TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Rasmus Rüngenen 164640IAPB

SOLUTION FOR KUBERNETES RESOURCE MONITORING

Bachelor's thesis

Supervisor: Ago Luberg
MSc

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Rasmus Rüngenen 164640IAPB

KUBERNETESE RESSURSSIDE MONITOORINGU LAHENDUS

bakalaureusetöö

Juhendaja: Ago Luberg
MSc

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Rasmus Rüngenen

05.01.2021

Abstract

Kubernetes is a container orchestration platform, which allows to configure and coordinate the lifecycle of containers. Kubernetes is originally made by Google and was released to the public in 2014 June.

The main purpose of this thesis is to implement the Kubernetes resource monitoring solution in Pipedrive. This was necessary because Kubernetes platform was introduced to Pipedrive, which at the time, had no way of monitoring.

In this thesis the plausible open-source solutions on the market were analysed against a set of requirements. These requirements were reliability, metrics usability, the total amount of metrics, and adjustability to Pipedrive's requirements. After the analysis, the author concluded that there are no suitable open-source solutions that would comply with the requirements. Due to no suitable solutions, the creation of custom implementation was required. The exporters' architecture was created keeping in mind the goals of data gathering time constraints, modularity, and the external configuration possibility of metrics and the requirements from the analysis.

As a result of this implementation, the author built a solution that follows previously set requirements and was deployed to the Pipedrive production environment as a core solution for alerting and monitoring of Pipedrive’s services resource usage.

After the implementation and deployment to the production environment, the author presents an overview related to the exporter in the production environment. The author also gathered feedback regarding the built solution from Pipedrive engineers and the responses were positive.

This thesis is written in English and is 31 pages long, including 8 chapters, 9 figures and 4 tables.

Annotatsioon

Kubernetese ressursside monitooringu lahendus

Kubernetes on konteinerite haldamise platvorm, mille abil on võimalik hallata ja koordineerida konteinerite elutsüklit ja tööd. Kubernetes on loodud Google poolt ning tehti avalikusele kättesaadavaks 2014. aasta juunis.

Lõputöö peamine eesmärk on implementeerida Kubernetese ressursside monitooringu lahendus Pipedrive'ile. Selle eesmärgi motivaatoriks on Kubernetese platvormi kasutusele võtmine antud firmas, mille tõttu oli vaja uut monitooringu lahendust.

Selle lõputöö raames analüüsis autor turul olevaid vabavaralisi lahendusi kindlaid kriteeriumeid jälgides. Nendeks kriteeriumiteks olid töökindlus, meetrikate kasutatavus ehk mis kujul need on, meetrikate koguarv koos vajamineva kettapinna suurusega ning vastavus Pipedrive’i nõuetele. Analüüsi tulemusena selgus, et olemasolevad lahendused ei vasta kriteeriumitele, seega otsustas autor implementeerida lahenduse. Antud rakenduse andmevaramuna kasutatakse Prometheusi. Selle lahenduse planeerimisel pidas autor meeles kolme eesmärki. Esiteks andmevaramu poolt tulevaid ajapiiranguid, mis standardina on 30 sekundit. Teiseks rakenduse modulaarsust ning kolmandaks meetrikate konfiguratsiooni eraldatust rakendusest. Lisaks peeti meeles ka analüüsis käsitletud kriteeriumeid.

Autor implementeeris rakenduse edukalt Pipedrive’i toodangu keskkonda ning see võeti kasutusele peamise konteinerite ressursside monitooringu rakendusena firmas. Antud lahenduse väljundit kasutatakse igapäevaselt antud firmas erinevate meeskondade poolt. Peamine eesmärk antud lahenduse väljundil on võimaldada näha ressursside kasutuse taset ning garanteerida teenuste stabiilsus. Lisaks on võimalik anda häiret, kui teenuse ressursside kasutused lähenevad kriitilisele piirile kasutades implementeeritud lahenduse meetrikaid.

Peale rakenduse edukat rakendamist toodangukeskkonnas küsis autor tagasisidet Pipedrive’i inseneridelt, kes kasutavad oma töös antud rakenduse meetrikaid. Tagasiside oli positiivne ning insenerid leidsid, et kokkuvõttes on tegemist rakendusega, mis aitab neil igapäevaseid toiminguid lihtsustada Kubernetese platvormil.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 31 leheküljel, 8 peatükki, 9 joonist, 4 tabelit.

List of abbreviations and terms

API	Application Programming Interface
cAdvisor	Container Advisor
CI/CD	Continuous Integration/Continuous Development
CNCF	Cloud Native Computing Foundation
FIFO	First in First out
GB	Gigabyte
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
K8s	Numeronym of Kubernetes
Kube	Abbreviation of Kubernetes
kube-state-metrics	Kubernetes state metrics
MB	Megabyte
Metric	Time series that is defined by its unique name and optional key-value pairs called labels
PromQL	Prometheus Query Language
REST API	Restful Application Programming Interface
Scraping	Prometheus configuration task, what defines the endpoints, where data should be retrieved.
SIG	Special Interest Group
TSDB	Time series database
UID	Unique Identifier
VM	Virtual Machine
YAML	YAML is a human-readable data-serialization language

Table of contents

List of figures

List of tables

1 Introduction

Cloud Native Computing Foundation, an organization that incubates open-source cloud solutions such as Kubernetes, releases yearly a survey that investigates the adaptation of cloud-native services. According to their 2019 survey report, 78 percent of the respondents are using the Kubernetes container orchestration platform in their production and testing environment. In 2018 this same percentage was 58, which means that the adoption rate has increased strongly [1].

Datadog, one of the leading monitoring service providers in the industry, has witnessed that containerization adoption has been on the rise since 2017. According to their research, over fifty percent of their customers' workloads are running in the Kubernetes container orchestration platform. Kubernetes has gained more traction and is considered the most widespread container orchestration platform according to their research conducted in 2020 November [2].

The incentive for this thesis is the introduction of Kubernetes platform to Pipedrive. Due to this, a new tool for monitoring was needed to gain visibility about the resource usage of the services running on the new platform. Kubernetes has a more complicated architecture than the previous container orchestration platform. Previous container orchestration platform was Docker Swarm [3]. This requires thorough analysis of the platform and the plausible points for monitoring.

This thesis consists of seven sections. The Introduction is the current section. The second section is about the background and gives an overview of the topics to give more context to this thesis. The third section analyses the existing open-source solutions for Kubernetes resource monitoring. The fourth section discusses the proposed method for the custom implementation of Kubernetes resource monitoring. The fifth section gives an overview of the implementation results with the iterations made during the development. The sixth part describes the implementation in the production environment. It also provides gathered feedback from engineers. The seventh section proposes ideas for future outcomes and the eight section is the summary of this thesis.

2 Background

In this section, an overview of background technologies, methods, and definitions will be introduced and explained, giving more knowledge related to the proposed method and analysis.

2.1 Monitoring

“From a technology perspective, **monitoring** is the tools and processes by which you measure and manage your technology system” [4].

It also supplies the insight and ability to diagnose, detect and alert based on the data that is being gathered. This data is a representation of the experience that customers are seeing while using your product. There is a common misconception that only technological teams can receive help from monitoring. Business departments can also make decisions based on the reports that allow to measure the state of product and technological investments [4].

To have a solid monitoring set for application development, it must be considered as a core component rather than an additional component. It is best to have metrics and monitoring taken into consideration during the planning phase [4].

2.2 Kubernetes

Kubernetes is an open-source platform for managing containerized workloads released to the public in June 2014 by Google [5].

