lowest_length[0], lowest_length[1] + 1, -index)
97              else:
98                  return lowest_length
99      return lowest_length
100
101 def main():
102     table_name_arg = sys.argv[5]
103     table_name_key_value = str.split(table_name_arg, '=')
104     if len(table_name_key_value) == 2:
105         if table_name_key_value[0] == 'table_name':
106             table_name = table_name_key_value[1]
107     try:
108         resource_query = f"SELECT DISTINCT ResourceId FROM {table_name}"
109         cursor.execute(resource_query)
110         resource_ids = pd.DataFrame(cursor.fetchall(), columns=['ResourceId'])
111
112         for resource_id in resource_ids['ResourceId']:
113             metric_query = ("SELECT DISTINCT metricName\n"
114                 f"FROM {table_name}\n"
115                 "WHERE ResourceId = ?"
116             )
117             cursor.execute(metric_query, (resource_id,))
118             metric_names = pd.DataFrame(cursor.fetchall(), columns=['metricName'])
119
120             for metric_name in metric_names['metricName']:
121                 data_query = ("SELECT *\n"
122                     f"FROM {table_name}\n"
123                     "WHERE ResourceId = ?\n"
124                     "AND metricName = ?\n"
125                     "ORDER BY timestamp"
126                 )
127                 cursor.execute(data_query, (resource_id, metric_name))
```

```
                raw_data = cursor.fetchall()
                column_names = [desc[0] for desc in cursor.description]

                table_compute = pd.DataFrame(raw_data, columns=column_names)

                # Convert timestamp column to datetime
                if 'timestamp' in table_compute.columns:
                    table_compute['timestamp'] = pd.to_datetime(table_compute['timestamp'], unit='ms')
                    table_compute['timestamp'] = pd.to_datetime(table_compute['timestamp'], unit='ms')
                        .dt.tz_localize('UTC')

                if 'average' in table_compute.columns:
                    table_compute['average'] = pd.to_numeric(table_compute['average'], errors='coerce'
                        )

                compute(table_compute, resource_id, metric_name)

    finally:
        cursor.close()
        connection.close()

