Application Stats Collection in Kubernetes via Telegraf Sidecars and Wavefront

by Bahubali Shetti (https://cloud.vmware.com/community/author/bshetti/)

This blog was originally posted here (https://medium.com/floatingapps/flask-django-and-mysql-stats-collection-via-telegraf-sidecars-wavefront-in-a-kubernetes-9dbb78dc905a), on August 7, 2018.

(This is a follow up for the blog: Monitoring VMware Kubernetes Engine and Application Metrics with Wavefront (https://cloud.vmware.com/community/2018/08/06/monitoring-vmware-kubernetes-engine-application-metrics-wavefront/))

Kubernetes (K8S), is becoming the defacto management tool to run applications homogeneously across resources (bare metal, public cloud, or private cloud). The single most widely deployed operational component in kubernetes is monitoring.

Almost everyone uses Prometheus in a cluster to aggregate the stats from the cluster. Grafana is then used to graph these stats.

Most articles are written showcasing Prometheus and grafana with a focus on cluster (node, pode, etc) stats. Rarely do any of these discuss application level stats.

While Prometheus has exporters (https://prometheus.io/docs/instrumenting/exporters/), i.e. mysql (https://github.com/prometheus/mysqld_exporter) (see setup (https://www.percona.com/blog/2016/02/29/graphing-mysql-performance-with-prometheus-and-grafana/)), nginx, there are alternative mechanisms to export application stats.

In this blog, I will explore the use of Telegraf, as a sidecar to extract stats from different application components such as Flask, Django, and mysql.

Flask – is a python based web framework used to build websites, and api servers.

Django – is a python based web framework, similar to Flask, but is generally used to facilitate the ease and creation of complex, database-driven websites.

Mysql – is an open-source relational database management system

Telegraf has a wide range of plugins. More than Prometheus' set of exporters. Telegraf can sendthese stats to multiple locations (i.e. Wavefront, Prometheus, etc). In this configuration I will showcase Wavefront (https://www.wavefront.com/sign-up/?utm_source=Website&utm_medium=referral&utm_campaign=website-top-ribbon).

Wavefront can aggregate all stats from all Kubernetes clusters vs Prometheus, which generally displays stats for the specific cluster its deployed in.

In subsequent blogs, I will explore use of telegraf with Prometheus for application stats in a specific cluster.

