



Waterford Institute *of* Technology

Kubernetes Observability

Dimitri Saridakis

Work submitted
within the
BSc in Applied Computing
Cloud & Networks

Lecturer: Richard Frisby

April 5, 2022

Contents

List of Figures	i
Listings	iii
1 Kubernetes Observability	1
1.1 Introduction	1
1.1.1 Motivation	1
1.1.2 Implementation Overview	2
1.2 Technologies	4
1.2.1 Prometheus	4
1.2.2 Grafana	8
1.2.3 Alertmanager	9
1.3 Deployment	10
1.3.1 Prometheus Operator Choice	10
1.3.2 Monitoring Kubernetes Node	11
1.3.3 Monitoring Strimzi and Kafka	12
1.3.4 Monitoring Flask Server	22
1.3.5 Additional PostGres Database Exporter for Prometheus	24
1.3.6 Prometheus Self Monitoring	25
1.4 Conclusion	25
Bibliography	27

List of Figures

1.1	Architecture of the proposed monitoring system.	3
1.2	Architecture of Strimzi Operator from [30].	3
1.3	Architecture of the observability stack, image from[25]	4
1.4	Job definition for Prometheus to scrape metrics from custom application. Each component to be monitored gets its own job definition.	4
1.5	Prometheus graph data query. From local cluster	5
1.6	Prometheus tabular data query. From local cluster	5
1.7	Prometheus targets in the GUI.	7
1.8	Prometheus recording rules yaml.	8
1.9	Prometheus recording rules in the Prom GUI	8
1.10	Grafana dashboard layout. From local cluster	9
1.11	Alertmanager GUI depicting some alerts	10
1.12	Helm chart for Prometheus Operator.	10
1.13	Node Exporter Service.	11
1.14	Node Exporter successfully registered as a healthy target in the Prometheus GUI.	11
1.15	Node Exporter dashboard.	12
1.16	Scrape Config Secret.	13
1.17	Strimzi PodMonitor	14
1.18	ConfigMap for Kafka broker for Prometheus integration.	16
1.19	Prometheus YAML for Strimzi components.	17
1.20	Prometheus successfully defining Strimzi Operator as a target.	17
1.21	Prometheus successfully defining both the Topic Operator and the User Operator as part of the Strimzi Entity Operator as a target.	18
1.22	Prometheus successfully defining the Kafka components as targets.	18
1.23	Prometheus successfully querying the Kafka components via the UI.	19
1.24	Grafana dashboard for the Strimzi Operator monitoring.	19
1.25	Grafana dashboard for the Kafka broker.	20
1.26	Grafana dashboard for the Zookeeper deployment.	20
1.27	Grafana dashboard for the Kafka exporter dashboard I/II.	21
1.28	Grafana dashboard for the Kafka exporter dashboard II/II.	21
1.29	Prometheus successfully registering the Flask App as a target.	24
1.30	Grafana dashboard for the Flask application.	24
1.31	Target successful for Prometheus Postgres Exporter deployment.	25
1.32	Prometheus self monitoring.	25

Listings

1.1	Kafka resource with metrics configuration applied.	14
1.2	Zookeeper resource with metrics configuration applied.	15
1.3	Kafka resource with Kafka Exporter configuration applied.	15
1.4	Flask App for Prometheus monitoring.	22
1.5	Deployment Resources for the Flask App	23
1.6	Code for the Flask Service.	23
1.7	Prometheus job for the Flask App.	24

1 Kubernetes Observability

1.1 Introduction

The sharp rise in the popularity of the microservices architecture [20], coupled with the rapid adoption of Kubernetes (k8s) as the industry-leading technology to deploy and orchestrate the services, workloads in K8s clusters have grown in both size and complexity as more businesses adopt this architecture.

With a larger number of services running in a cluster, the availability of the system in its entirety depends on the availability of the individual services which constitute the system. This is where the complexity explodes. As more and more microservices are brought online, there needs to be a way to for DevOps / Site Reliability Engineers (SREs) to monitor that each microservice is running and performing as intended.

Furthermore, monitoring resources and components of the system can be used to identify small issues before they grow into larger issues. The observability stack can be seen as a major component of both disaster recovery and disaster prevention.

Monitoring is achieved by gathering metrics. Metrics can be defined as a numerical representation of some measurement of the system. Metrics can be collected from the system and then used to determine the health of the system. Metrics usually come in a *time series* format i.e. the change of values over time. They can also be seen as the derivative, or the rate of change, of the measurement over time.

Different applications can have different types and forms of metrics. For example, a web server may have metrics like the number of requests, the number of errors, the number of bytes sent, etc. Whilst a database may have metrics like the number of queries, the number of active connections, write performance - to name but a few.

As an example, if a web server is slow at delivering a response, it may be useful to know what is going on with the server. The amount of memory, CPU and disk space that is being used by the server can be useful in identifying the cause of the slow response. This can be found out by measuring the system resources of the underlying server and coupled with other pieces of information about, say - the amount of HTTP requests that are being received, knowledge can be gleaned and action can be taken to improve load balancing or scale the amount of servers to handle the load.

This paper assumes the reader has basic knowledge of both Apache Kafka and Kubernetes, although more complex components of both will be elaborated on.

1.1.1 Motivation

The motivation for this research paper is to:

1. Explore and implement the most prominent methods of recording the behaviours of the system's components aka the *observability stack*.
2. To concentrate on open-source technologies, this happens to be straightforward as the most popular monitoring tools in industry are all open-source, although many have *enterprise* (paid) editions / options. There are other paid-for tools in this category, but they are nowhere near as popular.
3. To automate as much of the processes and setup as possible.

This will be achieved by focusing on the following:

1. Enabling monitoring of preexisting services in a cluster i.e. components that were configured without monitoring capabilities during initial configuration. This should prove the most challenging but also the most rewarding from a knowledge acquisition perspective.
2. Deployment of an application (server) from scratch with monitoring capabilities enabled. Serving up custom metrics.
3. Enabling the monitoring of the system, CPU, memory etc. of machine upon which Minikube is deployed.
4. Configuration of the monitoring / observability stack in the cluster.

For this paper the observability stack is the term that is used to denote the following components, all running in a Minikube (single node Kubernetes cluster for local development) cluster:

- Prometheus[26] is an open-source systems monitoring tool.
- Grafana[10] is an open-source tool for visualizing and analyzing the data collected by Prometheus.
- Alertmanager[22] is an open-source tool for sending alerts to relevant teams about the system performance.
- Exporters to allow components to be monitored by exporting metrics in a format that Prometheus can understand and consume. They act as a bridge / conversion layer between the components and Prometheus.

1.1.2 Implementation Overview

The implementation of this paper will be carried out in the following steps:

1. Install the observability stack on the cluster, keeping automation in focus, this will be accomplished through the use of operators and Helm.
2. Enable a preexisting service to be monitored, in this case it is a somewhat verbose example. The technology chosen is a functioning Apache Kafka cluster which is deployed via the Strimzi Operator, Figure 1.2, consists of many components:
 - **Kafka broker:** stores data in a distributed fashion in the form of topics and is the major component.
 - **Zookeeper:** is a highly available service that provides a distributed lock mechanism for the Kafka brokers and undertakes leader elections, amongst other responsibilities.
 - **Strimzi Operator:** is a Kubernetes Operator that manages the Kafka cluster, itself deployed by Helm.
 - **Kafka Exporter:** exports information about the topics and partitions in on the Kafka brokers.

The use of automation here may prove to make the process of retrospectively monitoring the cluster more difficult. Each service has numerous configuration files, (CRDs, ConfigMaps, Deployments, Services, ClusterRoles etc.) that need to be located and edited with the correct configurations manually. Since operators are used to automate the process, the initial configuration is not done by the user therefore the user has limited knowledge on the exact configuration files needed for the correct provisioning of the service.

3. Create a Flask application, enable monitoring with some custom metrics and deploy it to the cluster. The *Python- Prometheus-Flask-Exporter* library[29] is used here to allow for coherent communication between Prometheus and the Flask server.

4. Gather metrics about the underlying machine's resources i.e. the Kube node(both control and data plane as they are on the same machine). This is done via the *Node Exporter*, which allows for the communication between Kube API server and Prometheus.
5. For all of these applications the following will be implemented:
 - a) View metrics in Prometheus UI.
 - b) Create / import dashboards in Grafana UI to visualize the metrics.

Some alerts will be created to see Alertmanager in action.

The following, Figure 1.1, is an architectural diagram of the proposed system:

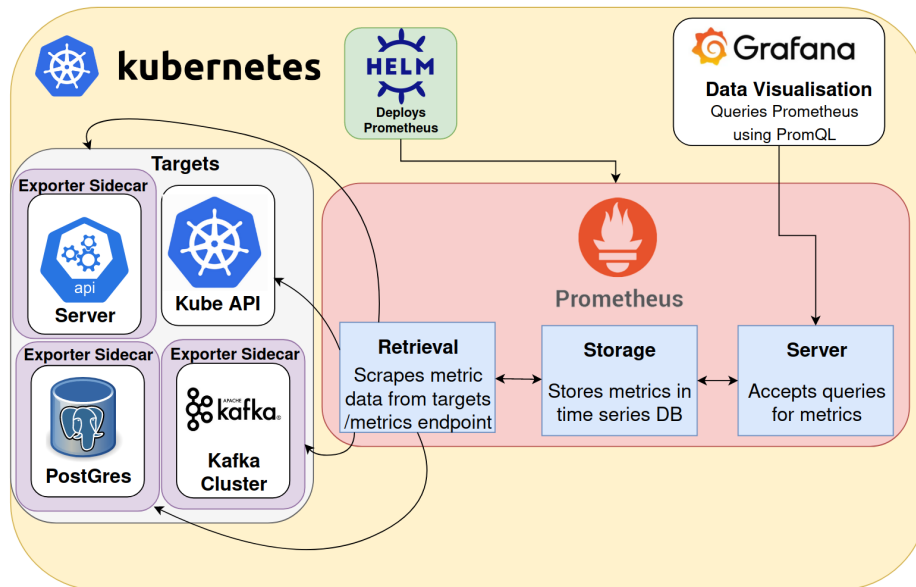


Figure 1.1: Architecture of the proposed monitoring system.

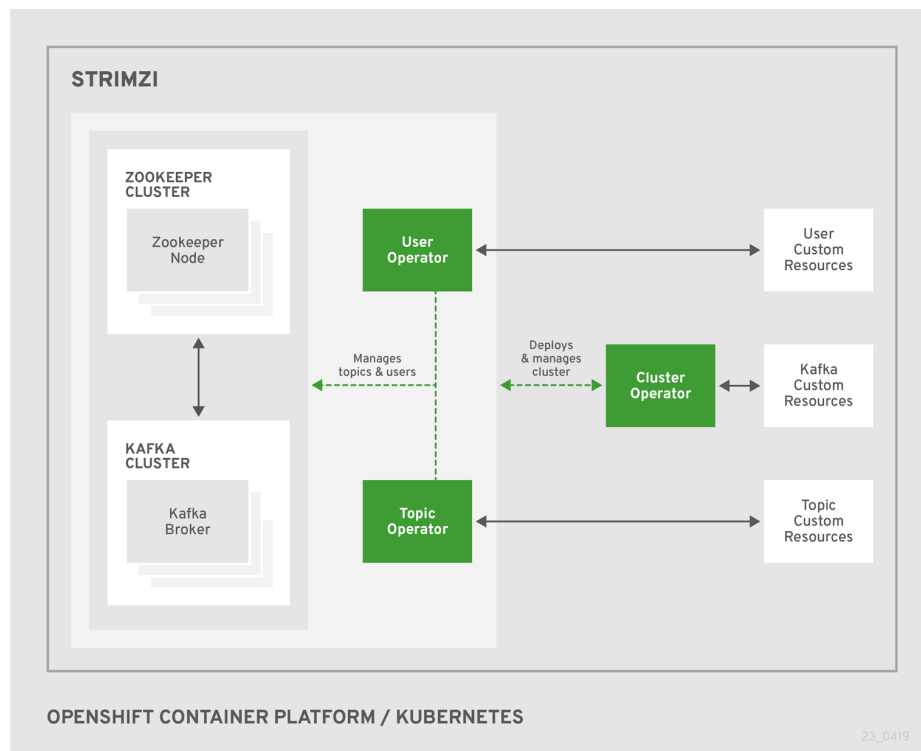


Figure 1.2: Architecture of Strimzi Operator from [30].