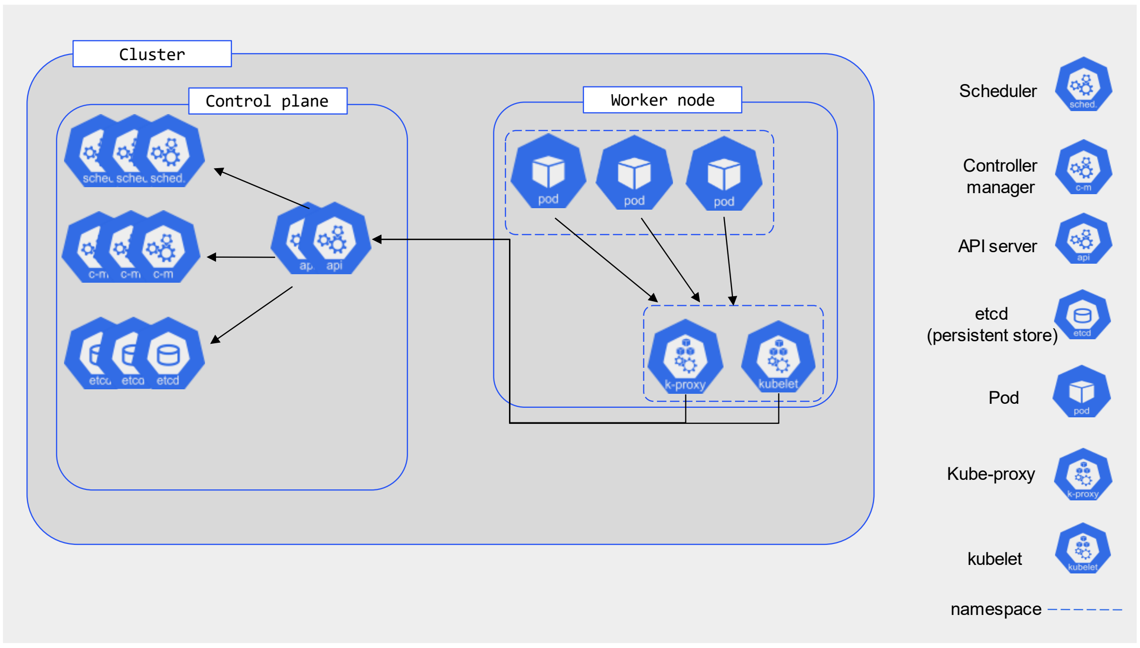


Figure 1. Example high-level architecture of Kubernetes cluster, while on the figure there is one control plane and one worker, there can be N amount of both inside one cluster. [6]

Cluster is the highest object in Kubernetes. It consists of control planes and workers. **Control plane,** referred to as the master node in some writings, is a set of components on virtual machines (VM) or bare-metal servers that handle making the global decisions about the cluster and responding to the state of the cluster. Control plane components can be on any node in the cluster. For simplicity, user-defined containers and pods are not running typically on the same machines, thus requiring a set of worker nodes. **Worker nodes** or **nodes** are a set of VMs specifically meant for running the pods and containers inside one cluster [7]. Control planes handle managing the cluster and workers are nodes where pods are scheduled. **Pods** can have a single container or a group of containers that share the same space. It can be thought of as a capsule to host multiple containers on one host [8]. **Containers** are an abstraction layer for a lightweight and portable executable Docker image that holds software and all its dependencies [9].

Every container runs at least one **workload** declared in the manifest file. **The manifest** file is a set of configurations that allow to add, change, or delete Kubernetes objects [10]. The most common workload is running a service that has been built into a Docker image.

For an added explanation regarding Figure 1, the following information is given. The **kubelet** can be thought of as an application that runs on each Kubernetes node and checks its status. It registers and controls the node it is running on with the Application

Programming Interface (API) server [11]. **API server** provides a Restful API (REST API), through which the one or more kubelets validate Kubernetes objects and if needed configure them. **Kube-proxy** acts as a networking interface between the worker node and control plane nodes. **The scheduler** assigns pods to nodes according to First in First Out (FIFO) and decides if the pod can be scheduled to a specific node. **Controller manager** is a component that runs the overall processes inside the Control plane node [7]. **Etcd** is key-value storage that has the configuration data, state, and metadata needed for the Kubernetes cluster [12].

2.3 Monitoring Kubernetes

Kubernetes monitoring was initially tested using Zabbix [13]. It is a host monitoring platform designed for real-time system metrics collection using their proprietary agents, that collect resource usage through these agents [14]. Initially the setup was made for one of the test environments clusters so that the agents would register every Kubernetes pod as one host.

The problem with Zabbix is that it is not meant for dynamic environments, where hosts that are monitored are disappearing and appearing. In Pipedrive there are over five hundred deployments each week, and every deployment creates a set of new pods which caused Zabbix to register them as new hosts while keeping the old hosts and their data due to an internal retention period. This meant that within a week, the high number of abandoned hosts started causing problems for the underlying Zabbix instance. After this testing phase, it was clear that there was a need for another more dynamic platform to monitor services running inside Kubernetes.

The next platform that was taken into consideration is Prometheus. Prometheus had much more promise than Zabbix since it did not have any relation to the resources in Kubernetes. One of the advantages that Prometheus has over Zabbix is that there are Kubernetes monitoring endpoints already outputting the data in a format that Prometheus accepts.

2.4 Prometheus

Prometheus is a systems and services monitoring toolkit built by Soundcloud in 2012 [15]. It was developed as an in-house monitoring system since solutions at that time were not enough for their needs [16]. In May 2016, it was accepted to Cloud Native Computing Foundation (CNCF) as its second project after Kubernetes [17]. Prometheus ecosystem consists of a server, alertmanager, pushgateway and user interface.

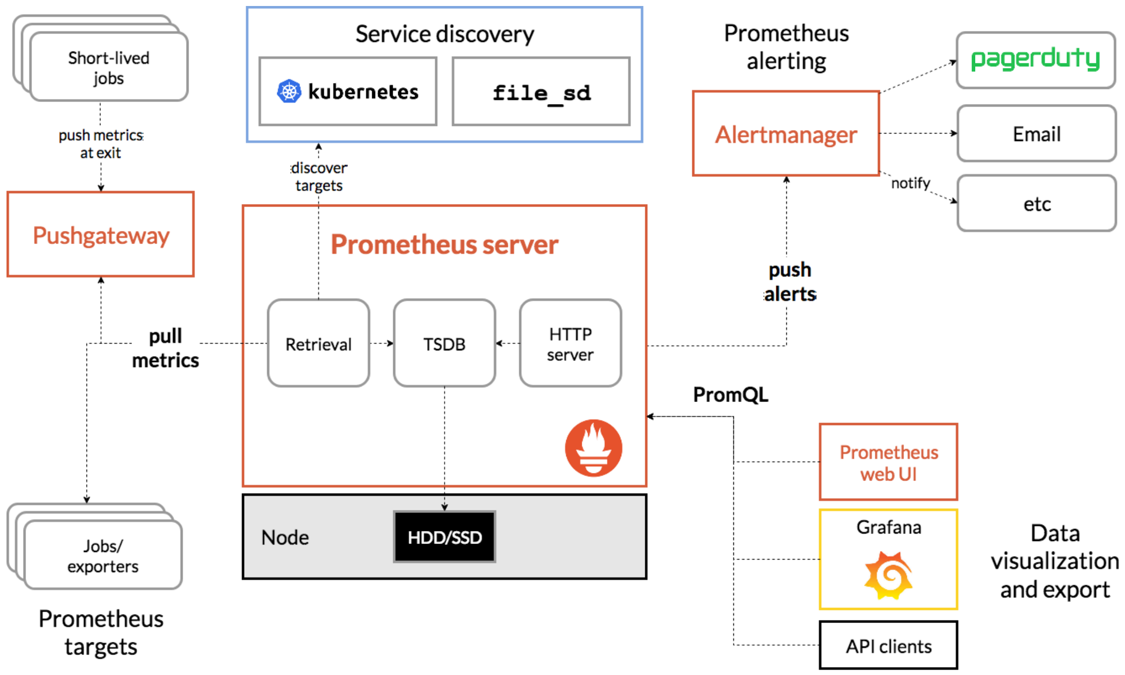


Figure 2. The example architecture of Prometheus [15]

The **server** consists of a time series database (TSDB) for data storage and a Hypertext Transfer Protocol Secure (HTTPS) server for data retrieval or sending. Prometheus defines its workloads using scraping. **Scraping** is a Prometheus configuration task, what defines the endpoints, where data should be retrieved. It also allows for the manipulation of data before storing it inside a TSDB. By default, **Prometheus collects** data every thirty seconds from its targets [15]. This data collection timing can also be modified in the configuration of a scrape.

Prometheus has a proprietary query language called **Prometheus Query Language** (PromQL) that provides the ability to query and aggregate data stored inside TSDB [15].

The **alertmanager** provides the ability to create alerts on the metric and its values ingested into Prometheus. Its configuration is stored inside of the Prometheus and it has integrations to other alerting platforms and the ability to send out alerts over webhooks [18].

The **pushgateway** allows external services that do not expose metrics to constantly send data to Prometheus. One of the upsides of it is that it allows services to send metrics to Prometheus that might not be present during the data collection. This solution is good for batch job-based services. Pushgateway does not change or aggregate data provided to it [19].

Prometheus server also has a **user interface** to query and visualize metrics using PromQL. Unfortunately, it lacks the possibility to create dashboards. Every time a user wants to see some data, they have to re-query it. Thus, Prometheus is mostly used as a storage for data. Visualization and dashboards for services are commonly sent to another service such as Grafana.