if __name__ == "__main__":
    main()
```

2.1.1 Optimization Example

The following figure reports the utilization of an Azure virtual machine:

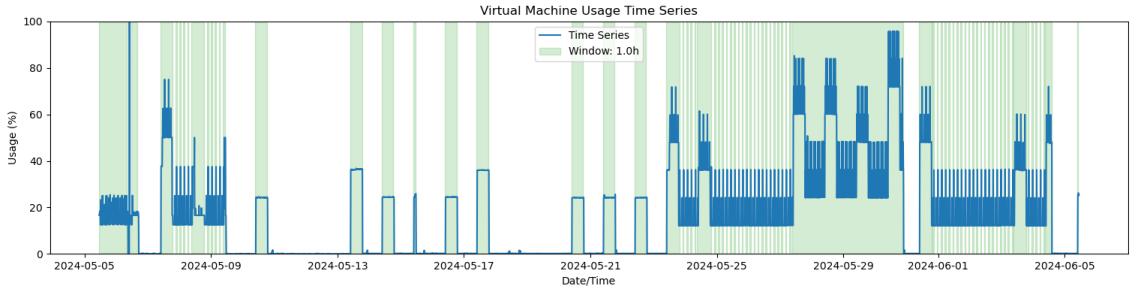

Figure 10: Azure Virtual Machine CPU usage time series. Green highlights the identified “high usage” areas.

The areas of the plot that have a green background have been marked to identify them as “high usage” windows and, therefore, to be analyzed further. The optimization algorithm splits all windows’ utilizations into 24 slots, representing each hour of the day. It then becomes apparent that the high-load times are happening during business hours:

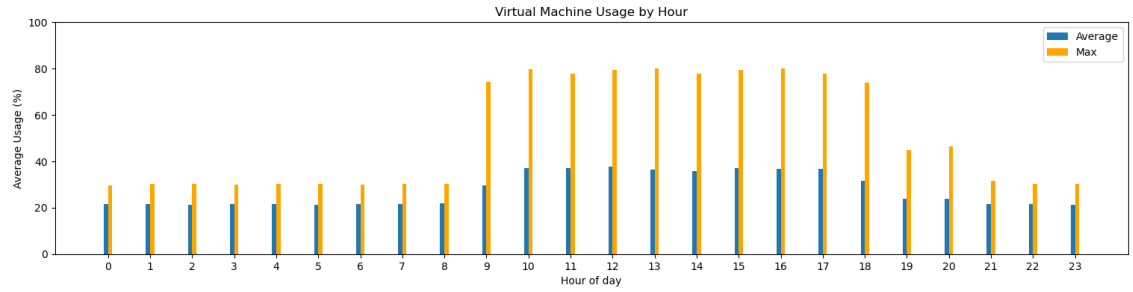

Figure 11: Azure Virtual Machine CPU usage time series split into the daily hours.

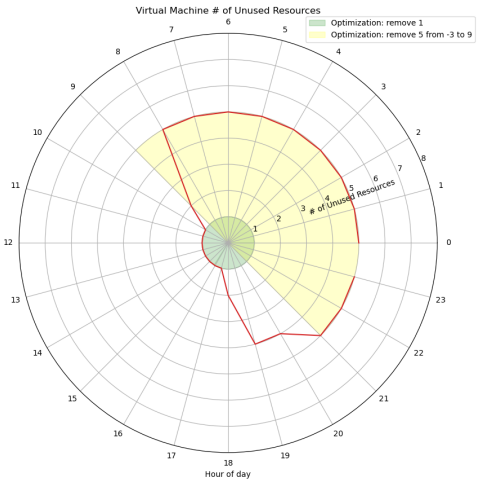

Figure 12: Polar axes plot representing with the red line the unused CPU cores of a Virtual Machine. The center of the axes is zero, and the outermost line is eight. Green shows how many CPU cores can be removed, and the yellow area shows the unused resources during "off hours."

This representation makes it easy to observe how the average and maximum CPU utilization are higher from 9:00 a.m. until 8:00 p.m. This allows us to deduce that, outside of these hours, there are wasted resources, as we can see from the polar axes plot on the left:

- even during peak-load hours, there is always at least one unused CPU core because the red line that represents unused CPU cores never reaches zero. This could allow the VM instance to be scaled down to a smaller one, reducing costs and wasted resources;
- From 9:00 p.m. to 9:00 a.m., during the night hours, the VM is also not being fully utilized since the red line hovers around five CPU cores unused, meaning in this instance, more than half of the compute capacity: this could prompt for a scale-down optimization during the night.

These two observations will be used to optimize the Virtual Machine: firstly, for right-sizing, and secondly, to determine when to periodically scale down. This should allow for a reduction of costs without meaningfully impacting performance.

2.2 Optimization Forwarding to Kubernetes

The computed optimization is packaged into a *JSON* file and sent to a REST API web service on Kubernetes that publishes the optimizations on a NATS service: finops-http-rest-queue. These optimizations are published on a topic specified as a query parameter in the HTTP POST message through the "topic" keyword. A set of NATS subscribers gathers the optimizations: finops-nats-subscriber. These subscribers are notified when new data is published and read the JSON file from the topic. The information can now be used for two main goals:

- forward the optimization to the Krateo operator that manages the services that need to be optimized, for example, the Azure Operator to modify the size of a Virtual Machine;
- send the optimization through a multi-group/multi-user authorization process, for example, to be approved by the Platform team, the Acquisition team, and Management.

In the first case, the optimization is automatically encoded in a CR for the finops-operator-vm-manager, which then analyzes it and decides how to manage the Virtual Machine. For example, it could scale up or down the virtual machine, stop it for the night, etc. In the second case, an approval queue is used to reach each approval group the optimization requires. The users receiving approval requests will be decided based on the Kubernetes integrated permission system (RBAC)[2].

2.3 Optimization Module Installation

The installation of the optimization components first requires installing the NATS service. It can be deployed using Helm:

Command Line