1.2 Technologies

This section gives a more in-depth look at the technologies used in this research paper. The following is the architecture diagram of the observability stack Figure 1.3:

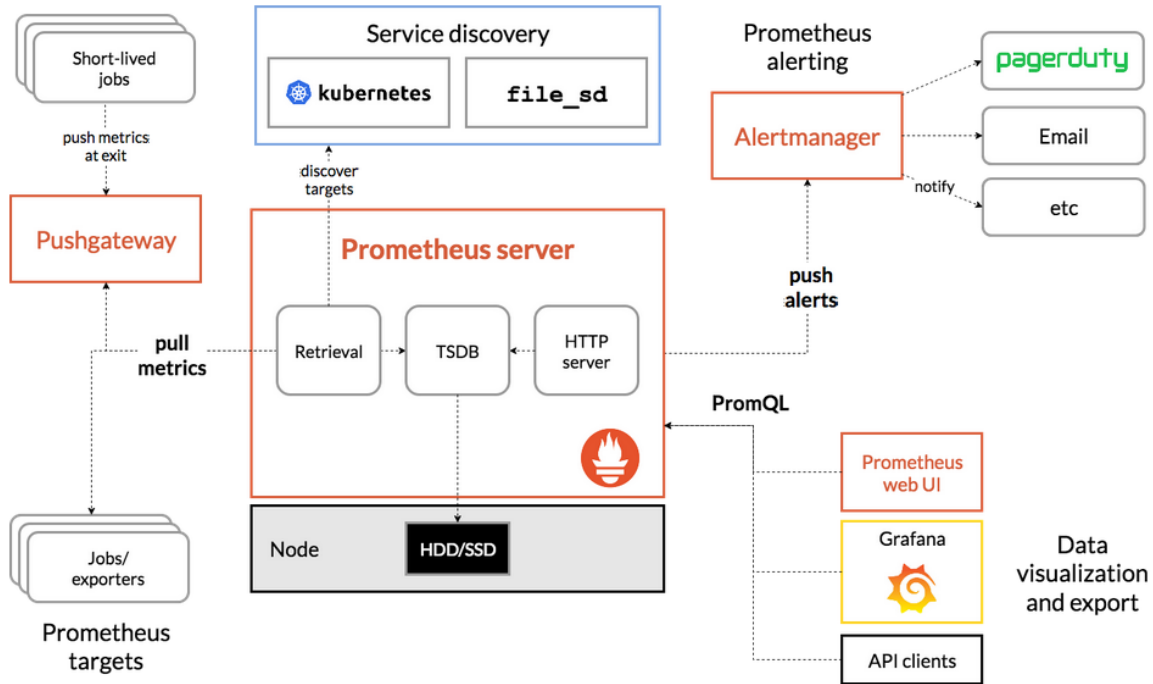


Figure 1.3: Architecture of the observability stack, image from[25]

1.2.1 Prometheus

Prometheus is managed by the Cloud Native Computing Foundation (CNCF)[3] and is the second graduated project in the of the CNCF foundation (after Kubernetes). Prometheus is mostly written in GoLang. The following components are used in a Prometheus stack:

- **Prometheus Server:** The main component of the stack. It is responsible for collecting metrics, which it does by periodically scraping the components that are being monitored, by default on an `/metrics` endpoint. It then stores this data in its own time series database. Prometheus knows which components to fetch metrics from by the *jobs* that are defined in the, conventionally named, `prometheus.yaml` file. A job definition looks like the following Figure 1.4:

```

- job_name: fin-db-server
  scrape_interval: 10s
  scrape_timeout: 10s
  static_configs:
    - targets: ["fin-db-server:5005"]

```

Figure 1.4: Job definition for Prometheus to scrape metrics from custom application. Each component to be monitored gets its own job definition.

The server discovers the components to monitor via interaction with the Kubernetes Service Discovery mechanism, as per Figure 1.3.

Prom Server is also responsible for pushing alerts to Alertmanager (This is covered in greater detail later).

- **Client Libraries:** Before services can be monitored, instrumentation needs to be added to the components code, which implement the Prometheus *metric types*. There are many official libraries available for GoLang, Python, Rust etc. along with numerous third-party libraries for other languages[23].
- **Exporters:** To import metrics from a third-party component i.e. Apache Kafka via the *JKX exporter* (which is utilized in this paper), database exporters, hardware, server and other exporters are also available. A comprehensive list is available here[24].
- **Pushgateway:** The Pushgateway is a service that allows for the pushing of metrics to Prometheus from ephemeral or batch jobs. As these kinds of jobs may not live long enough to be monitored, the Pushgateway is used to push metrics to Prometheus. This is a feature that is not used in this paper.

The metrics data stored in Prometheus is stored in a time series database, which is a key-value store. The metrics are queried using Prometheus Query language (PromQL). PromQL allows users to select and aggregate the data in real time. The query results can be viewed as a graph, Figure 1.5, or as tabular data as per Figure 1.6 in the Prometheus GUI. Queries can also be consumed by external tools or systems via the HTTP API, this is how Grafana gets data from Prometheus.

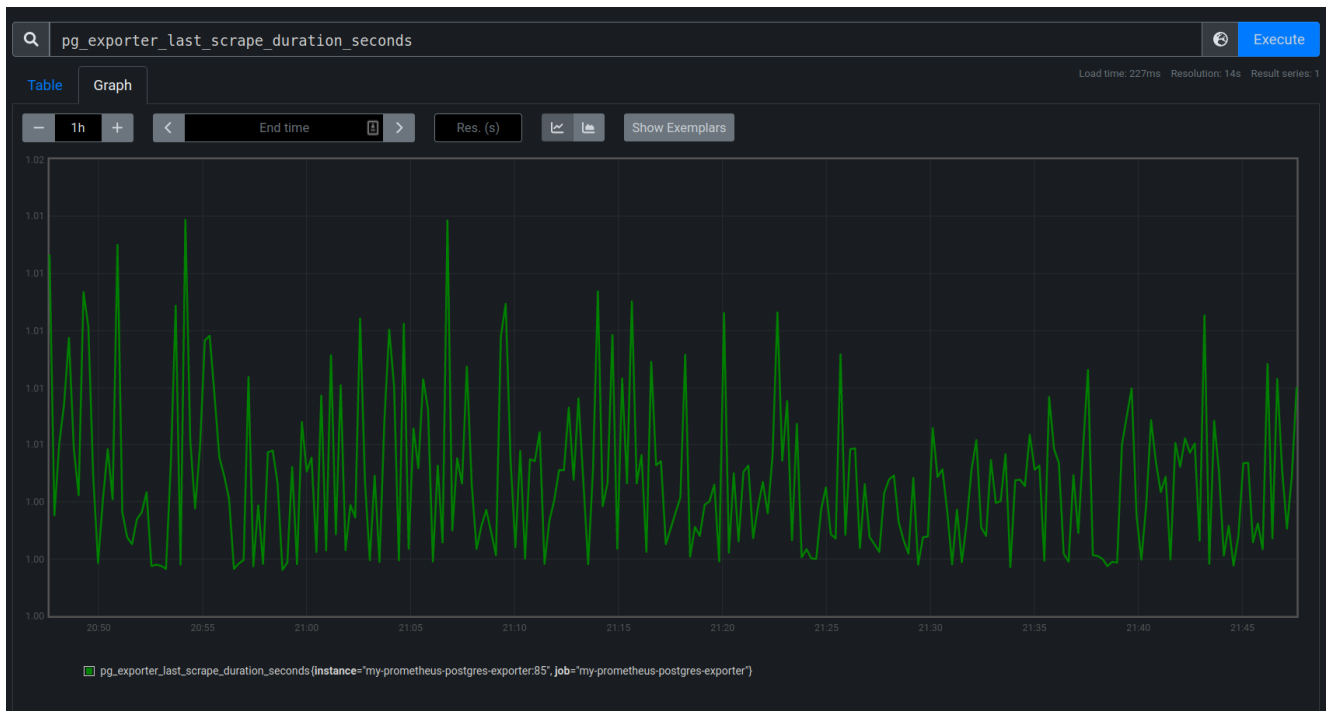


Figure 1.5: Prometheus graph data query. From local cluster

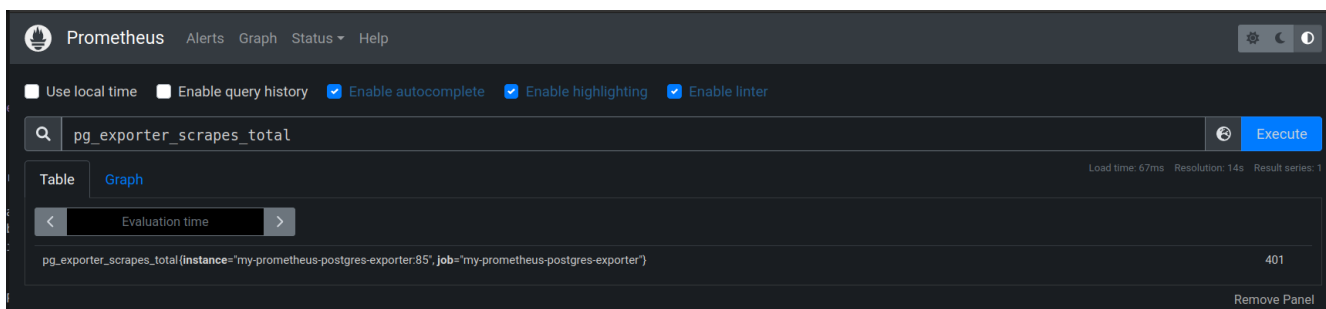


Figure 1.6: Prometheus tabular data query. From local cluster

PromQL is a functional and quite a powerful language. There are some basic data types, as per [28]:

- **Instant Vector:** are a set of time series containing a single sample for each time series, all sharing the same timestamp
- **Range vector:** a set of time series containing a range of data points over time for each time series
- **Scalar:** a simple numeric floating point value
- **String:** a simple string value.

The language also has operators and functions, which are used to manipulate the data [27]. Queries can become quite complex and whilst some time was spent with the language, not enough time was available to become truly proficient with it.

Metrics fall into one of four categories which each have their own use case.