2.4.1 Prometheus metrics

Prometheus stores all data as time series. Time series is a set of data points ordered by time [20]. **Metric** is a time series that is defined by its unique name and optional key-value pairs called labels [21]. Prometheus is suited for **numeric time series**, meaning the values of each metric must be numeric [22]. Since Prometheus metrics are numerical, it is possible to apply mathematical functions such as sum or rate to the metrics [23].

With each metric, there are two **metadata identifier**s for Prometheus to consume during ingestion called #HELP and #TYPE. The identifier **#HELP** provides a string that contains the metric name and gives a short explanation of the metric. **#TYPE** provides the type of the metric, such as Gauge, Counter, Histogram, Summary [24], [25]. These two metadata identifiers are important to Prometheus as it will define the type of metric before the writing to the TSDB. Before the type definition, each metric is set to untyped.

```
# HELP <metric_name> <Description>
# TYPE <metric_name> <type>
<metric_name>{<label name>=<label value>, ...} <metric_value>
```

Figure 3. Default Prometheus metric model

```
# HELP api_http_failed_requests_total Total failed requests to endpoint with
method
# TYPE api_http_failed_requests_total counter
api_http_failed_requests_total{method=”POST”, handler=”/items”} 241
```

Figure 4. Example of Prometheus metric

Due to the architecture of Prometheus, every unique combination of key-value labels creates a new time series, which can dramatically increase the cardinality of metrics stored in TSDB. Prometheus recommends not using label values that have high dimensionalities such as unique user IDs or email addresses [26].

2.4.2 Prometheus exporters

Prometheus has a vast ecosystem of exporters, which are all prebuilt by the open-source community such as CouchDB or Elasticsearch exporter. This allows for faster metrics gathering for the users and the creation of more complex monitoring systems out of the box. Prometheus has official instrumentation libraries for Go, Java or Scala, Python, and Ruby languages. There are other unofficial third-party client libraries for Node.js, C, and PHP [27]. Custom written exporters have the possibility to create custom business logic related metrics that can be monitored such as requests made to a specific endpoint in the last five minutes or the average duration of a specific database query.

2.4.3 Prometheus as a data storage

The author decided to choose Prometheus as the data storage for this project. One of the primary reasons the author chose Prometheus was that the Kubernetes has support for Prometheus [28], [29]. Another point is that Prometheus does not need to have relation with the underlying service as it just accepts the metrics and stores them inside the TSDB (see more in section 2.3). And the final reason for the choice was that Pipedrive already had Prometheus running in the production environment that made it convenient to start sending metrics to it.

3 Analysis of existing solutions

In this analysis, there were three open-source solutions picked as a candidate for the Kubernetes monitoring solution - Container Advisor, kube-state-metrics, and metrics-server. The solutions were picked due to the first solution being provided by Google, while the last two are provided by Kubernetes.

During the analysis, four categories were picked: reliability of the solution, the usability of metrics when creating dashboards, the total amount of metrics, and the adjustability to Pipedrive's requirements. The usability of metrics is being analysed from the labels point of view as the labels are related to the grouping and visualization of metrics. Pipedrive’s requirements were the ability to have custom metric names and custom labels. An example of custom labels is the team name, who owns the pod or container, or repository name label where the source code for the container is found.

3.1 Container Advisor

Container Advisor (cAdvisor) provides metrics about container resource usage. This service is developed and maintained by Google [30]. cAdvisor can be run standalone or in Kubernetes case, the metrics are integrated into kubelet (see more in section 2.2 above) making it easy to use as no configuration is needed.

Since it is built into kubelet, the discovery of containers is native and out of the box meaning that no configuration must be done additionally other than enabling the metrics in the kubelet declarative manifest.

3.1.1 Reliability

Pipedrive used cAdvisor metrics before monitoring the containers in Docker Swarm. This product was therefore considered to have a prominent level of reliability and due to it being inside of kubelet binary the reliability of this metric exporter is considered highest.

3.1.2 Metrics usability

Container Advisor supplies multiple default labels that can be used to aggregate the data. Most metrics have the following labels: container, id, image, name, namespace, pod. The **container** label provides the name of the container, for example, calico-node. The label **id** has the Kubernetes pod class with the Quality-of-Service class. The label **image** is the Docker image running inside the container with the tag. The final label is a **name**.

3.1.3 Total amount of metrics

The Container Advisor gives out around 7760 metrics per worker. In Pipedrive there are 30 worker nodes per environment and per region, that will sum up to 231 800 metrics per one scrape. To gather the data for an hour with intervals set as 30 seconds would result in 300 Megabytes (MB) or 0.3 Gigabytes (GB) per hour of data. This would mean that the data storage would need to be over 7.2 Gigabytes to store 24 hours of data collected only from that source.

Table 1. cAdvisor unique metrics by type and the total amount of metrics

Type	**Unique metrics**	**Total lines of metrics**
CPU	13	1808
Memory	12	2332
Network	9	1666
Filesystem	18	860
Other (Tasks, information, last seen)	7	1093

In Pipedrive the general retention of metrics is 30 days so that would add up to 216 Gigabytes. There are 59 unique metrics exposed by the cAdvisor.

3.1.4 Adjustability to Pipedrive’s requirements

This solution does not supply any label remapping or renaming of metrics on the exporter side. If there is a need to drop and not store some metrics, it can be done through Prometheus in-flight ingestion rules, which will put more pressure on the Prometheus server. In-flight ingestion rules allow for data to be tested against a set of rules before saving them to TSDB.

One solution to more metrics would be to fork the project and rewrite the creation of internal metrics to support other labels. This solution has a major downside as it would mean keeping the codebase always up to date with the upstream solution and that eventually creates unnecessary development overhead.

3.2 kube-state-metrics

Kubernetes state metrics (kube-state-metrics) is a service that listens to Kubernetes APIs and aggregates the data to Prometheus format about the state of objects inside the cluster. This service is developed and maintained by the Kubernetes project. Its focus is on nodes, pods, and deployments health metrics inside the platform [31]. One important note is that kube-state-metrics does not have resource metrics about pods and containers. Data output by this service is solely related to the Kubernetes cluster. So, to get the metrics about pods, there needs to be a hybrid data collection combined with other out of the box solution.

3.2.1 Reliability

Since this service is officially developed and maintained by the Kubernetes project, they emphasize the reliability and uniformity of the metrics and have the same grade stability as the Kubernetes API objects themselves. While analysing the stability through restarts and errors of this service over 3 months, there were no downtime witnessed on this service.

3.2.2 Metrics usability

Kube-state-metrics is oriented more towards the analytics of the cluster state. Most metrics have component, group, scope, resource, and verb labels. The **component** label provides the component that this metric is about, for example, api server or etcd (see more in section 2.2). **Group** gives an overview if this metric is about apps or other Kubernetes internal services such as Calico or events. The **scope** is the object that the metric is connected, for example, cluster or namespace. The **resource** label is the same as the scope. The **verb** is the action that this metric observes such as get, post, watch, list.

3.2.3 Total amount of metrics

Kube-state-metrics data is exposed from the control plane nodes contrary to Container Advisor metrics, which are exposed from each worker node from the cluster. This means that metrics are aggregated across the control plane and produce a less overall number of metrics.

This service gives out a total of 33154 metrics per scrape. For a US-based test datacentre that would mean 99 462 lines of metrics per one scrape. Since data is once again gathered every 30 seconds then one hour of metrics will add up to 672 Megabytes or 0.672 GB.

Table 2. kube-state-metrics unique metrics by type and the total amount of metrics

Type	**Unique metrics**	**Total lines of metrics**
API server	27	27841
Etcd	3	4499
Rest client	3	200
Workqueue	7	415
Other (open API, process information)	51	199

For one day of data collection the needed storage, in this case, accumulates to 16.128 GB and the 30 days retention 483.84 GB of storage. There are 91 unique metrics exported by kube-state-metrics.

3.2.4 Adjustability to Pipedrive's requirements

Kube-state-metrics like cAdvisor does not offer any adjustability of metrics out of the box. This means the adjustability relies on changing the code of the solution which can be considered time consuming.

3.3 metrics-server

The metrics-server belongs to the Special Interest Group (SIG) community of Kubernetes [32]. This project is maintained and developed by the open-source community. It collects the data from kubelets running on nodes and its main output is to be used by other

Kubernetes services such as Horizontal Pod Autoscaler. While other out of the box solutions expose Prometheus metrics as a response, this service exposes metrics in the JSON format [32]. Metrics-server must be deployed manually, and it does not come built-in to the Kubernetes cluster.