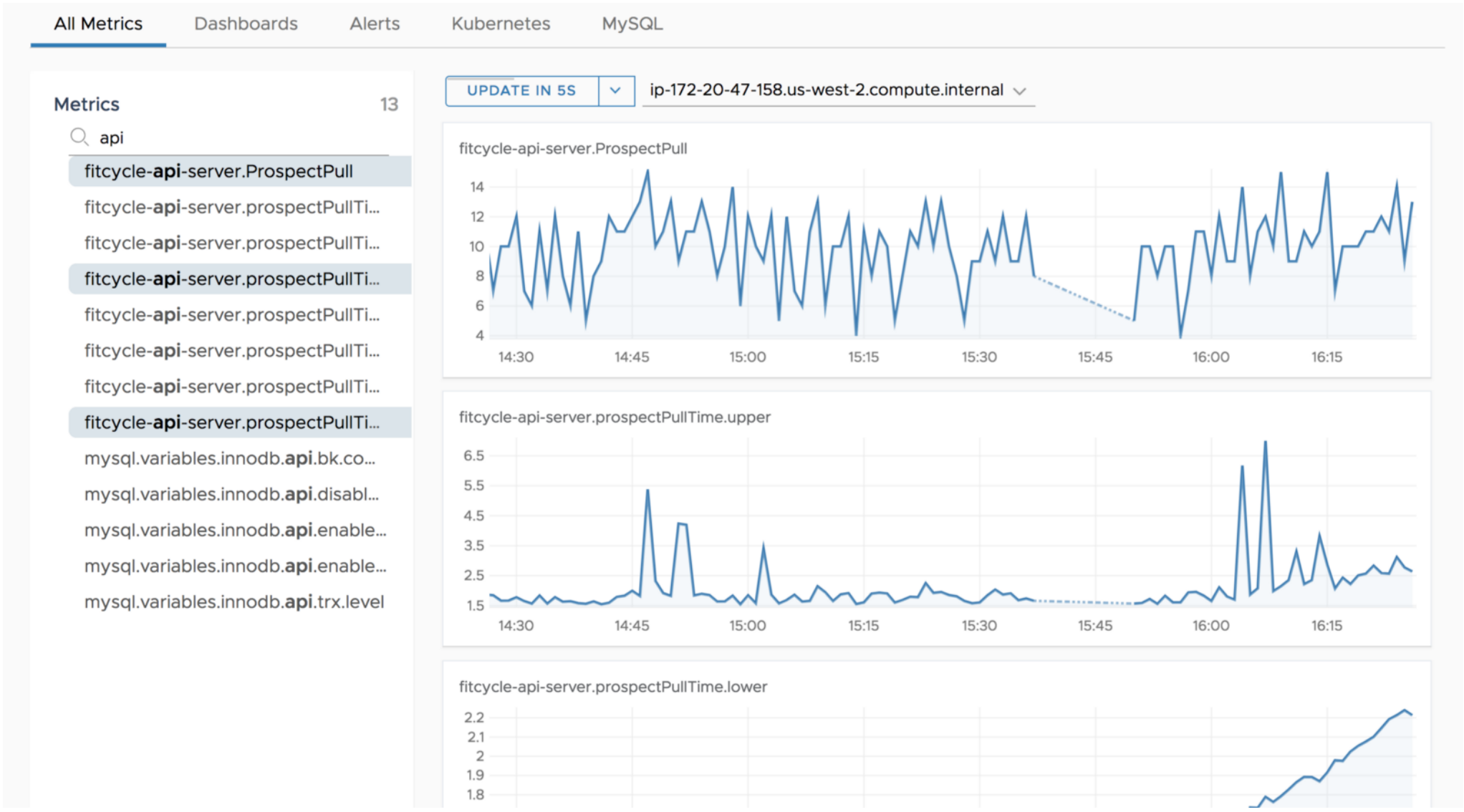
Application & cluster stats in Wavefront

Before walking through the detailed Telegraf setup with Wavefront, it's useful to see the end product. Since, Telegraf is collecting stats from flask, django, and mysql containers and sending them to Wavefront, the following graphs show the output in Wavefront. In addition, Wavefront also shows the cluster stats (node/pod/namespace stats).

Configuration and creation of the sidecars *and the configurations used is* detailed *in the next few sections.*

Application stats:

api-*server stats (*flask based*)*



(https://cloud.vmware.com/community/wp-content/uploads/9/sites/9/2018/08/1_f-6z_q07zc2O00wShNV9Wg.png)