- **Gauge:** A gauge is a metric that represents a single numerical value that can increase or decrease, i.e. memory usage, number of running processes etc.
- **Counter:** A counter is a metric that represents a single numerical value that can only increase, such as the number of requests served etc.
- **Histogram:** A histogram is a metric that represents a distribution of numerical values. The histogram metric type measures the frequency of value observations that fall into specific predefined buckets[11]. Can be used to measure response size or request duration.
- **Summary:** A summary is a metric that represents a distribution of numerical values. This type preceded histograms and the main difference is that histogram quantiles are calculated on the Prometheus server whilst summary quantiles are calculated on the application side. This means that summaries cannot be calculated on multiple applications. There are some use cases for which summaries are useful, but by and large, histograms are now used much more.

Prometheus has its own GUI in which one can query and view metrics data in various ways. One can view targets and statuses The following figure shows the registered targets in the Prometheus UI and their status (Note: this is a highly experimental cluster, and as such there are a few targets down):

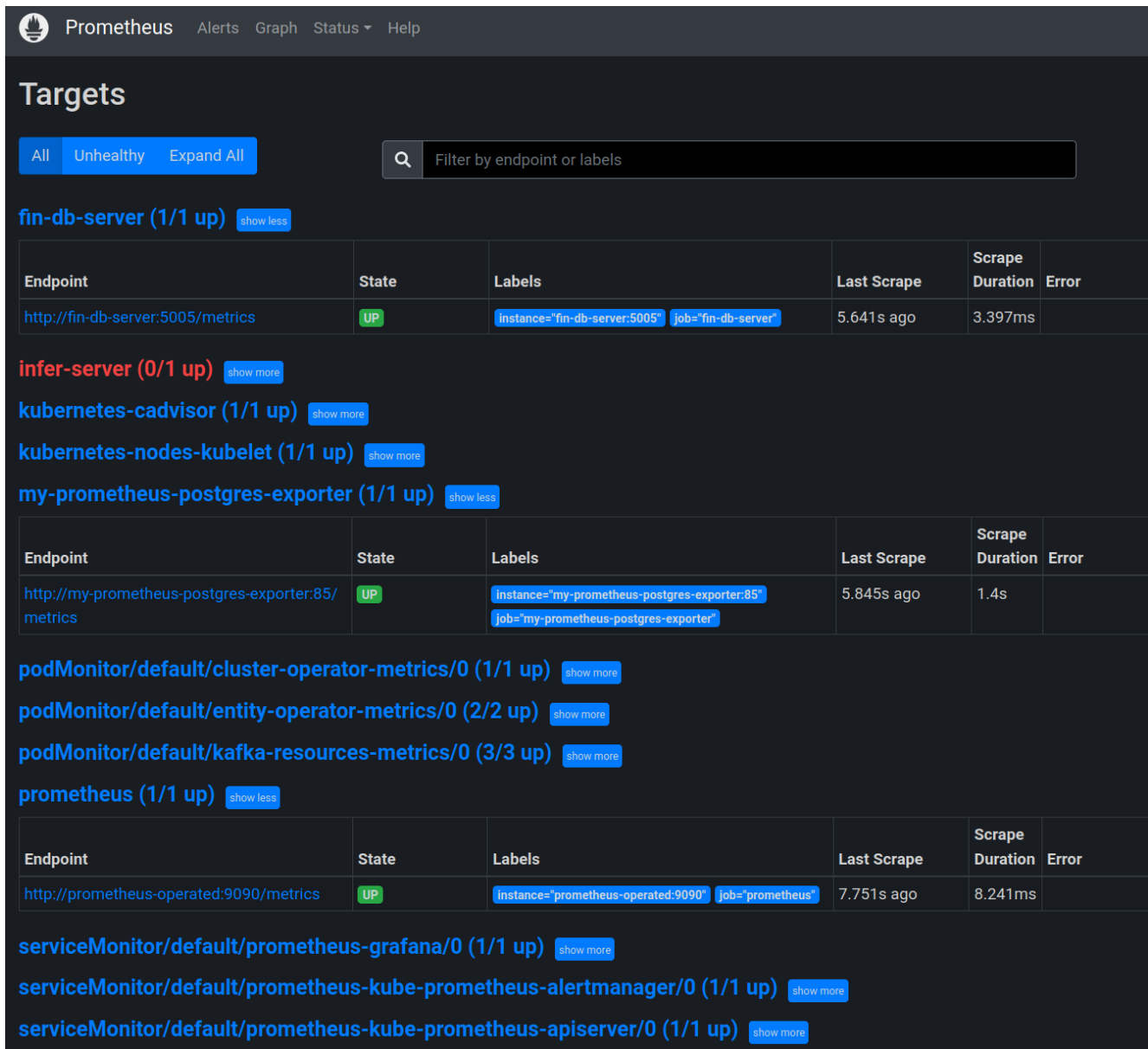


Figure 1.7: Prometheus targets in the GUI.

There are any number of queries one can carry out but the graphing could be easier to follow. This leads us to Grafana, Section 1.2.2.

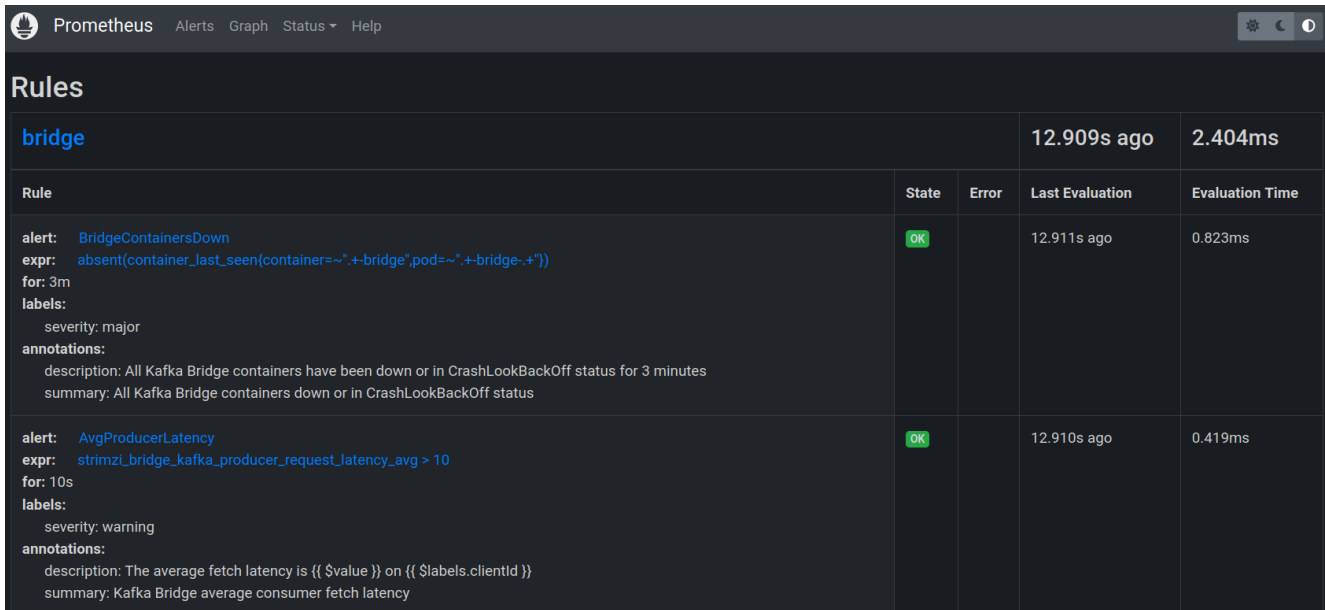
Prometheus recording rules are used to define which metrics are recorded by which components. This is a mechanism that allows for the monitoring of components to be controlled by the Prometheus server. The rules are configured in a yaml, Figure 1.8 and are represented in the Prometheus GUI as Figure 1.9 The following figure shows the recording rules for the components in the Prometheus UI:

```

apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    role: alert-rules
    app: strimzi
    name: prometheus-k8s-rules
spec:
  groups:
  - name: kafka
    rules:
    - alert: KafkaRunningOutOfSpace
      expr: kubelet_volume_stats_available_bytes{persistentvolumeclaim=~"data(-[0-9]+)?-(.+)-kafka-[0-9]+"} * 100
      for: 10s
      labels:
        severity: warning
      annotations:
        summary: 'Kafka is running out of free disk space'
        description: 'There are only {{ $value }} percent available at {{ $labels.persistentvolumeclaim }} PVC'
    - alert: UnderReplicatedPartitions
      expr: kafka_server_replicamanager_underreplicatedpartitions > 0
      for: 10s
      labels:
        severity: warning
      annotations:
        summary: 'Kafka under replicated partitions'
        description: 'There are {{ $value }} under replicated partitions on {{ $labels.kubernetes_pod_name }}'

```

Figure 1.8: Prometheus recording rules yaml.



Rule	State	Error	Last Evaluation	Evaluation Time
bridge alert: BridgeContainersDown expr: <code>absent(container_last_seen(container=~".+",bridge".pod=~".+",bridge-.+))</code> for: 3m labels: severity: major annotations: description: All Kafka Bridge containers have been down or in CrashLookBackOff status for 3 minutes summary: All Kafka Bridge containers down or in CrashLookBackOff status	OK		12.911s ago	0.823ms
alert: AvgProducerLatency expr: <code>strimzi_bridge_kafka_producer_request_latency_avg > 10</code> for: 10s labels: severity: warning annotations: description: The average fetch latency is {{ \$value }} on {{ \$labels.clientId }} summary: Kafka Bridge average consumer fetch latency	OK		12.910s ago	0.419ms

Figure 1.9: Prometheus recording rules in the Prom GUI

1.2.2 Grafana

Whilst Grafana and Prometheus work hand in hand, and are usually deployed together, Grafana is a separate project, also written in Golang. Grafana may be deployed on its own or can be deployed as part of the Prometheus Operator. This is how this paper implements Grafana. Since Grafana is a separate project, it is not bound to only use Prometheus as its data source. Grafana can also use other data sources such as AWS CloudWatch, Elasticsearch or a collection of databases and other tools[5]. Grafana also supports logging, such as *Loki* another tool from the makers of Grafana, as a data source.

Grafana is mainly used as a visualisation tool. There is an ‘Explore’ section to the GUI in which PromQL queries can be run. The results of the query are returned. But where Grafana really stands

out is the dashboard view. Here any number of graphs can be created and saved into a dashboard to allow for quick and easy visualisation of an application or any source of data that is needed. The graphs are displayed in a grid, and the user can drag and drop them around to create a layout, as per Figure 1.10.

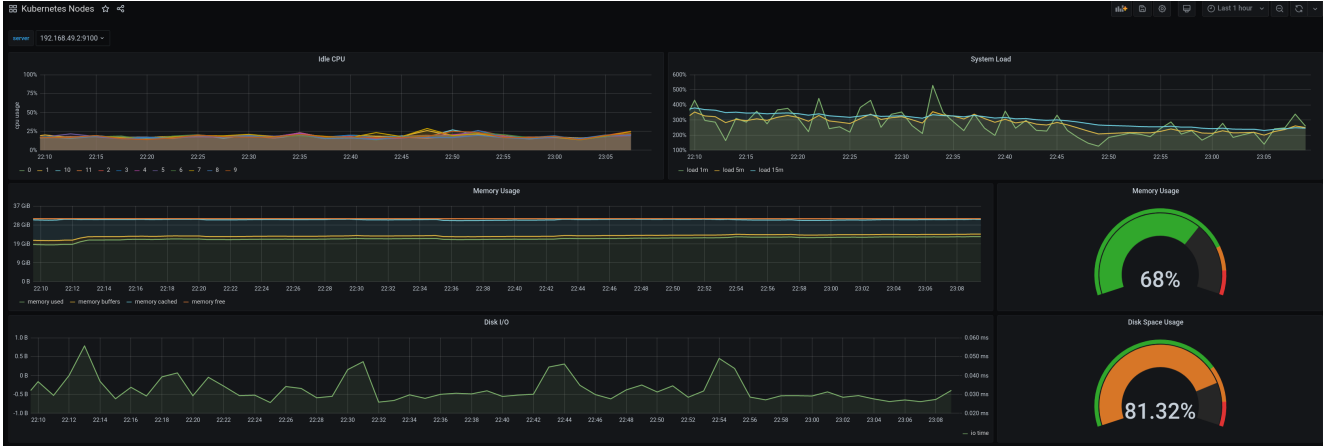


Figure 1.10: Grafana dashboard layout. From local cluster

Grafana dashboards are defined in JSON files and can be shared with other users. Some custom dashboards are available on the Grafana website, many have thousands and some even millions of downloads.