3.3.1 Reliability

The author analysed the reliability metrics-server by looking at the restarts count and errors over three months and found no significant issues. While this project is based on open-source contributions the quality gate of the latest changes is managed the same way as is the Kubernetes project itself, ensuring the stability and quality of releases.

3.3.2 Metrics usability

Since the metrics-server outputs the data in the JSON format, the comparison of labels cannot be made.

3.3.3 Total amount of metrics

It is not possible to compare the number of metrics with other solutions as this service does not give out metrics in Prometheus format. The output of this is gathered from the metrics-server pod on one of the control plane nodes. Adjustability to Pipedrive's requirements

Since this output is significantly different than other out of the box solutions the adjustability must be handled on another service or consumer that scrapes these metrics. Prometheus cannot consume JSON metrics.

3.4 Summary

Existing solutions offer a variety of metrics for engineers to visualize the data. From container CPU and memory usage to Kubernetes cluster internal metrics such as the state of etcd or API server (see more in section 2.2). These metrics are useful since they can expose any outliers and issues with the Kubernetes internal or custom services running inside the cluster.

On the other hand, the Container Advisor does not expose any metrics related to pods, while it does have a pod label. Pod labels can be used to aggregate the data, although it would cause a lot of pressure on the data visualization user interface, where metrics are queried. Also, there are no indications on what node the pods/containers are in the cluster, making the debugging of potential issues complicated.

Another problem that came out of the analysis is the amount of data required to store, with 7 GB and 16.128 GB for cAdvisor and kube-state-metrics respectively per day. That amount of storage must be taken into consideration as the total amount of storage needed is related to data retention. With metrics-server, the data retention could not be compared directly to other out of the box solutions since it does not have the Prometheus style output.

In Pipedrive the default storage retention is 30 days. That would mean 699.84 GB of storage only for Kubernetes cluster metrics. To compare the size, in the Pipedrive production environment, the joint disk space for Prometheus installation is 700 GB and that holds all the metrics ranging from services metrics to infrastructure metrics with the data retention of 30 days.

Table 3. Metrics storage evaluation

Source	**Size of scrape in Megabytes**	**Size of scrapes per day in Gigabytes**	**Total storage required for 30 days in Gigabytes**
Container Advisor	2,5	7,2	216
kube-state-metrics	5,6	16,128	483,84

Another key problem is the label adjustability as neither cAdvisor, nor kube-state-metrics supply the possibility to configure other labels to metrics nor change metric names without forking the solutions and writing additional code. This causes bad usability because the labels do not match across multiple metrics and are vaguely defined.

4 Proposed method

In this section, the proposed method for the developed service will be presented with an overview of its architecture goals, technological stack, metrics and labels, and local development environment.

After the analysis of plausible existing solutions in the market, the results were presented to engineers in the Monitoring and Infrastructure team. One of the primary concerns from Infrastructure engineers was the requirement of storage that would increase rapidly due to the total amount of metrics that must be stored for 30 days. Monitoring engineers also suggested that the adjustability of those is below expectations and thus suggested the development of customized service.

The author of this thesis decided to create a solution to present all the needed features brought up in the analysis of existing solutions. The author did not plan to create a solution that would be a direct competitor to existing solutions in terms of speed of the application or the resources the service consumes. Instead, plan was for a service that would export resource usage from Kubernetes cluster and would be easily expandable to add more resources later.

4.1 Proposed Architecture

Kubernetes service exporter consists of three parts:

a) Data gathering and storing

b) Metrics creation and configuration

c) Exposing of metrics

The first and most important goal for the architecture was the **time restriction**. Prometheus has a default timeout for gathering data from exporters set to thirty seconds (see more in section 2.4). That means the request, data gathering, and parsing, with the creation of metrics, must fit into this time restriction.

The second goal was **modularity**. The point was to separate the parts of code in the same ways as objects are separated in the Kubernetes ecosystem. This means that pods, deployments, services, nodes, and clusters should have different handlers inside the service. For example, deployments resource would have its handler file called deployments.js, which handles the data collection and format before storing it in an in-memory storage. This allows to add more resources by creating a new handler to match the resource and expand it without the need to change the whole codebase.

The third point was the **external metrics configuration method**. This was solved by, after each data gathering the metrics configuration is requested from an external source (see more in section 5.3). One of the key ideas behind this would be to have the possibility to change the metrics configuration without the need to cause an interruption to the exporter by redeploying it. Having interruptions in the service that should supply data for alerting and monitoring purposes is considered a bad practice. The external metrics configuration provides a satisfactory solution to change metrics or even add new ones.

4.2 Service technological stack

This service programming language is Node.js. One of the reasons for using Node.js is that this programming language is supported by Pipedrive's Continuous Integration/Con tinuous Development (CI/CD) platform and has support for Pipedrive's internal libraries.

The service must also have a web server that serves the `/metrics` endpoint to expose the data in Prometheus format. For that, the decision was to use the Express package [33] as it has proven to be reliable for other exporters in the production environment of Pipedrive.

For faster array iteration and object parsing, the author decided to use *Lodash*, as it has many built-in functions to for instance reduce an array of objects to a subset of objects without having to write the implementation from scratch.

For the creation of the metrics, Prometheus does not have a Node.js library that is maintained by their organization. Instead, they have opted to support a third-party client that offers metrics implementation out of the box, so the author opted to use the *prom-client* package.

For metrics storage, there was a decision made to store the metrics in Consul [34] key/value store. To access the Consul K/V store for metrics configuration, Pipedrive's internal *Diplomat* package is used. It allows accessing the configuration stored inside the regions Consul.

To get the data from the Kubernetes API, the *k8s* package is used. The main reason is the simplicity of authentication handling between client and API while keeping the ability to query all the objects exposed by the API. It also can use a KUBECONFIG flag (see more in section 5.4) or read the rights directly from the file.

4.3 Pre-requisites for metrics and labels

Before development, there had to be precise prerequisites set to achieve a reliable and scalable solution. One of these prerequisites was creating a list of metrics and labels for each metric that were required.

That also included keeping the same label format throughout the metrics, for example, pod_name and podName label names have the same values, while making it difficult to remember which metrics had corresponding label names. It is also useful to have the same label names, because Prometheus can apply mathematical aggregation functions over some labels. The initially planned metrics and labels can be found in Appendix 2.

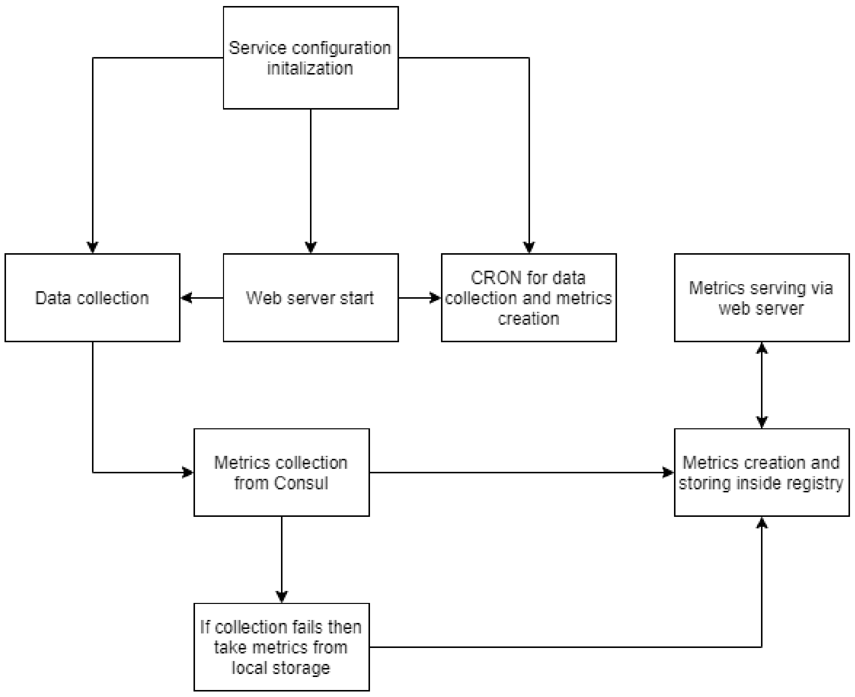
5 Implementation

In this section, the results of the development of the Kubernetes resource monitoring service, also known as kube-scraper, will be presented with the results of its architecture, data gathering and storing, and the metrics creation and exposing.

The most time-consuming part of this implementation was data gathering and storing. One of the reasons was the unknown responses of the Kubernetes API that the author had to analyse and then make decisions on the data that would be stored in the in-memory storage.