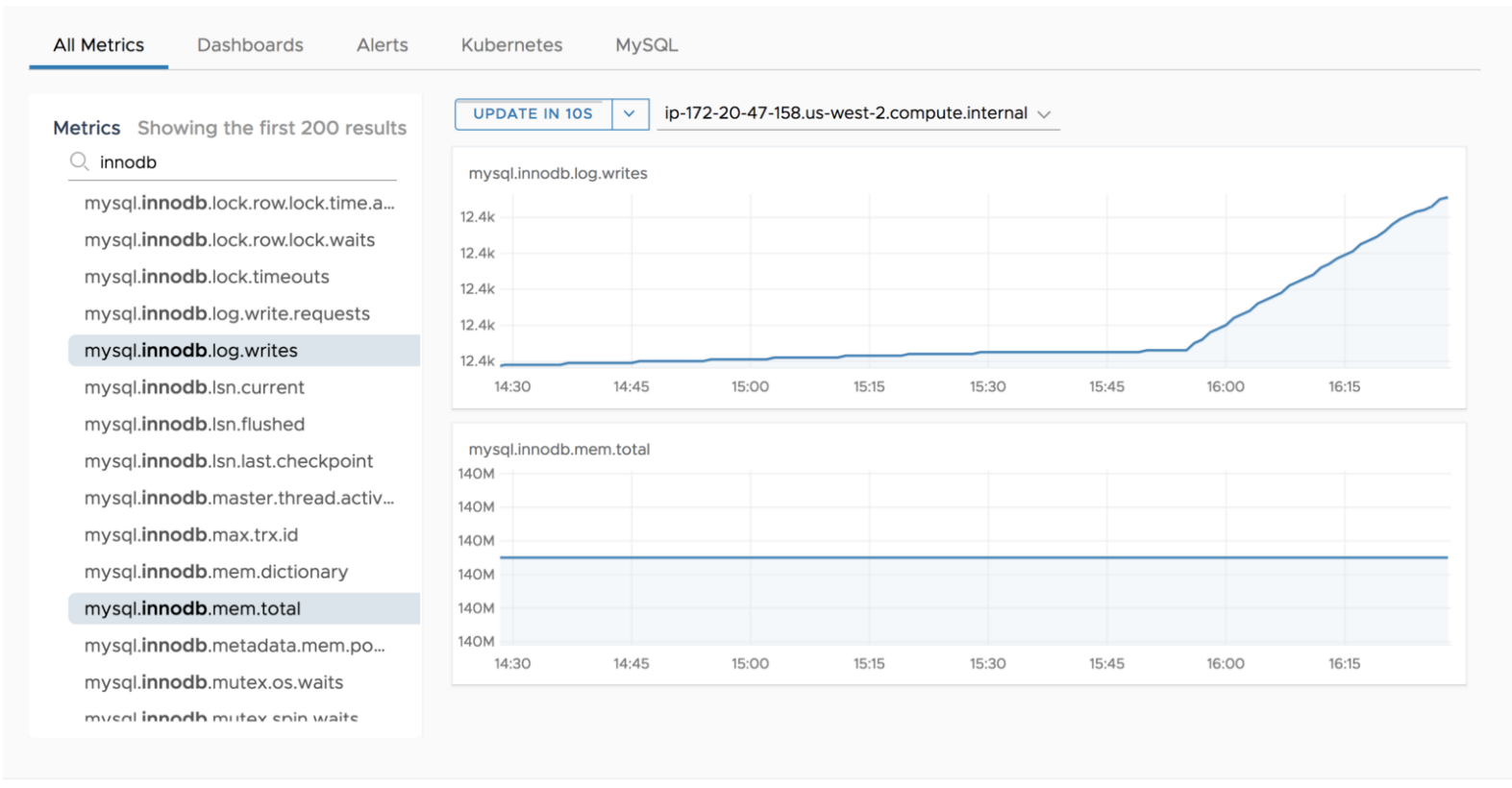
Stats detailed above are generally added by the developer in python for specific api calls in flask. The two stats on display are for a particular API call (get all signed up users):

“timer” per API call— several metrics such as Timer_stddev, Timer_mean, Timer_upper, etc are displayed per call.

Total number of times this call is made in any given period

The application outputs these stats via statsd (port 8125), which is collected by a telegraf sidecar collector in the same pod as the api-server. Again, I will detail this later in the blog.