Grafana also has alerting capabilities. Alerts can be defined to be triggered when a certain condition is met, although this is not something that this paper will cover.

1.2.3 Alertmanager

Alertmanager is the component of the Prometheus stack, Figure 1.3, which is responsible for sending alerts to users. It acts as a translation layer between the Prometheus server and popular alerting tools such as Slack, Pagerduty, email etc.

Alerts are defined in Prometheus in yaml format. They are very similar to the way that Prometheus recording rules are defined.

The number of alerts can quickly get out of hand, for example, if a node goes down / offline which contains the system's database deployment then Prometheus will send an alert back on behalf of all configured applications. This may amount to numerous alerts, however, Alertmanager is designed to group alerts of a similar type together. These groups are then sent out in a single notification instead of dozens [22].

Groups can be custom defined in the routing file. There are also ways of silencing alerts, such as silencing alerts for a certain time period and suppressing alerts that are already firing.

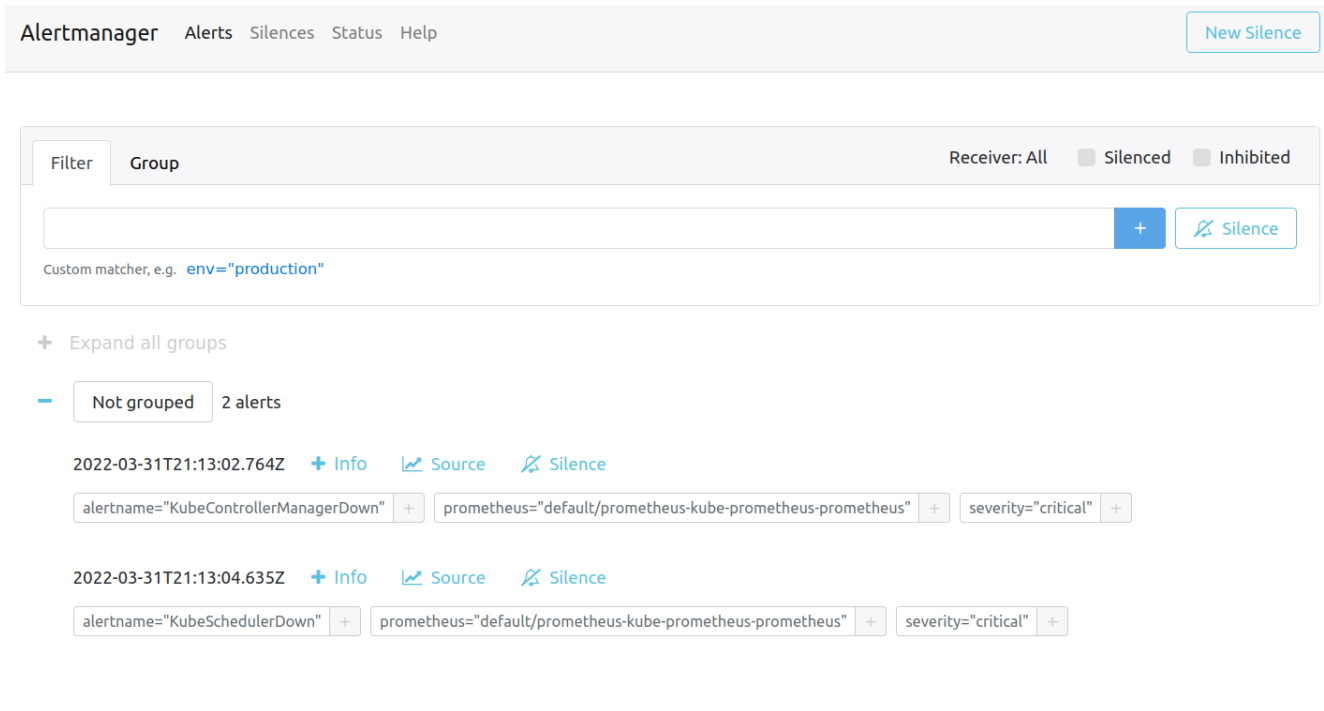


Figure 1.11: Alertmanager GUI depicting some alerts

1.3 Deployment

The first aim to be tackled was the deployment of the Prometheus Operator. The automation goal meant that choosing installation via Helm would be the optimal choice.

1.3.1 Prometheus Operator Choice

There are a number of Prometheus Operators out there and for this paper two differing operators were installed. At first a project with many differing components coming preinstalled / configured was chosen. Whilst this came with some Grafana dashboards along with some other perks, there was also a huge amount of resources that were applied to the cluster via the Helm chart.

This meant that the next task of retrospectively activating monitoring on the Kafka cluster deployed by the Strimzi Operator was incredibly difficult. More freedom from preexisting CRDs, service roles and other resources was needed. To demonstrate the verbosity of the challenge at hand Figure 1.12 shows how many lines of code are deployed via Helm as part of the ‘slimmer’ (and subsequently chosen) operator. The smaller operator chart has almost 45,000 lines of code.

```

44843     admissionReviewVersions: ["v1", "v1beta1"]
44844     sideEffects: None
44845
44846     NOTES:
44847     kube-prometheus-stack has been installed. Check its status by running:
44848     kubectl --namespace default get pods -l "release=prometheus"
44849

```

Figure 1.12: Helm chart for Prometheus Operator.

Trying to figure out exactly which resource from the Strimzi Operator needed to be updated and how they interacted with the Prometheus Operator was very challenging. In the end the operator used throughout the paper was a Prometheus Community Operator [15].

1.3.2 Monitoring Kubernetes Node

The monitoring of the Kubernetes node was somewhat straightforward. With the initial Prometheus Operator installed, the metrics and dashboards were automatically generated. The metrics were collected from the node and the metrics were then pushed to the Prometheus server which were then queried by Grafana. However, upon the change in Prom Operator, this did not come ‘for free’.

To monitor the Kubernetes node the *Node Exporter*[18] needed to be installed on the node. This needed to be a `daemonset` as the container needed to be run as root to gain access to the underlying OS for the metrics. Using this resource[13] for rough guidance the node was successfully spun up. A Prometheus job is then applied, akin to Figure 1.4, which monitors the Node Exporter Service Figure 1.13. The final step is to allow communication via a Kube Service Figure 1.13.

```

dimdakis@dimdakis-xps in ~/fyp/moneykey_transactions-api on infer_server_kube
[base]$ kube describe service prometheus-prometheus-node-exporter - yaml
Name: prometheus-prometheus-node-exporter
Namespace: default
Labels: app=prometheus-node-exporter
        app.kubernetes.io/managed-by=Helm
        chart=prometheus-node-exporter-3.1.0
        heritage=Helm
        jobLabel=node-exporter
        release=prometheus
Annotations: meta.helm.sh/release-name: prometheus
             meta.helm.sh/release-namespace: default
             prometheus.io/scrape: true
Selector: app=prometheus-node-exporter, release=prometheus
Type: ClusterIP
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.107.84.78
IPs: 10.107.84.78
Port: http-metrics 9100/TCP
TargetPort: 9100/TCP
Endpoints: 192.168.49.2:9100
Session Affinity: None
Events: <none>

```

Figure 1.13: Node Exporter Service.

The labels play a particularly important role for this resource as the Prom job had to match the required labels to allow for the monitoring to take place. Once these resources are in place, the node exporter status can be seen in the Prometheus GUI:

serviceMonitor/default/prometheus-prometheus-node-exporter/0 (1/1 up) show less						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
http://192.168.49.2:9100/metrics	UP	container="node-exporter" endpoints="http-metrics" instances="192.168.49.2:9100" job="node-exporter" namespace="default" pod="prometheus-prometheus-node-exporter-5xxkd" service="prometheus-prometheus-node-exporter"	24.250s ago	211.419ms		

Figure 1.14: Node Exporter successfully registered as a healthy target in the Prometheus GUI.

There are a number of great community dashboards available for the Node Exporter for Grafana, *Node Exporter Full* [19] was chosen:



Figure 1.15: Node Exporter dashboard.

The dashboard is a fantastic way to monitor the Kubernetes node. There are a plethora of charts available with extremely useful information.

1.3.3 Monitoring Strimzi and Kafka

This section covers the monitoring of the Kafka cluster deployed by the Strimzi Operator. Following the Strimzi Operator documentation [7] along with many other online resources such as [9], and others in the bibliography (Section 1.4) the Kafka cluster is successfully being monitored by Prometheus. Custom rules for Prometheus to interact specifically with Kafka cluster are created, as per Figure 1.8. The rules template along with some default rules is provided by the team behind the Strimzi Operator.

With the correct rules in place, the next step was to create additional scrape jobs for Prometheus to target the Strimzi cluster, this is done by creating a new secret with the details of the scrape configuration, as per Figure 1.16. The secret is then applied to Kube cluster:


```

1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: additional-scrape-configs
5  type: Opaque
6  stringData:
7    prometheus-additional.yaml: |
8      - job_name: kubernetes-cadvisor
9        honor_labels: true
10       scrape_interval: 10s
11       scrape_timeout: 10s
12       metrics_path: /metrics/cadvisor
13       scheme: https
14       kubernetes_sd_configs:
15         - role: node
16           namespaces:
17             names: []
18       bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount
19       tls_config:
20         ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
21         insecure_skip_verify: true

```

Figure 1.16: Scrape Config Secret.

The Prometheus Operator allows for the monitoring of pods or services via the `ServiceMonitor` or `PodMonitor` resources. The documentation explaining the difference and which to choose / what they actually do is not the greatest as pointed out by a Prom Operator contributor in this GitHub issue, [21]. The `ServiceMonitor` is the preferred option as services by their very nature already have exposed endpoints. The `PodMonitor` option should be used if there is an application needed in the cluster which, for whatever reason, cannot have a service exposed.

Heeding this advice, a `ServiceMonitor` was applied. It was non-functional. The reason being that many of the components in the Kafka cluster did not have / work very well with a service. In particular the Entity Operator. The Entity Operator is deployed by the Strimzi Operator but it itself, is comprised of the Topics Operator and the User Operator. As can be seen, the issue of nested Operators and resources can get quite deep and confusing.

Another unfortunate part of the retrospective application is that debugging is extremely time-consuming and one doesn't know what resource is causing the issue as successful monitoring only comes with all successfully configured resources. This means that any resource applied in the cluster may be the culprit for the issue.

As an example, there may have been no issue with the Prom Rules (there was, but `kube logs <pod-name>` is an effective way to determine if the issue is yaml syntax related) but the issue may be with a nested operator or another applied resource.