5.1 Service initialization with the first cycle

When the service starts, the first step is the **configuration initialization**. During that the in-memory storage is created for the Kubernetes objects, Kubernetes API configuration is set based on the environment the exporter is running in, and the libraries are initialized.



Service configuration initalization
Data collection
Web server start
CRON for data collection and metrics creation
Metrics serving via web server
Metrics collection from Consul
Metrics creation and storing inside registry
If collection fails then take metrics from local storage

Figure 5. Service initialization with the first lifecycle

After the configuration is initialized three different parts of the service start in parallel. The first part is the **data collection triggering**, that starts collecting and storing the data from Kubernetes API. The second part is the **creation of a web server** that has the metrics endpoint and listens to a port specified in the manifest (see more in section 2.2) of this service. The third part is the **initialization of a CRON job** that collects data and creates metrics every thirty seconds.

After running three parallel steps, the next step is **collecting metrics configurations** from Consul. The configuration is stored inside service memory until the triggering of the next metrics collection. If the collection from Consul fails, then there is a fail-safe built into the exporter that allows the default metrics to be exposed.

The fourth step is the **creation and storing of metrics** based on the data stored in memory.

The last step is the **exposing of metrics** through the metrics endpoint of the exporter. This part is valid if the exporter receives the request of metrics from Prometheus before the CRON job has created its first cycle.

5.2 CRON job for data collection and metrics

The purpose of this CRON job is to make handling of the data collection and metrics creation separate. In the first iteration of this service, the data was collected after there was an external request made to the exporter by Prometheus.

While the collecting and creating the metrics fit into the timeframe of Prometheus data collection, the author decided to create more future proof solution. This decision was made to ensure that this service can handle bigger Kubernetes clusters with more resources.

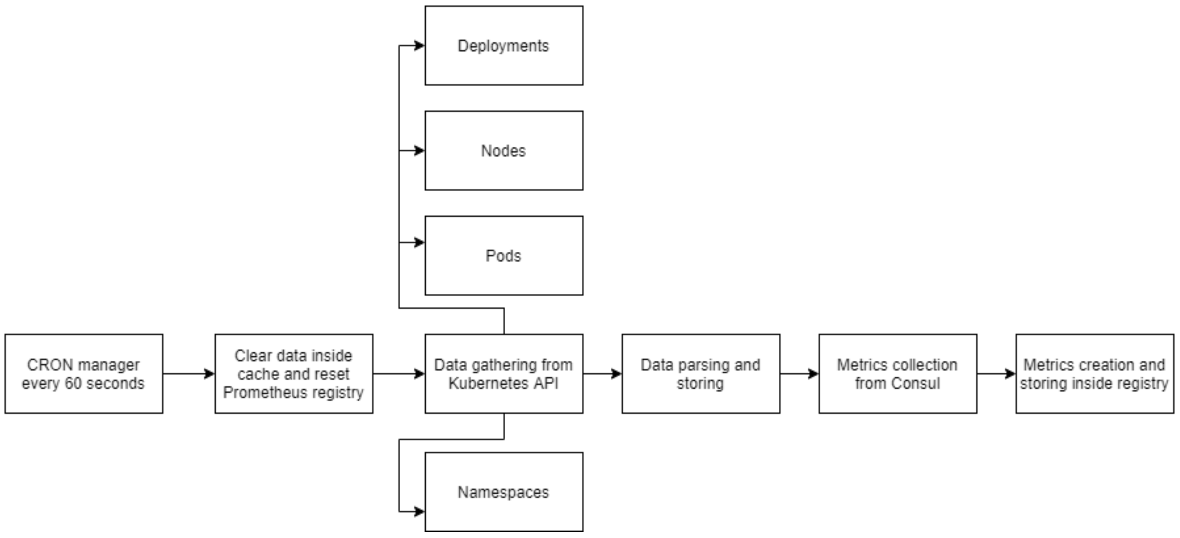

Deployments
Nodes
Pods
CRON manager every 60 seconds
Clear data inside cache and reset Prometheus registry
Data gathering from Kubernetes API
Data parsing and storing
Metrics collection from Consul
Metrics creation and storing inside registry
Namespaces

Figure 6. CRON job one lifecycle

The implementation has the same part from data gathering until the creation of metrics and storing them inside the registry that the initialization does. This process is running every sixty seconds, which guarantees near-real-time results.

5.3 Requests to exporter

As a result of creating a separate CRON job for data collection and metrics creation, one of the positive outcomes was the speed of requests that were done from the Prometheus side.

Due to the nature of kube-scrapers' architecture, the external requests to it are fast and do not need to keep the network connection alive until the full flow has been done. This is guaranteed by the fact that data collection and metrics creation is handled separately.

Every request made to the metrics endpoint allows the exporter to reply with the data it already has. In addition to the futureproofing, another idea on data collection was to decrease the time Prometheus spends gathering the data as it takes internal resources for the service to have an open connection and uphold a network connection until the full flow from data collection to exposing metrics is carried out. This issue was also remediated once the CRON job was implemented in the service.

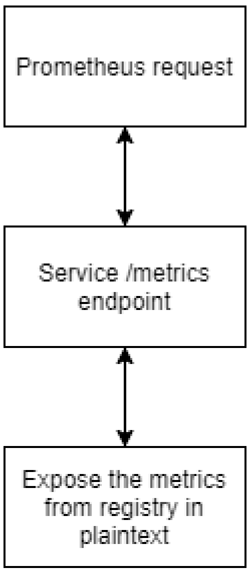

Prometheus request
Service /metrics endpoint
Expose the metrics from registry in plaintext

Figure 7. Request to exporter

When a request comes from outside to the exporter, the request is handled by the Express web server created during the start-up of service. It then calls out Prometheus client library to expose the metrics from the registry in plaintext and the contents of it are attached to the context of response.

5.4 Data gathering and storing

Data gathering and storing was created based on the “ask as much as possible and store as little as required” idea. This idea means that for each Kubernetes resource that is being requested from the API, there is a subset of request-response data that is being stored inside of the kube-scrapers in-memory storage. What it allows for is better usage and a smaller footprint of memory. If there is a need to add some part of the API response to the storage, then changes to the mappings must be made inside of the code. For the storage, the data is stored inside of the Node.js applications' constant variables. This means that for each Kubernetes resource object there is a variable that is initialized in the first cycle. This solution did not prove to have a significant impact on memory usage and thus no further data stores were considered.

Since the exporter requires authentication access to Kubernetes API, the best way was to supply an environment variable called **KUBECONFIG** to the *k8s* package. For development purposes, author decided to use administrator configuration, which has access to all the endpoints inside of the cluster. While this means, in the development environment there is access to Create, Read, Update, Delete (CRUD) operations via API, the risk was mitigated as the exporter does not have any update and delete calls towards the API and all the development was done using test environment Kubernetes clusters. In the production environment, kube-scraper does not have administrator configuration and has explicitly provided API endpoints with read-only rights to limit access.

Due to the nature of Kubernetes API, there was also a need for **conversion** before storing the data. The endpoints returned resource metrics in strings. **Random-access memory** (RAM) data was provided in different formats such as bytes, kilobytes, megabytes, and gigabyte as a unit of measurement [35]. The output for memory conversion is measured in megabytes as it allows for clear conversion, for example, if the API response had one gigabyte as memory usage, the conversion output would be 1024 megabytes, where only the numeric value would be saved into the storage.

For **central processing unit** (CPU), Kubernetes measures the resources in CPU units [35]. In the API responses millicpu, microcpu, nanocpu as units of measurement for CPU. Thus, the conversion was once again needed to get a baseline value that would be similar across the metrics. The output of the conversion is the number of CPU cores used for service. For example, if the service uses five hundred millicpu-s it is translated to 0.5 CPU units and like RAM data, it is stored as a numerical value.

As shown in Figure 6, the data collection consists of making requests to the Kubernetes API and getting the results back. The gathering is started by **getting the Pods information** from Kubernetes API. It is important to note that the requests are done in the following order: pod, node, deployments, namespaces. Pod data objects already include the containers that are inside the pod, which makes the mapping of data easier although causes issues when the data does not exist in the response. This issue is trivial as there is always one container inside the pod [8]. Every pod object that is saved into the storage consists of following: an **array of container objects**, **metadata** related to the pod such as namespace and owner team name, **resources** like limit and requests, **spec** that allows shows attached volumes to pod and node name the pod is on, and **status** that contains the statuses of container.