mysql stats

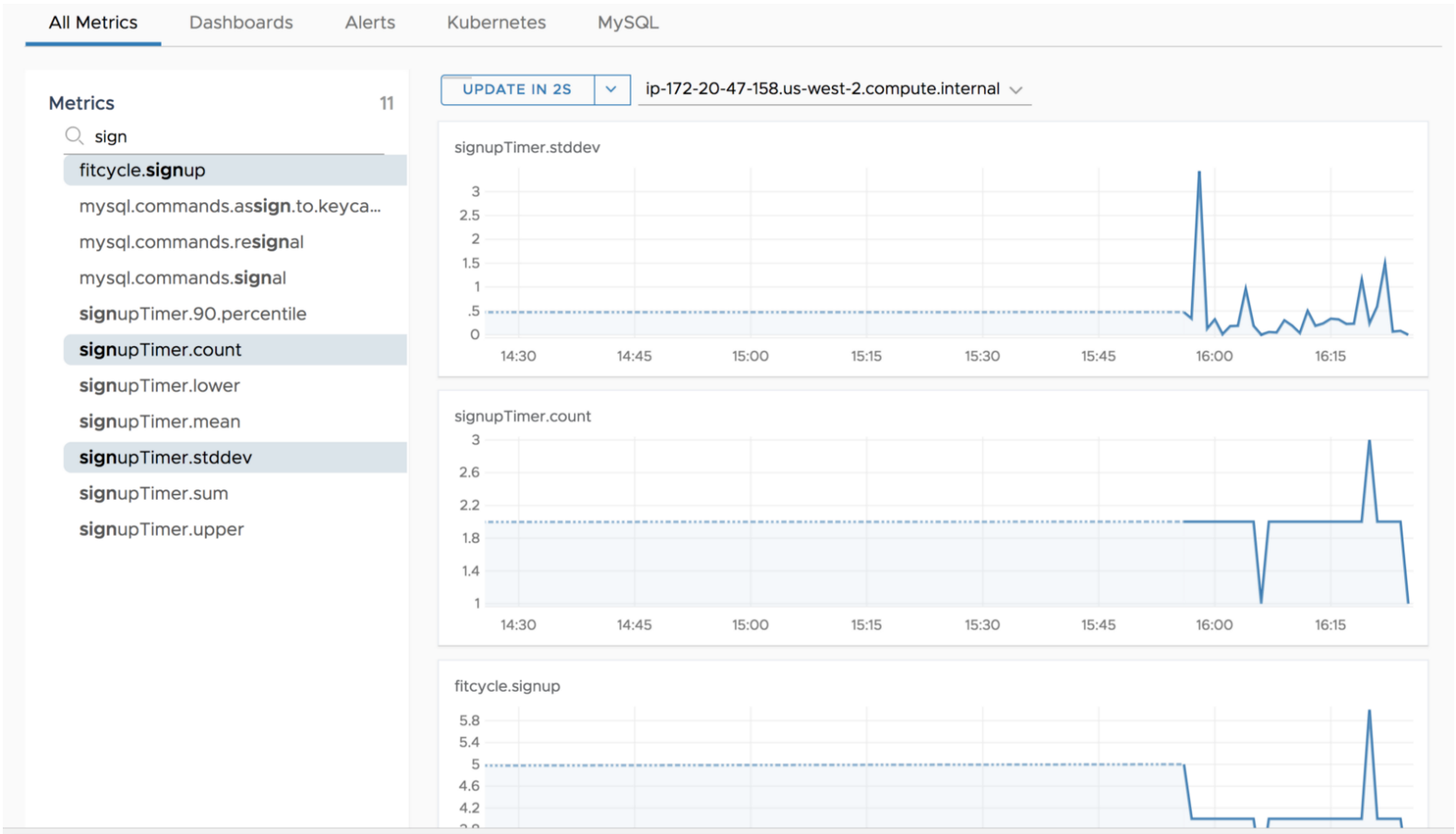


(https://cloud.vmware.com/community/wp-content/uploads/9/sites/9/2018/08/Image2.png) mysql stats

mysql stats are obtained via a pull from mysql directly. Approximately 200+ statsd can be pulled.

These stats are output via telegraf configured as a mysql collector.

web server stats (Django based)