Upon further research, the decision was taken to use the more inclusive `PodMonitor` resource. This method is more flexible as it allows for the monitoring of any pod resource, but it is seen as the less favorable method in the grander Kubernetes community. During the research to implement the `PodMonitor` resource, it was noticed that this is the primary way which metrics are collected from Strimzi currently but is an active area of discussion on GitHub. In fact, the ability to use the `PodMonitor` was born from the exact issue that this paper faced and the PR to allow Strimzi to use the `PodMonitor` was only merged in summer '20[31]. This now appears to be the de facto methodology for monitoring with Strimzi.

With the decision made and research conducted, the `PodMonitor` resource was applied to the cluster, as per Figure 1.17, as can be observed a separate `PodMonitor` resource must be created for each of the Kafka components along with the operators.

```

You, 3 days ago | 1 author (You) | com.coreos.monitoring.v1.PodMonitor (v1@podmonitor.json)
1  apiVersion: monitoring.coreos.com/v1
2  kind: PodMonitor
3  metadata:
4    name: cluster-operator-metrics
5    labels:
6      app: strimzi
7  spec:
8    selector:
9      matchLabels:
10       strimzi.io/kind: cluster-operator
11    namespaceSelector:
12      matchNames:
13       - default
14    podMetricsEndpoints:
15     - path: /metrics
16       port: http
17  ---
18  apiVersion: monitoring.coreos.com/v1
19  kind: PodMonitor
20  metadata:
21    name: entity-operator-metrics
22    labels:
23      app: strimzi
24  spec:
25    selector:
26      matchLabels:
27       app.kubernetes.io/name: entity-operator
28    namespaceSelector:
29      matchNames:
30       - default
31    podMetricsEndpoints:
32     - path: /metrics
33       port: healthcheck
34  ---
35  apiVersion: monitoring.coreos.com/v1
36  kind: PodMonitor
37  metadata:

```

Figure 1.17: PodMonitor applied to the Strimzi Operator (only some of the code is shown as it is repetitive).

The Kafka resource, of `kind: Kafka` (Strimzi native ‘kind’), must be edited to include the metrics configuration (Note: The entire file is substantial, therefore, only the additions are shown):

Listing 1.1: Kafka resource with metrics configuration applied.

```

1  ... some code...
2
3  metricsConfig:
4    type: jmxPrometheusExporter
5    valueFrom:
6      configMapKeyRef:
7        key: kafka-metrics-config.yml
8        name: kafka-metrics
9
10 ... more code...

```

This enables Strimzi to locate the metrics configuration file and have it used by the broker. Similarly, in the section of the Kafka resource, the Zookeeper metrics config file needs to be included:

Listing 1.2: Zookeeper resource with metrics configuration applied.

```

1  ... some code...
2
3  metricsConfig:
4    type: jmxPrometheusExporter
5    valueFrom:
6      configMapKeyRef:
7        key: zookeeper-metrics-config.yml
8        name: zookeeper-metrics
9
10 ... more code...

```

This Zookeeper metrics config file is very similar to the Kafka metrics config file, Figure 1.18, but is written for Zookeeper. This is also a community file.

The final alteration of the Kafka resource is to include the configuration for the Kafka Exporter:

Listing 1.3: Kafka resource with Kafka Exporter configuration applied.

```

1  ... some code...
2
3  kafkaExporter:
4    groupRegex: ".*"
5    topicRegex: ".*"
6    logging: debug
7    enableSaramaLogging: true
8    readinessProbe:
9      initialDelaySeconds: 15
10     timeoutSeconds: 5
11    livenessProbe:
12      initialDelaySeconds: 15
13      timeoutSeconds: 5
14
15 ... more code...

```

Once the Kafka Exporter is added to the Kafka resource the Strimzi Operator handles the deployment of the Kafka Exporter.

The new metrics ConfigMap with the metrics configuration for the Kafka Broker is the next resource applied. This is done as per Figure 1.18. This is a community file for Kafka metrics, and applied to the cluster.

```

You, 4 days ago | 1 author (You) | io.k8s.api.core.v1.ConfigMap (v1@configmap.json)
1 kind: ConfigMap
2 apiVersion: v1
3 metadata:
4   name: kafka-metrics
5   labels:
6     app: strimzi
7 data:
8   kafka-metrics-config.yml: |
9     # See https://github.com/prometheus/jmx\_exporter for more info about JMX Promethe
10    lowercaseOutputName: true
11    rules:
12      # Special cases and very specific rules
13      - pattern: kafka.server<type=(.+), name=(.+), clientId=(.+), topic=(.+), partiti
14        name: kafka_server_$1_$2
15        type: GAUGE
16        labels:
17          clientId: "$3"
18          topic: "$4"
19          partition: "$5"
20      - pattern: kafka.server<type=(.+), name=(.+), clientId=(.+), brokerHost=(.+), bro
21        name: kafka_server_$1_$2
22        type: GAUGE
23        labels:
24          clientId: "$3"
25          broker: "$4:$5"

```

Figure 1.18: ConfigMap for Kafka broker for Prometheus integration.

Following on, a `ClusterRole`, `ClusterRoleBinding` and `ServiceAccount` are created to allow Prometheus to interact with the Strimzi components as per Figure 1.19. This yaml successfully created the required resources although quite a bit of debugging was required to figure out how to piece this all together. A ClusterRole AKA a `Rules Based Access Control (RBAC) Role` is a role which is needed to set permissions in a given name space. The role must then be bound to the created ServiceAccount. A ServiceAccount is akin to a user account, when a user interacts with the Kube API server via kubectl, however it is the type of account that is given to any process running in a container in a particular namespace. There is usually a service account per application, if one is needed.

Now that all the required resources have been applied to the Kubernetes cluster, a `jmx exporter` [14] is required to translate the metrics from Kafka brokers to Prometheus. A similar approach is needed as the go-between for Prometheus and Zookeeper. Luckily these are already present in the Strimzi Operator, and the operator will provision what is needed so, no additional configuration is needed.

```

You, 3 days ago | 1 author (You) | v1@prometheus.json | com.coreos.monitoring.v1.Prom
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: ClusterRole
3  metadata:
4    name: prometheus-server
5    labels:
6      app: strimzi
7  rules:
8    - apiGroups: [""]
9      resources:
10       - nodes
11       - nodes/proxy
12       - services
13       - endpoints
14       - pods
15     verbs: ["get", "list", "watch"]
16    - apiGroups:
17      - extensions
18      resources:
19        - ingresses
20     verbs: ["get", "list", "watch"]
21    - nonResourceURLs: ["/metrics"]
22     verbs: ["get"]
23
24  ---
25  apiVersion: v1
26  kind: ServiceAccount
27  metadata:
28    name: prometheus-server
29    labels:
30      app: strimzi
31
32  ---
33  apiVersion: rbac.authorization.k8s.io/v1
34  kind: ClusterRoleBinding
35  metadata:
36    name: prometheus-server
37    labels:
38      app: strimzi
39  roleRef:
40    apiGroup: rbac.authorization.k8s.io

```

Figure 1.19: Prometheus YAML for Strimzi components.

With these steps all taken care of the Prometheus Operator should list the Strimzi components as targets.

podMonitor/default/cluster-operator-metrics/0 (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.8:8080/metrics	UP	container="strimzi-cluster-operator" endpoint="http" instance="172.17.0.8:8080" job="default/cluster-operator-metrics" namespace="default" pod="strimzi-cluster-operator-85bb4c6-77ldr"	1h 24m 38s ago	3.284ms	

Figure 1.20: Prometheus successfully defining Strimzi Operator as a target.

As can be observed, the Strimzi Operator itself is now a Prometheus target. Next is to check for the Topic Operator and the User Operator, which make up the Entity Operator, as per Figure 1.21.

podMonitor/default/entity-operator-metrics/0 (2/2 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.10:8080/metrics	UP	container="topic-operator" endpoint="healthcheck" instance="172.17.0.10:8080" job="default/entity-operator-metrics" namespace="default" pod="kafka-cluster-entity-operator-5fb4956bcd-hglwn"	1h 24m 34s ago	12.241ms	
http://172.17.0.10:8081/metrics	UP	container="user-operator" endpoint="healthcheck" instance="172.17.0.10:8081" job="default/entity-operator-metrics" namespace="default" pod="kafka-cluster-entity-operator-5fb4956bcd-hglwn"	1h 24m 45s ago	2.210ms	

Figure 1.21: Prometheus successfully defining both the Topic Operator and the User Operator as part of the Strimzi Entity Operator as a target.

Finally, we can get down to the actual Kafka components, as per Figure 1.22. For monitoring to be successful each Kafka component must be a Prometheus target and in a healthy state too.

podMonitor/default/kafka-resources-metrics/0 (3/3 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.25:9404/metrics	UP	container="kafka" endpoint="tcp-prometheus" instance="172.17.0.25:9404" job="default/kafka-resources-metrics" kubernetes_pod_name="kafka-cluster-kafka-0" namespace="default" node_ip="192.168.49.2" node_name="minikube" pod="kafka-cluster-kafka-0" strimzi_io_cluster="kafka-cluster" strimzi_io_kind="Kafka" strimzi_io_name="kafka-cluster-kafka"	1h 24m 34s ago	2.931s	
http://172.17.0.24:9404/metrics	UP	container="kafka-cluster-kafka-exporter" endpoint="tcp-prometheus" instance="172.17.0.24:9404" job="default/kafka-resources-metrics" kubernetes_pod_name="kafka-cluster-kafka-exporter-5494b4f589-8l96f" namespace="default" node_ip="192.168.49.2" node_name="minikube" pod="kafka-cluster-kafka-exporter-5494b4f589-8l96f" strimzi_io_cluster="kafka-cluster" strimzi_io_kind="Kafka" strimzi_io_name="kafka-cluster-kafka-exporter"	1h 24m 36s ago	49.694ms	
http://172.17.0.26:9404/metrics	UP	container="zookeeper" endpoint="tcp-prometheus" instance="172.17.0.26:9404" job="default/kafka-resources-metrics" kubernetes_pod_name="kafka-cluster-zookeeper-0" namespace="default" node_ip="192.168.49.2" node_name="minikube" pod="kafka-cluster-zookeeper-0" strimzi_io_cluster="kafka-cluster" strimzi_io_kind="Kafka" strimzi_io_name="kafka-cluster-zookeeper"	1h 24m 33s ago	51.210ms	

Figure 1.22: Prometheus successfully defining the Kafka components as targets.