To get **data about nodes**, there were two different endpoints used. The first endpoint is the metrics-server node metrics endpoint. As was pointed out in the analysis, one of the open-source solutions was metrics-server (see more in section 3.3) that was considered as one of the plausible sources of data for this exporter. This endpoint allows to get the memory and CPU data of nodes. The second endpoint gave out the filesystem data that had to be mapped according to each node from the first request. As a combination after the two requests were merged into one, there was a need to create a mapping of the node data that would be stored inside the memory. Once again, the separate converter methods were useful as the resources did not follow a strict response as was later also confirmed when looking at the documentation of resources. The resulting array consists of node objects. Every object about a node consists of **metadata** such as name of the node and UID of node, **pods** that are located on this node and **resources** of the node such as available, limit, total, usage of memory and CPU.

Namespaces are directly queried from Kubernetes API and stored into the internal storage. The namespace name is stored in the metadata that is later used to show what namespaces are existing in a cluster. These metrics are useful for debugging the deployment of new Kubernetes clusters and making sure all required namespaces are existing.

Deployment data is collected from the Kubernetes API endpoint. For the deployments the saved fields are **metadata** similar to pod objects metadata, **app** label that indicates the service name, **status** of the deployment and **template**. From deployment metrics author also decided to create service objects. These objects contain combination of pods and deployments data and were hence decided to not query directly from the API again.

With the **cluster object**, author decided to not query the same data again since all the required metrics could be done using the previously gathered data. That meant a save in both requests to outside endpoints and time to complete these requests. All the cluster data is done using the aggregation of nodes, pods, and containers data to output the required metrics. One of the risks of aggregating cluster data is the requirement of all the previously mentioned data. This means that if the data about some of the components is missing, so is the cluster data. The decision was made to accept this risk and relieve it by making sure that even if some parts of the data are missing the in-memory data for them is returned as empty arrays instead of undefined values.

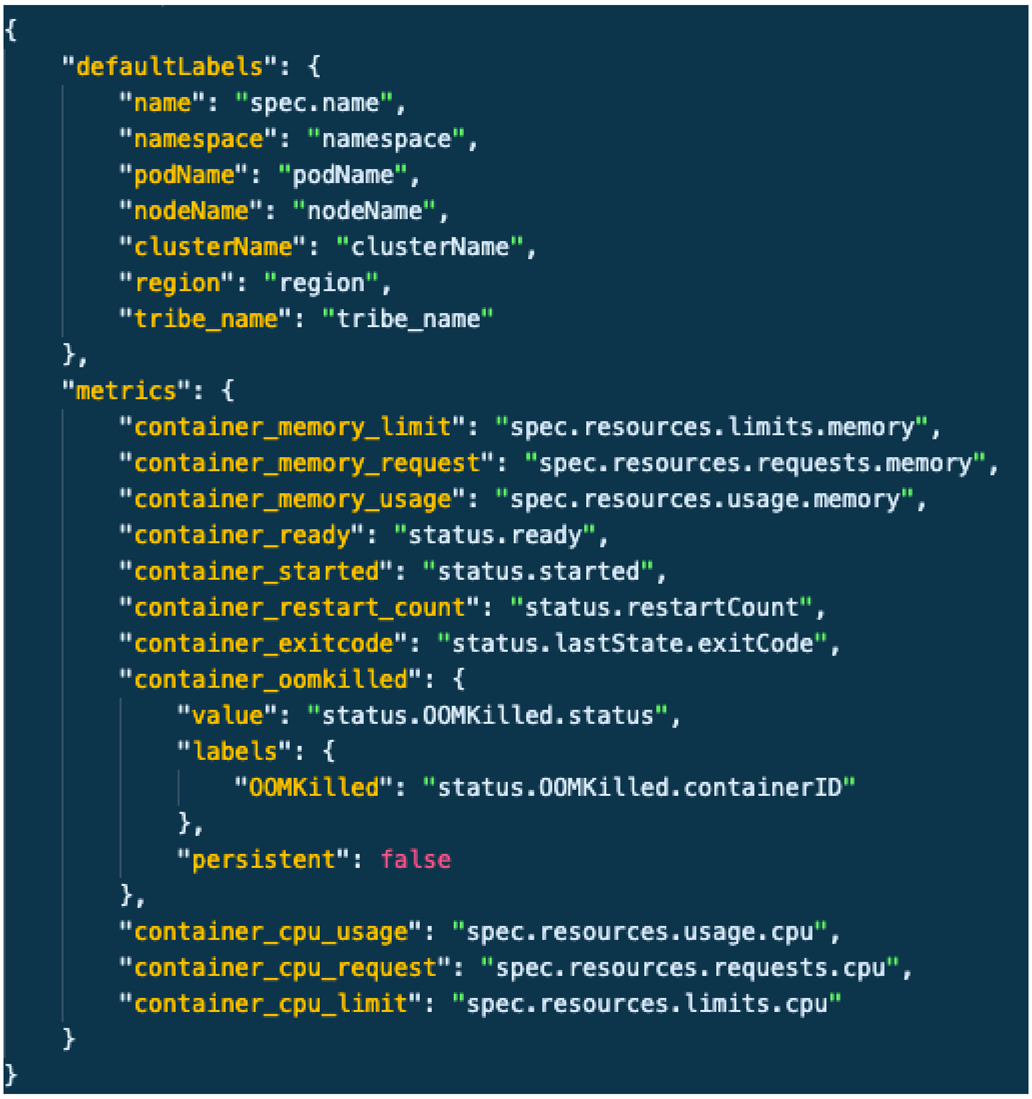
After metrics are stored inside of the in-memory storage, there is last step that involves adding cluster name, for example, `k8s-1`, to the metadata of containers, pods, and nodes. This allows the possibility to create a label `cluster_name`, which shows the cluster this pod is in.

5.5 Metrics configuration

During the planning phase, one of the pre-requisites was to create a modular service that would not need to be redeployed for a small metrics configuration. One of the solutions proposed was the external metrics configurations.

As a result, when the service is asking for metrics for the first time from Consul, the exporter creates a folder under one general configuration folder called “kube-scraper”. Inside this folder, there is a file called *available_metrics* and a folder called *metrics*.

The **file** called *available_metrics* holds a string of possible exported Kubernetes objects: pods, cluster, containers, nodes, deployments, namespaces, services. Each element stands for an object that data is being gathered for. This is built to allow for fast enablement or disablement of metrics if there is either an issue with the data collection or some of the metrics are not needed from a specific region. **Metrics** folder contains the following Kubernetes API objects: clusters, containers, deployments, nodes, services, and pod files. After some iterations on the structure of this configuration, the result came to be that each of this file holds a JSON object, where there are two main objects inside one big object.



```
{
    "defaultLabels": {
        "name": "spec.name",
        "namespace": "namespace",
        "podName": "podName",
        "nodeName": "nodeName",
        "clusterName": "clusterName",
        "region": "region",
        "tribe_name": "tribe_name"
    },
    "metrics": {
        "container_memory_limit": "spec.resources.limits.memory",
        "container_memory_request": "spec.resources.requests.memory",
        "container_memory_usage": "spec.resources.usage.memory",
        "container_ready": "status.ready",
        "container_started": "status.started",
        "container_restart_count": "status.restartCount",
        "container_exitcode": "status.lastState.exitCode",
        "container_oomkilled": {
            "value": "status.OOMKilled.status",
            "labels": {
                "OOMKilled": "status.OOMKilled.containerID"
            },
            "persistent": false
        },
        "container_cpu_usage": "spec.resources.usage.cpu",
        "container_cpu_request": "spec.resources.requests.cpu",
        "container_cpu_limit": "spec.resources.limits.cpu"
    }
}
```

Figure 8. Example of metrics configuration

The first object is named **defaultLabels**. This allows the exporter to get the labels that it must apply for each metric that is connected to this Kubernetes object inside the exporter. The key shows the label name, and the value is the location inside the object stored in the in-memory storage.

The second object is the **metrics** object. The key shows the name of the metric and the value shows the location inside the object stored in the in-memory storage. There is also a possibility to override or add custom labels to certain metrics. This can be achieved by adding the value as an object and supplying the value and labels as separate fields inside that object.

Additionally, there was a need to be able to add specific labels to certain metrics. To this need, the author decided to implement an **additional label possibility** by adding a label object inside the metric. As shown in Figure 8, the `container_oomkilled` differentiates from the other key/values by having an object instead of a string as a value. Inside that object, it is possible to add a label that also follows the same mapping logic as regular labels. It was also required to add persistent value as false to make a new metric instead of adding the label to an old metric.