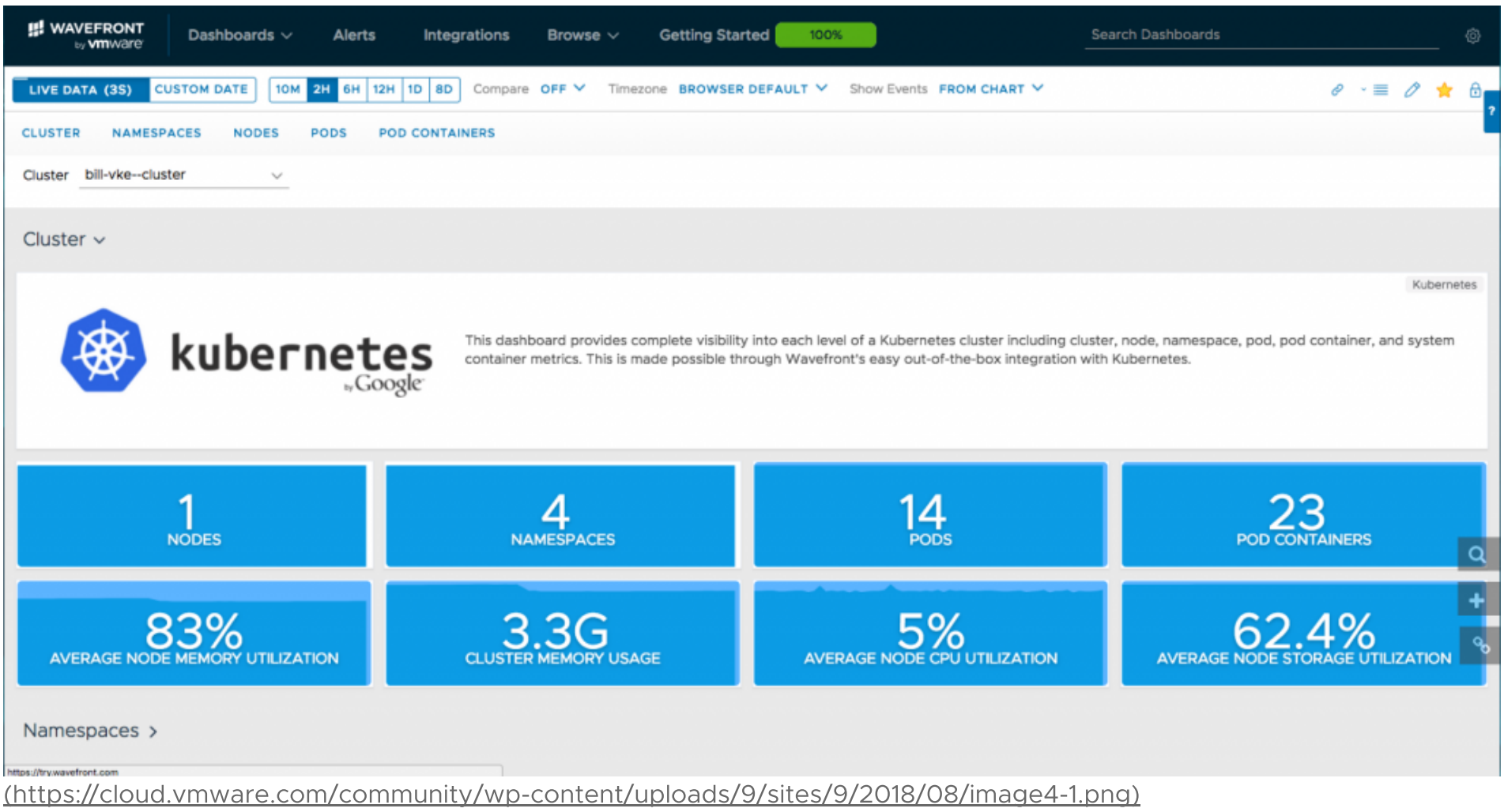
(https://cloud.vmware.com/community/wp-content/uploads/9/sites/9/2018/08/image3-1.png)

django based application stats

Stats detailed above are generally added by the developer in python for specific views in Django. In this case, a forms page is being measured. The two stats on display are for a particular “view” (form page):
“timer” calculating the “time” it takes to insert data into a database from the form page — its includes several metrics such as Timer_stddev, Timer_mean, Timer_upper, etc.
Total number of times the form is filled out.
The application outputs these stats via statsd (port 8125), which is collected by a telegraf sidecar collector in the same pod as the web-server. Again, I will detail this later in the blog.

Cluster Stats:

In addition to application stats, the entire set of cluster stats is also displayed. This is achieved using heapster, with output to Wavefront.
The following “cluster” stats are generally shown:
Namespaces level stats
Node level stats
Pod level stats
Pod container stats
The following set of charts show the standard Kubernetes dashboard in Wavefront.