As this is now the case; Zookeeper, the Kafka broker and the Kafka exporter are now targets, Figure 1.22, queries can be run against the components via Prometheus UI:

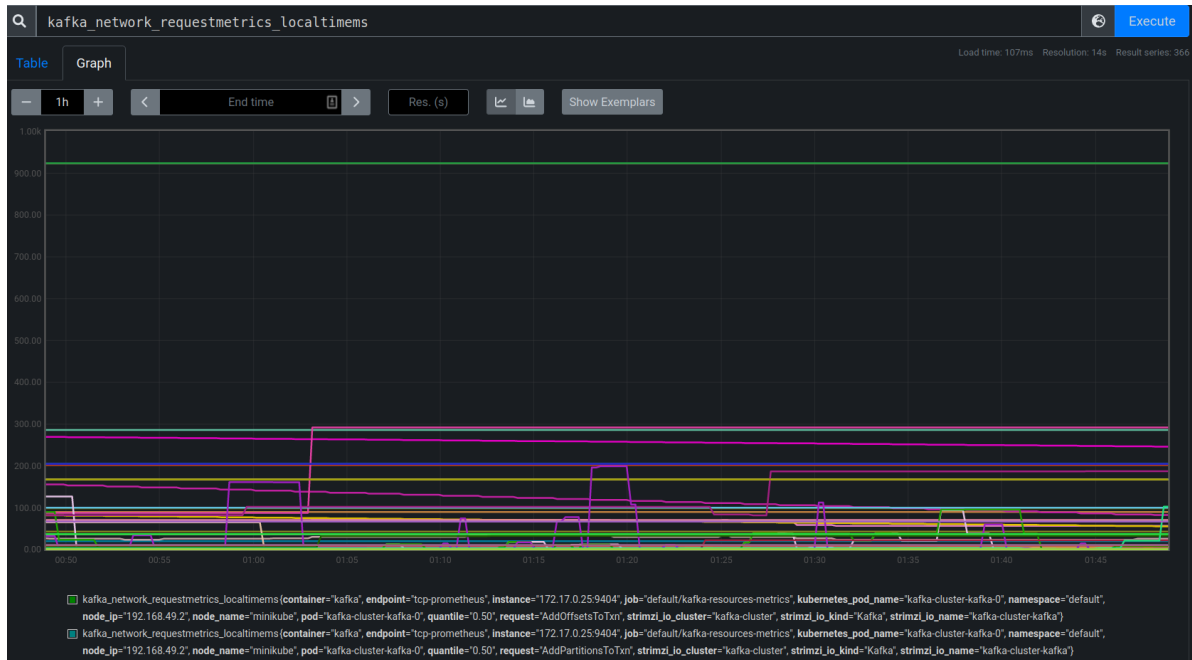


Figure 1.23: Prometheus successfully querying the Kafka components via the UI.

Back to the open-source community to find some Grafana dashboards for Kafka Monitoring and the end product is the following:

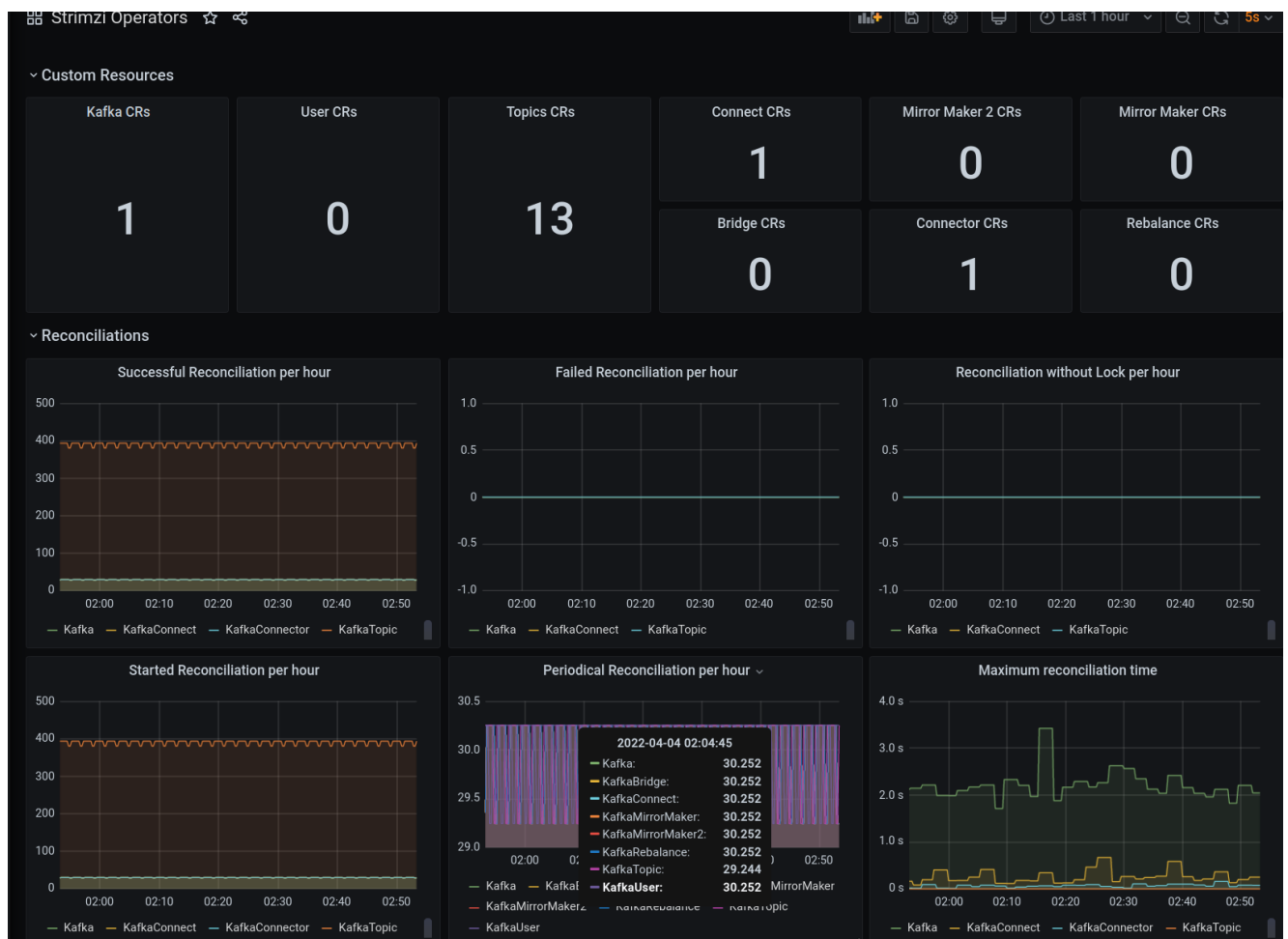


Figure 1.24: Grafana dashboard for the Strimzi Operator monitoring.



Figure 1.25: Grafana dashboard for the Kafka broker.

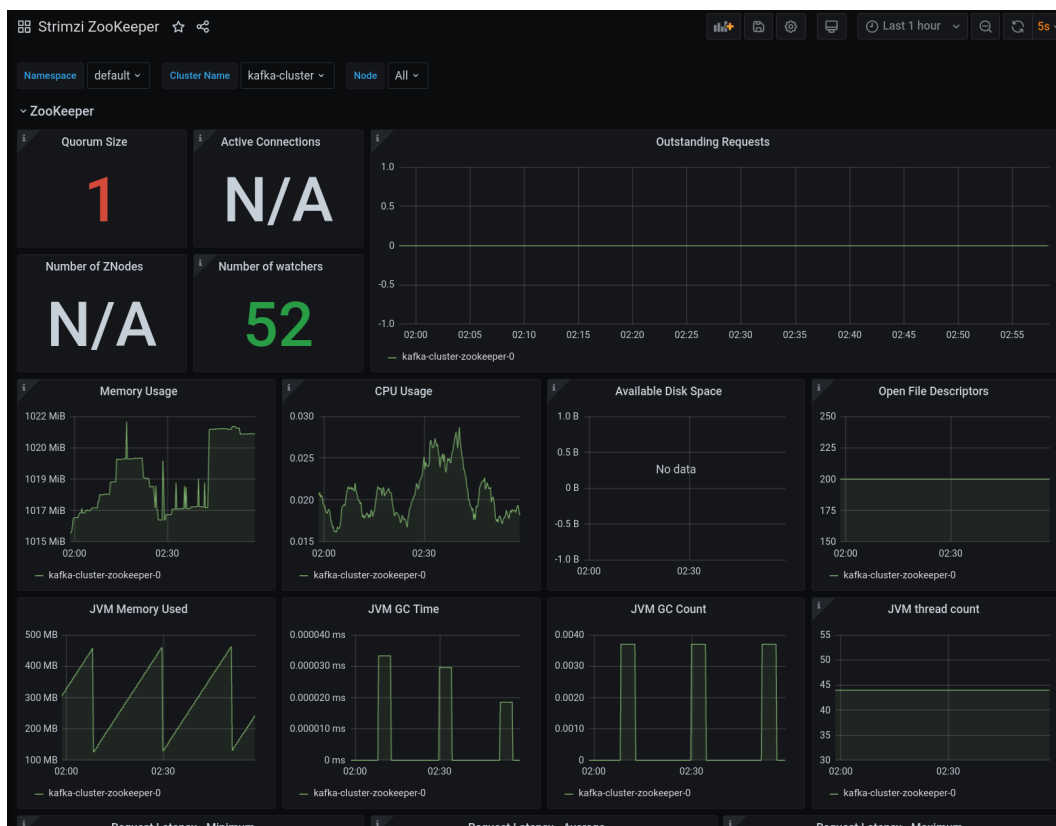


Figure 1.26: Grafana dashboard for the Zookeeper deployment.

The Kafka Exporter shows some very interesting metrics in relation to the actual data stored in the Kafka brokers in topics.

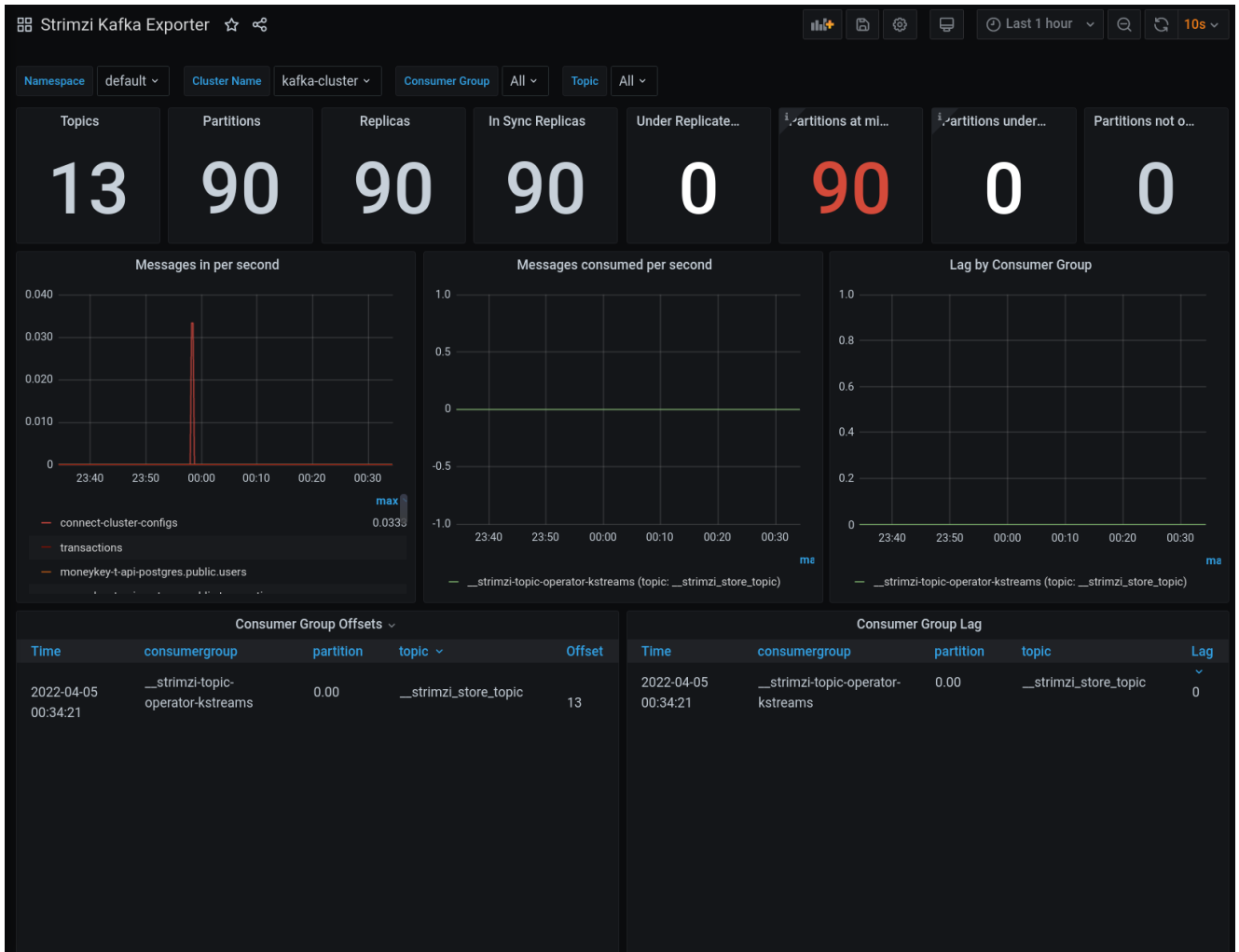


Figure 1.27: Grafana dashboard for the Kafka exporter dashboard I/II.