While building the service, there was a decision made to create a **fail-safe** in case the Consul K/V store is unavailable. For that, there are basic metrics configuration stored with the same structure as in Consul and the exporter can fall back to default metrics inside of the code if it does not get the response from an external source. Although this goes against the external metrics configuration idea, it was decided to keep it to make sure the service does not have heavy relations towards outer services and can survive the loss of external data source.

5.6 Metrics creation and exposing

The exporter uses Express to create a web server that can serve the metrics as plain text. By default, Prometheus organization recommends exposing the metrics from the `/metrics` endpoint. With Kubernetes exporter, the decision was made to use **Gauge** [25] metrics, since most of the monitored values are arbitrarily increasing and decreasing in time.

The **creation of metrics** starts by making an external request to Consul. As the first step all the enabled resources from the previously mentioned files are fetched. Then another request is done to get the JSON configurations from the folder. If the requests to get the configuration fails, then the service defaults to the created fail-safe that was previously mentioned.

Since metrics are grouped similar to the way data is cached inside the exporter, the creation of metrics is considered trivial timewise and can be excluded as a time-consuming event inside the exporter. Metrics are generated with a specific naming convention using the prom-client library as show in Figure 9.

```
<prefix><kubernetes_object><metric_name>{<label_name>=<label_value>} <metric_
value>
```

Figure 9. Naming convention of exposed metrics

The prefix of every metric is kube_scraper, which indicates the name of the service outputting this metric. **Kubernetes object** is related to the name of the objects that are gathered for instance cluster, node, pod, container, service. **Metric name** is created using the best practices [26] provided by Prometheus and following a logical structure to ease the use of them later in the visualization. **Label names and values** are mapped based on the configuration taken from the Consul. **The metric value** is created by getting the value of a metric from the stored object from a specific location that is provided as a key in the configuration.

After the metrics have been saved to a registry provided by the Prometheus Node.js library, the results can be exposed through the metrics endpoint.

5.7 Deployment of the exporter

The deployment of this exporter is done using Chef [36]. Chef provides the ability to provision the exporter to each Kubernetes cluster via YAML configuration. To deploy a newer version of the exporter two things need to be changed. Firstly, a new image with the changes must be built. In Pipedrive the repository for Docker images is Docker Hub [37]. Image is built locally using Docker commands and then pushed to the repository. After that, the version environment variable inside the manifest of the exporter must be updated. Once the variable reaches master, it triggers a change in the manifest and signals Kubernetes to configure the deployment. During the deployment, a new image with the tag `latest` is pulled from the repository.

To guarantee the quality of each deployment the changes are tested thoroughly in the test environment for at least a week to ensure the stability of new code and spot any critical issues that might not show up instantly.

6 Implementation in Production Environment

Kubernetes exporter has been in the production environment since February 2020 with the latest deployment in August 2020.

Pipedrive has two datacentres located in Germany and the United States of America. Each of these regions has 3 Kubernetes clusters with fifty-four nodes. On those fifty-four nodes, 2667 pods are running. These clusters hold all the live microservices that makeup Pipedrive.

Every cluster has **one kube-scraper** that collects data about the objects from that cluster. This means in total there are 3 scrapers per one datacentre that expose the metrics. Metrics are then scraped from the Prometheus that is in that datacentre and then shipped off to a third region where data aggregation is done for visualization and alerting.

The developed service gives out a total of 19818 metrics per scrape. Data will be gathered every 30 seconds and one hour of metrics would add up to 562 MB or 0.562 GB. For one day of data collection, the needed storage accumulates to 13.536 GB and the 30-days retention 406.08 GB of storage.

The solution requires nearly two times more storage than cAdvisor for thirty days of data retention. On the other hand, it must be considered that cAdvisor metrics contain only information about containers whereas the metrics exposed by Kubernetes exporter give out data about containers, pods, deployments, services, nodes, and cluster.

There are 57 unique metrics exported by developed Kubernetes exporter.

Table 4. Total amount of developed exporter metrics

Type	**Unique metrics**	**Total lines of unique metrics**
Containers	12	13399
Pods	7	4563

Type	Unique metrics	Total lines of unique metrics
Nodes	16	223
Cluster	19	18
Service	3	1440

One of the **positive outcomes** of the exporter was the separate configuration. This has proved to be a helpful tool for multiple cases where there was a request for new metrics. Regarding the reliability of the Consul, that stores the external configuration, there have been no observed issues with getting the metrics configuration from that data source.

Kubernetes exporter allowed four different departments in Pipedrive's engineering to gain visibility about the resources in the Kubernetes cluster.

Developers are interested in these metrics because they can see and monitor their services in the production environment. These metrics have also allowed them to see if the changes made to the services' codebase caused elevated or decreased resource usage.

For the **DevOps Tooling** tribe, these metrics give an insight into the deployments and the scheduling of services to nodes. They can see if the spreading of pods works across all nodes inside the Kubernetes cluster, and react to the potential issues should they arise. They can also suggest optimal resource allocation to developers for their services.

For the **Infrastructure** team, the main usage of these metrics is the resource utilization of nodes. The relevant metrics for them are memory and CPU resource usage and limits – these can cause the overfilling or failures of nodes.

The **Site Reliability Engineering** team has found the Kubernetes exporter to be beneficial by providing insight into the overall metrics. Also, it has allowed for a smoother transition of platforms during the implementation of Kubernetes.

6.1 Feedback from the engineering teams

The following section introduces feedback gathered from engineers across multiple teams in Pipedrive. Questions regarding feedback are presented in the Appendix 3.

“Kube-scraper helped to filter out Kubernetes internal metrics and only provide those, which are useful for us. In addition, it helped to ease the migration from Docker Swarm to Kubernetes and minimised the necessary changes to our related monitoring,” said George Javakhidze, Lead Site Reliability Engineer in Pipedrive.

“Kubernetes resource exporter has allowed DevOps Tooling to do data-driven decisions regarding Kubernetes. Besides visualization, we also use the metrics to alert and guarantee stable environment within the Kubernetes cluster.”

“As an example of usage of metrics, I have monitored Kubernetes resource usage reservations by services, to make sure services reserve optimal number of resources. This ensures that clusters are well utilized and not overprovisioned,” brought out Pavel Baranov, Senior DevOps Engineer in Pipedrive. “As for the additional data, I would like to have more Kubernetes objects such as replicasets or daemonsets, which would allow to create even more complex dashboards and correlate data in a more meaningful way. In terms of labels, it is important that child objects have information about their parents such as container metrics have labels about pod it is in which current metrics have.”

“With kube-scraper we can easily configure metrics and their labels. This allows us to quickly modify Prometheus metrics without any kind of redeploy or restart. As an example, we wanted the possibility to filter all services by their team or tribe. With kube-scraper it was very easy to just add an additional metrics,” told Tambet Paljasma, Senior Infrastructure Engineer. Regarding more features Tambet suggested to also have IP addresses of pods. This would help Infrastructure engineers to debug some network or service discovery related issues more easily.

“Kube-scraper has had impact in many different ways throughout the company itself – as it concentrates the information provided by Kubernetes itself in a more granular method, it helps the engineers working closely with Kubernetes recover their systems in a more timely manner,” told Kristjan Hiis, Monitoring team lead.

Regarding the reliability of implemented service none of the engineers had complaints about it.

To summarize the feedback from engineers, current implementation has enabled them to alert and visualize the data in a meaningful way. There were some improvements that engineers suggested, and these have been taken as future improvements of this thesis.

7 Future outcomes

During the writing of this thesis, the solution the author implemented was not released to the public. The author has plans to open-source the kube-scraper after adding more metrics configuration stores. This open sourcing may lead to more metrics being added to the exporter that this thesis has not covered. The tool is widely used inside the company that it was created for but needs more testing to be released as an open-source solution. One of the proposed data sources would be a Kubernetes ConfigMap. This could contain the metrics that can be also changed on the fly without causing disruptions to the service. To achieve more data sources, the author has plans to add storage handlers that can be chosen during initialization.

From the feedback of engineers, it was also suggested to add replicasets and daemonsets metrics as well as data about Container Network Interface. This would allow to increase the observability of the Kubernetes cluster and allow for more in-depth monitoring of it.

Another possible addition in the future would be to create tests that would mock the Kubernetes API responses. That would allow to test functions that would rely on input from Kubernetes API and enhance the reliability and stability of this service.

8 Summary

The goal for this thesis was to implement a Kubernetes resource monitoring solution in Pipedrive. The first trial was done using Zabbix monitoring system and then it was concluded that this system was not sufficient. After that Prometheus as a datastore was considered and opted for.