```
$ helm repo add nats https://nats-io.github.io/k8s/helm/charts/
$ helm repo update
$ helm install -n finops nats nats/nats
```

[2]This feature is not available in rel. 0.3.0.

After installing NATS, deploy the finops-http-rest-queue through a regular deployment file. If using a managed Databricks on Azure, the deployment requires a Service of type "NodePort", to allow external traffic to reach the container inside the cluster. Otherwise, you can also use Helm:

Command Line
```
helm install -n finops finops-http-rest-queue krateo/finops-http-rest-queue
```

Then, install the finops-nats-subscriber using Helm. Ensure to configure each subscriber with the correct parameters in the values file of the chart:

Command Line
```
git clone https://github.com/krateoplatformops/finops-nats-subscriber-chart.git
nano . . .
helm install -n finops finops-nats-subscriber-1 ./finops-nats-subscriber-chart
```

Finally, install the finops-operator-vm-manager:

Command Line
```
helm install -n finops finops-operator-vm-manager krateo/finops-operator-vm-manager
```

3 Use Case Example

This section presents two examples on the possible uses of the Krateo Composable FinOps module. The first one showcases a scenario where a set of virtual machines are underutilized and can be scaled down. The second example presents a use case where the system targets the optimization of a virtual machine's CO_2 emissions, by performing the deployment in a region where the energy coming from grid comes from green sources

3.1 Utilization Optimization

Let there be a company that runs its cloud through Microsoft Azure services. The company requires a web service to allow users to upload a data form through a secure interface. A set of virtual machines is deployed on the cloud to operate the web service through multiple instances for load balancing. These instances are then all connected to a single database. The web service targets 750ms as end-to-end compute time and a maximum of one hundred thousand concurrent users. It is continuously running but is only actively used during business hours.

Setup: The setup is based on the Azure Virtual Machine of instance size E64-32as_v4, which has 32 virtual CPUs and 512GB of system memory. The average CPU utilization during the day never peaks above 50%, and the target response time is always below 500ms with a peak number of concurrent users on average close to 80000. However, During the night, the CPU utilization drops close to zero, and the average user count is only in the dozens. This means there is room for cost improvement without compromising the target performance.

Optimization: Data collected by the Krateo Composable Finops module from Azure includes resource usage, such as CPU usage percentage and available memory. Additionally, the web service has been configured to expose its end-to-end latency for processing, allowing Krateo to include this data in its optimization. We observed from the data collection that since CPU usage never peaked above 50%, half of the virtual machine's CPU cores were unused. This means that the target performance should not be impacted by scaling to an instance size E64-16as_v4, with only 16 CPUs. Secondly, the virtual machine is seldom

accessed at night and could be scaled down even further to an instance E32-16as_v4, to be scaled up during the day.

Result: The average cost of E64-32as_v4 and E64-16as_v4 instances is $\approx$ 2700 dollars a month. The E32-16as_v4 costs $\approx$ 1400 dollars. Assuming 12 hours per day of E64-32as_v4 and 12 hours of E32-16as_v4, we obtain an optimized monthly cost of 1300 dollars per Virtual Machine, a 48% cost reduction for the same performance.

3.2 Carbon Footprint Optimization

Let there be a company running cloud services across multiple regions, in this instance, Azure cloud. This company aims to minimize its carbon footprint by optimizing virtual machine placement based on the availability of green energy through the selection of location, availability zones, etc. Using the Krateo Composable FinOps module, the company can ingest data regarding energy grids in various regions, identifying which cloud data centers rely heavily on renewable energy sources like solar or wind. When workload demand increases, the system can dynamically target those regions where green energy is most abundant, reducing CO2 emissions. For instance, during peak solar hours in a region, the module transfers more workloads there to take advantage of clean energy.

4 Conclusions

The Krateo Composable FinOps module allows for the standardization of the collection, optimization, and deployment of cost-reduction optimizations, allowing for more streamlined operations and higher resource visibility. In turn, it also allows for better management of cloud resources and on-premises costs and reduces the energy and CO2 impact on the environment.