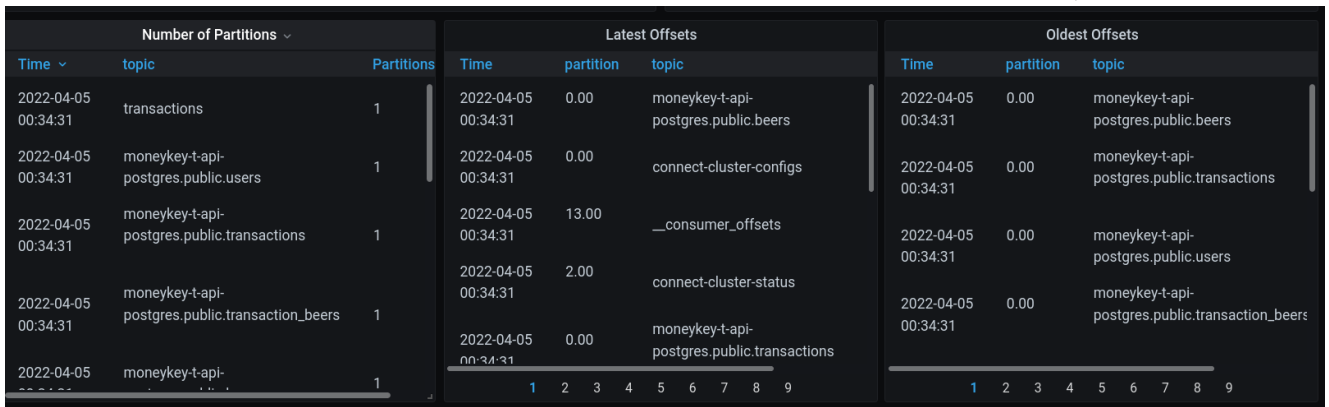


Figure 1.28: Grafana dashboard for the Kafka exporter dashboard II/II.

As can be observed all topics are listed along partitions and offsets.

This section has proven the age-old saying, ‘fail to prepare, prepare to fail’. There is no sound rationale for not implementing metrics from the start. The amount of data that can be gleamed is extraordinary. Once one becomes familiar with PromQL the ability to query the metrics becomes much easier.

1.3.4 Monitoring Flask Server

This section takes a look at the implementation of monitoring an application from the start, i.e. having metrics in focus during the design and development phase of the app. For this a simple Flask server has been created and containerised. This wasn't as straightforward as initially planned on account of the chosen technology. Initially the Flask App implemented Gunicorn[12] as the WSGI server, however, this proved to be too much configuration in the time that was remaining before the deadline. The reason being is that Gunicorn can run several processes in parallel, which is not ideal for the monitoring. Much time was spent trying to figure out how to achieve this, but could not be done. It is, however, possible to use Gunicorn with Prometheus there are workarounds involving Gunicorn's `multi-process-mode`. This implementation also needs a `CollectorRegistry` to store the metrics before until the Prometheus scrape. With time running out, the paper switched implementation to a simple Flask server (no Gunicorn multiprocessing parallelism). This approach was successfully implemented, although Flask is not a production ready solution, therefore it is somewhat disappointing that the Gunicorn approach failed.

The Flask app utilizes the `prometheus_flask_exporter` to implement the Prometheus data types in python, expose the metrics on a `/metrics` endpoint and as such act a translational layer between the Flask App and the Prometheus server. The code for the app is as follows:

Listing 1.4: Flask App for Prometheus monitoring.

```

1 import logging as log
2 from flask import Flask, request
3 from prometheus_flask_exporter import PrometheusMetrics
4
5 app = Flask(__name__)
6 metrics = PrometheusMetrics(app)
7
8 metrics.info('app_info', 'Application info', version='0.0.1')
9
10 # GET index route
11 @app.route('/', methods=['GET'])
12 def index():
13     return 'Hello, World!'
14
15 @app.route('/index')
16 def index_page():
17     return 'Yasu, World!'
18
19 if __name__ == '__main__':
20     # load_dotenv()
21
22     # log to stdout
23     log.basicConfig(level=log.INFO)
24     log.info('Starting up')
25     app.run(host='0.0.0.0', port=5005)
26     log.info('Listening on http://localhost:5005')

```

The metrics are called in line 7 and 9. The rest of the app is straightforward as to is the Dockerfile to containerise the app.

The next steps are to create and apply the deployment resources for the Flask App.

Listing 1.5: Deployment Resources for the Flask App

```

1 apiVersion: apps/v1
2 kind: Deployment                                # Type of the kubernetes resource
3 metadata:
4   name: fin-db-server                          # Name of the Kubernetes resource
5   labels:
6     app: fin-db-server
7 spec:
8   replicas: 1                                  # Number of pods to run at any given time
9   selector:
10    matchLabels:
11      app: fin-db-server                        # This deployment applies to any Pods
12      matching the specified label
13 template:                                     # This deployment will create a set of
14   metadata:                                   pods using the configurations in this template
15     labels:                                  # The labels that will be applied to all
16       app: fin-db-server                    of the pods in this deployment
17 spec:                                       # Spec for the container which will run
18   containers:                               in the Pod
19     - name: fin-db-server
20       image: dimakis/fin-db:latest          # The image we are getting from dockerhub
21       imagePullPolicy: Always
22       # imagePullPolicy: Always
23       #IfNotPresent                         # If we have not pulled it before, get
24       it from dockerhub
25   ports:
26     - name: http
27       containerPort: 5005                  # Should match the port number that the
28       Go application listens on

```

Following on from the deployment a service needs to be created for the Flask App:

Listing 1.6: Code for the Flask Service.

```

1 apiVersion: v1
2 kind: Service                                # Type of the kubernetes resource
3 metadata:
4   name: fin-db-server                        # Name of the Kubernetes resource
5   labels:                                   # Labels that will be applied to the resource
6     app: fin-db-server
7 spec:
8   type: NodePort                            # Gives each node an external IP that's accessible from
9   selector:                                outside the cluster and also opens a Port.
10    app: fin-db-server                      # Selects any Pod with labels 'app=fin-db-server'
11   ports:
12     - name: http
13       port: 5005
14       targetPort: 5005

```

Prometheus must now be configured to scrape the Flask App. This is one again done by adding a Prometheus job to the `additional-scrape-config` secret:

Listing 1.7: Prometheus job for the Flask App.

```

1 - job_name: infer-server
2   scrape_interval: 10s
3   scrape_timeout: 10s
4   static_configs :
5     - targets: ["infer-server:5000"]

```

Once this job applied and Kubernetes completes its control loop, the Prometheus server should recognise the Flask app as a scrape target:

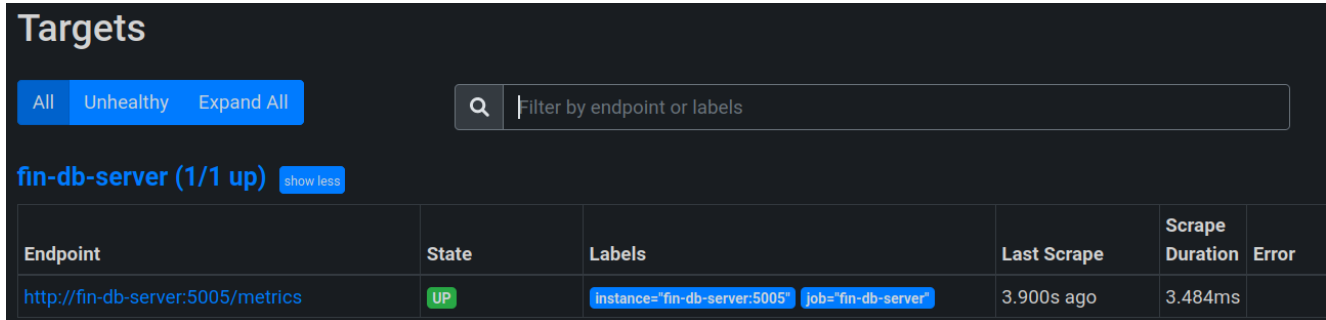


Figure 1.29: Prometheus successfully registering the Flask App as a target.

A new dashboard is needed for the Flask App. This is done by adding a new dashboard to Grafana, running some curl commands in a loop and viewing the output:

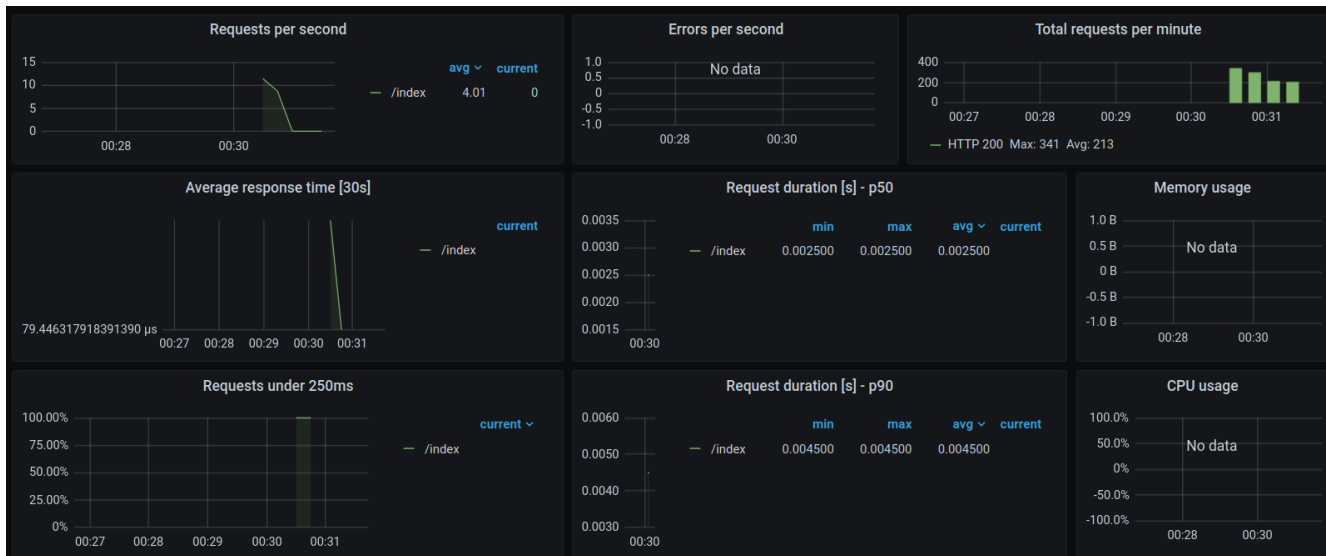


Figure 1.30: Grafana dashboard for the Flask application.