After finding a platform where to store the data, analysis was conducted by the author of this theses to find currently existing open-source monitoring solutions for Kubernetes clusters that would fit the criteria. The analysis concluded that there were no suitable solutions found. After that author designed and created the solution for Kubernetes resource monitoring.

As a result, a custom Prometheus exporter, which has an external configuration for its metrics, was created. It allows to add metrics and labels for the metrics without service disruption. Thus, the exporter was built with scalability in mind.

The previously mentioned solution runs in every Kubernetes cluster in Pipedrive. It supplies the required data to let the engineers see resource usage over time and know if something goes wrong with their services.

The implementation received positive feedback from engineers from different teams inside of Pipedrive. Thus, the solution built within the scope of this thesis has successfully solved the given problem.

References

[1] “CNCF_Survey_Report.pdf,” [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf . [Accessed 02 12 2020].

[2] “11 Facts About Real-World Container Usage | Datadog,” [Online]. Available: https://www.datadoghq.com/container-report/ . [Accessed 02 12 2020].

[3] “Deploy to Swarm | Docker Documentation,” [Online]. Available: https://docs.docker.com/get-started/swarm-deploy/. [Accessed 30 12 2020].

[4] J. Turnbull, in *Monitoring with Prometheus,* Turnbull Press, 2018, pp. 6-9.

[5] “What is Kubernetes? | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 29 11 2020].

[6] “community/icons at master · kubernetes/community,” [Online]. Available: https://github.com/kubernetes/community/tree/master/icons. [Accessed 30 12 2020].

[7] “Kubernetes Components,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 10 10 2020].

[8] “Pods | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/. [Accessed 10 10 2020].

[9] “What is a Container? | App Dockerization | Docker,” [Online]. Available: https://www.docker.com/resources/what-container. [Accessed 11 10 2020].

[10] W. a. K. M. Deployments. [Online]. Available: https://devspace.cloud/docs/cli/deployment/kubernetes-manifests/what-are-manifests#:~:text=Kubernetes%20manifests%20are%20used%20to,apply%20%2Df%20my%2Dfile.. [Accessed 19 10 2020].

[11] “kubelet | Kubernetes,” [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/#synopsis. [Accessed 19 10 2020].

[12] “What Is Etcd and How Do You Set Up an Etcd Kubernetes Cluster?,” [Online]. Available: https://rancher.com/blog/2019/2019-01-29-what-is-etcd/. [Accessed 19 10 2020].

[13] [Online]. Available: https://www.zabbix.com/. [Accessed 19 10 2020].

[14] Z. Agent. [Online]. Available: https://www.zabbix.com/zabbix_agent. [Accessed 20 10 2020].

[15] “Overview | Prometheus,” [Online]. Available: https://prometheus.io/docs/introduction/overview/. [Accessed 22 10 2020].

[16] “Prometheus: Monitoring at SoundCloud | SoundCloud Backstage Blog,” [Online]. Available: https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud. [Accessed 12 12 2020].

[17] “Prometheus 1.0 is here | Cloud Native Computing Foundation,” [Online]. Available: https://www.cncf.io/blog/2016/07/18/prometheus-1-0-is-here/. [Accessed 12 12 2020].

[18] "Github - prometheus/alertmanager: Prometheus Alertmanager," [Online]. Available: https://github.com/prometheus/alertmanager.

[19] "Github - promtheus/pushgateway: Push acceptor for ephemeral and batch jobs.," [Online]. Available: https://github.com/prometheus/pushgateway. [Accessed 15 10 2020].

[20] "Time series - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Time_series. [Accessed 04 01 2020].

[21] "Data model | Prometheus," [Online]. Available: https://prometheus.io/docs/concepts/data_model/. [Accessed 22 10 2020].

[22] "Overview | Prometheus," [Online]. Available: https://prometheus.io/docs/introduction/overview/#when-does-it-fit. [Accessed 01 01 2020].

[23] "Query functions | Prometheus," [Online]. Available: https://prometheus.io/docs/prometheus/latest/querying/functions/. [Accessed 03 01 2020].

[24] "Prometheus Metrics, Implementing your Application | Sysdig," [Online]. Available: https://sysdig.com/blog/prometheus-metrics/. [Accessed 22 12 2020].

[25] "Metric types | Prometheus," [Online]. Available: https://prometheus.io/docs/concepts/metric_types/#metric-types. [Accessed 27 11 2020].

[26] "Metric and label naming | Prometheus," [Online]. Available: https://prometheus.io/docs/practices/naming. [Accessed 15 10 2020].

[27] "Client libraries | Prometheus," [Online]. Available: https://prometheus.io/docs/instrumenting/clientlibs/. [Accessed 19 10 2020].

[28] "Metrics For Kubernetes System Components | Kubernetes," [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/system-metrics/. [Accessed 05 01 2020].

[29] "Tools for Monitoring Resources | Kubernetes," [Online]. Available: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/. [Accessed 05 01 2021].

[30] "GitHub - google/cadvisor: Analyzes resource usage and performance characteristics of running containers.," [Online]. Available: https://github.com/google/cadvisor. [Accessed 18 10 2020].

[31] "GitHub - kubernetes/kube-state-metrics: Add-on agent to generate and expose cluster-level metrics.," [Online]. Available: https://github.com/kubernetes/kube-state-metrics. [Accessed 20 10 2020].

[32] "GitHub - kubernetes-sigs/metrics-server: Cluster-wide aggregator of resource usage data.," [Online]. Available: https://github.com/kubernetes-sigs/metrics-server. [Accessed 20 10 2020].

[33] "Express - Node.js web application framework," [Online]. Available: https://expressjs.com/. [Accessed 3 11 2020].

[34] "Intro to Consul | Consul by HashiCorp," [Online]. Available: https://www.consul.io/docs/intro. [Accessed 29 12 2020].

[35] "Managing resources for Containers | Kubernetes," [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-memory. [Accessed 30 10 2020].

[36] “Chef Infrastructure Management | Infrastructure Management Automation | Chef,” [Online]. Available: https://www.chef.io/products/chef-infrastructure-management. [Accessed 30 12 2020].

[37] “Docker Hub,” [Online]. Available: https://hub.docker.com/. [Accessed 30 12 2020].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Proposed metrics and labels

Container

Labels: Container name, Namespace, Pod name, Node name, Container owner (Tribe name), Cluster name, Region, Environment, Image of the container

Metrics:

- Container memory
  - Usage
  - Limit
  - Requested
- Container CPU
  - Usage
  - Limit
  - Requested
- Container Network (optional)
- Container I/O traffic (optional)
- Container storage
  - Usage
  - Limit
  - Requested
- Count for container restarts
- Container exit codes

Pod

Labels: Pod name, Pod ID, Namespace, Node name, Pod owner (Tribe name), Cluster name, Region, Environment, Service name

Metrics:

- Pod memory usage/limit/requested
- Pod CPU usage/limit/requested
- Pod health checks - readiness and liveness probes with status

- Pod I/O traffic (optional)
- Pod Network (optional)
- Pod Out of Memory
- Pod disruption budget (optional)
- Pod Storage
- Count for Pod restarts

Node

Labels: Node name, Node UID, Environment, Cluster name, Region

Metrics:

- Node memory
  - Available
  - Allocatable
  - Limits
  - Requested
  - Total
  - Unrequested
  - Usage

- Node CPU
  - Available
  - Allocatable
  - Limits
  - Requested
  - Total
  - Unrequested
  - Usage
- Total count of allocatable pods to node
- Total count of running pods on node

Service

Labels: Service name, Namespace, Cluster name, Region, Service owner (Tribe name), Service Repository name (Optional), Environment, Docker tag (Optional), Docker image (Optional), Instances, Job name – name of deployment (Optional)

Metrics:

- Service replication
- Service available replicas

Cluster

Labels: Domain, Cluster name, Region, Environment

Metrics:

- Cluster memory
    - Available
    - Allocatable
    - Free
    - Limits
    - Requested
    - Total
    - Unrequested
    - Usage

- Cluster CPU
    - Available
    - Allocatable
    - Free
    - Limits
    - Requested
    - Total
    - Unrequested
    - Usage
- Total count of pods in the cluster
- Total count of nodes in the cluster
- Total count of containers in the cluster

Appendix 3 – Feedback questions to engineering teams

In total four questions were given to engineers to gather feedback related to the implementation of the exporter.

1. How has kube-scraper enabled you to deal with the tasks related to Kubernetes? If possible, bring out few examples.

2. Is there some data you would like to get from Kubernetes cluster via kube-scraper that doesn't currently exist?

3. Have you had any issues with the kube-scraper during your usage?

4. Regarding the metrics, are there any labels that you would like to add to metrics or change the format of existing ones?