1.3.5 Additional PostGRES Database Exporter for Prometheus

With the Flask App deployed, Prometheus and Grafana both operational, an unsuccessful attempt was made at integrating metrics from a postgresql database. This was done via the `prometheus_postgres_exporter` which is a Prometheus Exporter for Postgres. The exporter is successfully implemented, and is discoverable by Prometheus but even with the correct, user, db name secrets and other necessary credentials it is not able to scrape the database.

my-prometheus-postgres-exporter (1/1 up) show less					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://my-prometheus-postgres-exporter.85/metrics	UP	instance="my-prometheus-postgres-exporter.85" job="my-prometheus-postgres-exporter"	9.688s ago	1.9s	

Figure 1.31: Target successful for Prometheus Postgres Exporter deployment.

Quite a bit of time was spent on this too.

1.3.6 Prometheus Self Monitoring

The final monitoring component added to the stack was the Prometheus self monitoring. Whilst implementing and troubleshooting the various monitoring components in the stack, it was discovered that the Prometheus server itself serves up its own metrics on its own endpoint. As Prometheus is already configured as a data source in Grafana the implementation of this was trivial. The dashboard was merely imported and straight away started receiving metrics.

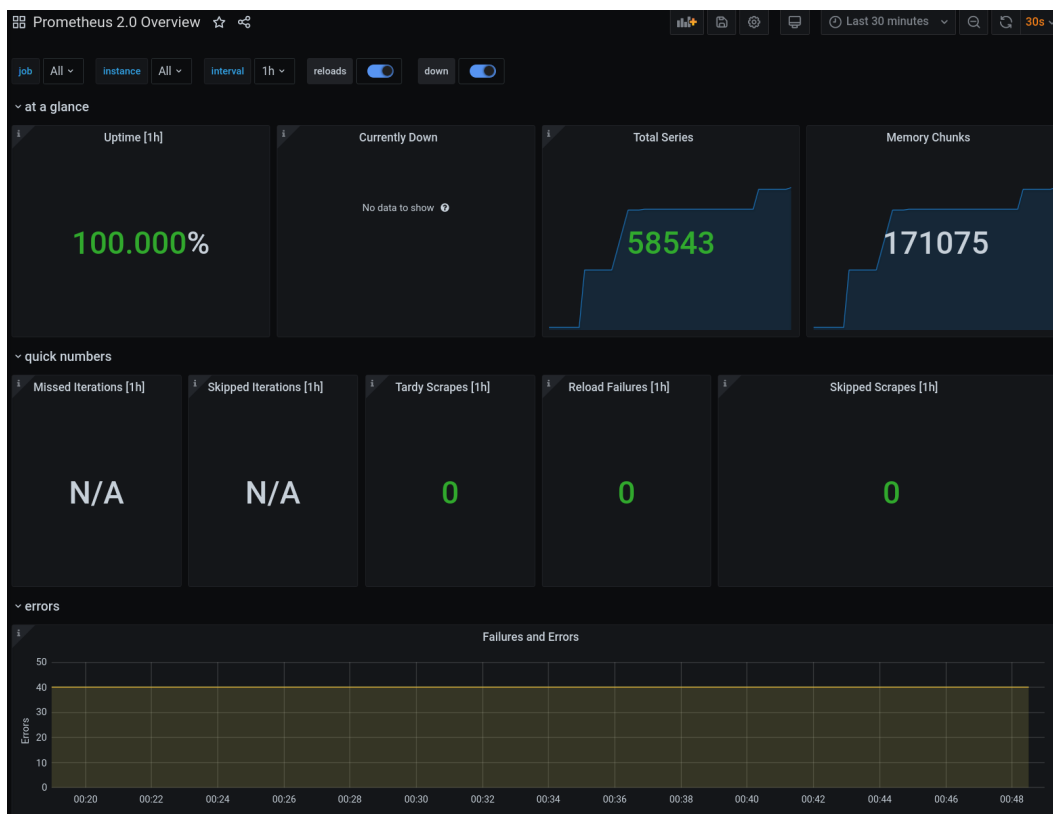


Figure 1.32: Prometheus self monitoring.

1.4 Conclusion

This paper looked at the various ways to implement Prometheus and tried to cover the differing scenarios one might face when implementing monitoring. It isn't always the case that monitoring was left out from the cluster creation process, sometimes the organisation may be utilizing a different monitoring stack and decide to switch, with minimal downtime to the system an obvious goal. This paper shows that, whilst some downtime is necessary in this set up, a full production cluster with multiple nodes and replicas which may be drained in small batches or if the org has a blue-green deployment strategy then the crossover may occur with no downtime required.

Every time I work with Kubernetes the technology finds new ways to amaze me. Firstly with just how quickly and painlessly (after the admittedly steep learning curve) components can be upgraded. I

tested new versions of the applications numerous times and Kubernetes never failed to bring up the pod. All very quickly too. I even brought up a pod somewhat by accident, with dubious configurations and yet again, Kube had the application up and running (and actually quite difficult to kill).

The Operator pattern, the extension of the Kubernetes API via this pattern is a fantastic way of provisioning and managing resources. Once one is familiar-ish with what components need and the different types of 'kinds' and native Kube resources (ConfigMaps, Deployments etc.) then someone with little Kubernetes experience can effectively spin up a huge amount of highly functional resources by utilizing template files / resource definitions provided by the teams who work on the technologies every day.

The ability to nest operators and use them to create a larger resource is an extremely powerful concept.

Prometheus works very well. The ecosystem is quite mature, and the community is very active. The amount of data that can be gleamed from the system is large but so to are the different metrics that can be extracted. Being able to query the data in various forms is a big plus. As to is the ecosystem at large. There are exporters / operators for any third party application I've looked for and as such I can see no reason why a team wouldn't avail of them. It is an integral tool in the quest for a Highly Available and efficient system.

Grafana is also a very useful tool. The visualisations and dashboards are beautiful and there are a lot of options, in terms of community dashboards freely available from the Grafana website, that work with a variety of data sources.

These technologies are definitely going to be part of future projects I work on. You don't actually know what you're missing until you have it. The dashboards and metrics are an amazing way to get a quick overview of the system in real time.

The choice of Strimzi to monitor was a great one. There are quite a few components and as such whilst working through them, I was able to gain a much better understanding of Kubernetes, the various components and how they fit together. I'm somewhat disappointed that I didn't get to fix the postgresql exporter and the Unicorn metrics, but I'm happy very with the overall experience.

I would like to explore Grafana's integration with other tools some more. I also toyed with the idea of deploying a time series database such as QuestDB or InfluxDB as I see the way time data can be utilized by exploring and querying Prometheus and it is definitely something I would like to see implemented.

Bibliography

- [1] Viktor Adam. *Prometheus Flask Exporter*. Apr. 2022.
- [2] *Charts/Stable/Prometheus-Operator at Master · Helm/Charts*. <https://github.com/helm/charts>.
- [3] *Cloud Native Computing Foundation*. <https://www.cncf.io/>.
- [4] *Configure Service Accounts for Pods*.
<https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/>.
- [5] *Data Sources*. <https://grafana.com/docs/grafana/latest/datasources/>.
- [6] *Deploy Prometheus Monitoring Stack to Kubernetes with a Single Helm Chart*.
<https://dev.to/kaitoi11/deploy-prometheus-monitoring-stack-to-kubernetes-with-a-single-helm-chart-2fbd>.
- [7] *Deploying and Upgrading (0.28.0)*.
<https://strimzi.io/docs/operators/latest/deploying.html#proc-metrics-kafka-deploy-options-str>.
- [8] Kirill Goltsman. *Deploying Traefik as Ingress Controller for Your Kubernetes Cluster*. May 2019.
- [9] *Grafana and Prometheus Setup With Strimzi, a.k.a. Kafka on Kubernetes. - DZone Integration*.
<https://dzone.com/articles/grafana-and-prometheus-setup-with-strimzi-aka-kafk>.
- [10] *Grafana: The Open Observability Platform*. <https://grafana.com/>.
- [11] Tom Gregory. *The 4 Types Of Prometheus Metrics*. Dec. 2019.
- [12] *Gunicorn - Python WSGI HTTP Server for UNIX*. <https://gunicorn.org/>.
- [13] *How To Setup Prometheus Node Exporter On Kubernetes*.
<https://devopscube.com/node-exporter-kubernetes/>. Apr. 2021.
- [14] *JMX Exporter*. Prometheus. Apr. 2022.
- [15] *Kube-Prometheus-Stack 34.7.1 · Prometheus/Prometheus-Community*.
<https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack>.
- [16] *Monitoring Python Flask Microservices with Prometheus · Viktor Adam's Blog*.
<https://blog.viktoradam.net/2020/05/11/prometheus-flask-exporter/>. May 2020.
- [17] Josphat Mutai. *Setup Prometheus and Grafana on Kubernetes Using Prometheus-Operator — ComputingForGeeks*.
<https://computingforgeeks.com/setup-prometheus-and-grafana-on-kubernetes/>. July 2021.
- [18] *Node Exporter*. Prometheus. Apr. 2022.
- [19] *Node Exporter Full Dashboard for Grafana*. <https://grafana.com/grafana/dashboards/1860>.
- [20] *O'Reilly's Microservices Adoption in 2020 Report Finds That 92% of Organizations Are Experiencing Success with Microservices*.
<https://www.businesswire.com/news/home/20200716005101/en/O%E2%80%99Reilly%E2%80%99s-Microservices-Adoption-in-2020-Report-Finds-that-92-of-Organizations-are-Experiencing-Success-with-Microservices>. July 2020.
- [21] *PodMonitor vs ServiceMonitor What Is the Difference? · Issue #3119 · Prometheus-Operator/Prometheus-Operator*.
<https://github.com/prometheus-operator/prometheus-operator/issues/3119>.
- [22] Prometheus. *Alertmanager — Prometheus*.
<https://prometheus.io/docs/alerting/latest/alertmanager/>.

Bibliography

- [23] Prometheus. *Client Libraries — Prometheus*.
<https://prometheus.io/docs/instrumenting/clientlibs/>.
- [24] Prometheus. *Exporters and Integrations — Prometheus*.
<https://prometheus.io/docs/instrumenting/exporters/>.
- [25] Prometheus. *Overview — Prometheus*. <https://prometheus.io/docs/introduction/overview/>.
- [26] Prometheus. *Prometheus - Monitoring System & Time Series Database*. <https://prometheus.io/>.
- [27] Prometheus. *Query Functions — Prometheus*.
<https://prometheus.io/docs/prometheus/latest/querying/functions/>.
- [28] Prometheus. *Querying Basics — Prometheus*.
<https://prometheus.io/docs/prometheus/latest/querying/basics/>.
- [29] *Prometheus-Flask-Exporter · PyPI*. <https://pypi.org/project/prometheus-flask-exporter/>.
- [30] *Strimzi Documentation (0.12.2)*. <https://strimzi.io/docs/0.12.2/>.
- [31] *Using PodMonitor for Scraping Kafka Related Metrics by Ppatierno · Pull Request #3351 · Strimzi/Strimzi-Kafka-Operator*. <https://github.com/strimzi/strimzi-kafka-operator/pull/3351>.
- [32] *Using RBAC Authorization*. <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>.