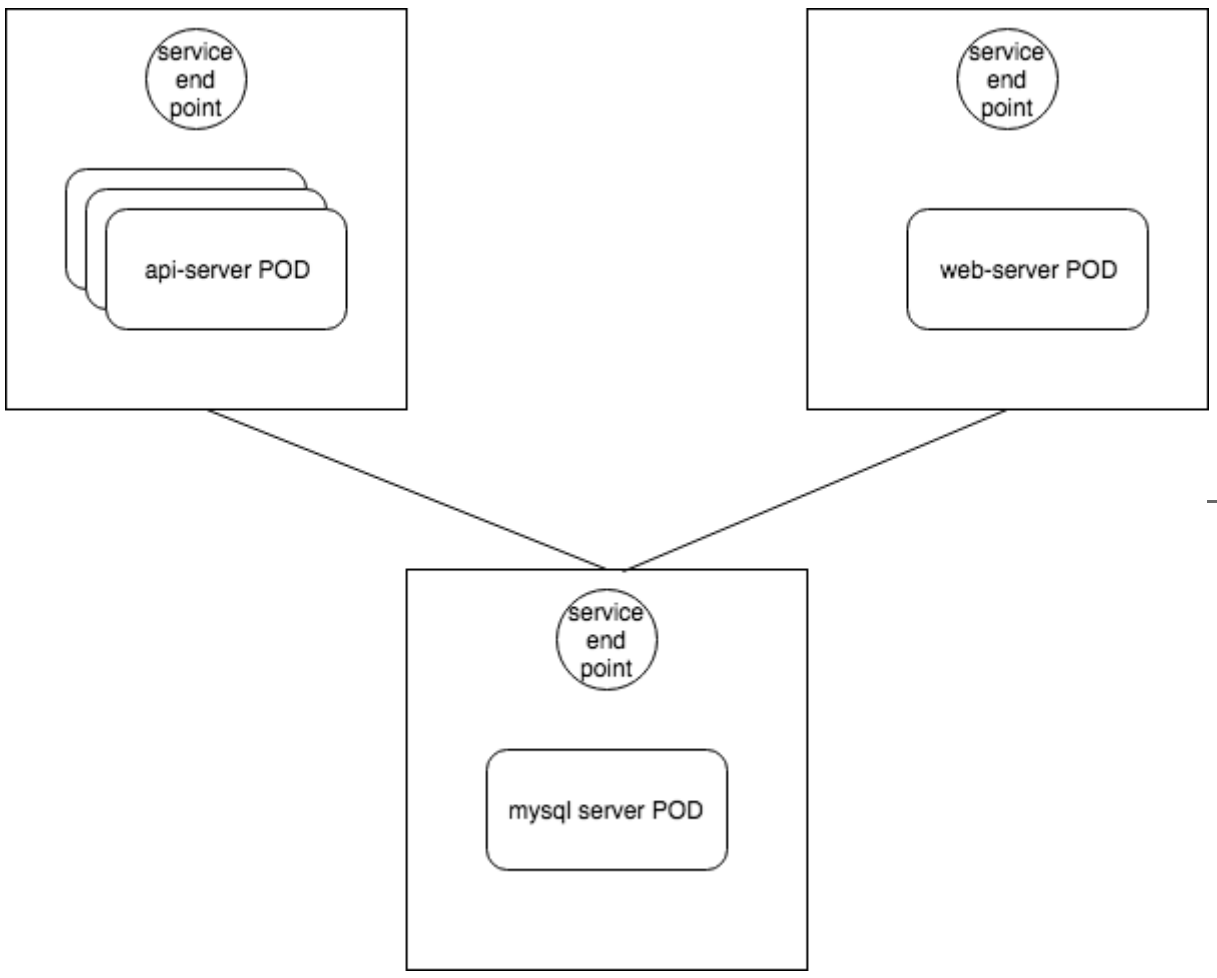


(https://cloud.vmware.com/community/wp-content/uploads/9/sites/9/2018/08/image4-1.png)

(https://cloud.vmware.com/community/wp-content/uploads/9/sites/9/2018/08/image7.png)Cluster stats

Sample Application (called Fitcycle)

In order to walk through the configuration, it's important to understand the application. I built an application with statsd output (stdout) for flask and Django and deployed it in kubernetes.
The sample app is called fitcycle and is located here (https://github.com/bshetti/container-fitcycle).
You can run this in any K8S platform (GKE, EKS, etc). I specifically ran it in VMware Kubernetes Engine (VKE). Once deployed, the following services are available:
Main webpage and form page for fitcycle is served by a Django server (supported by web-server PODs)
API is served by a Flask based server (api-server PODs) – it has multiple replicas
mysql server is served by the mysql POD
nginx ingress controller – which is preloaded by VMware Kubernetes Engine (not shown in the diagram below). Nginx ingress controller uses a URL based routing rule to load balance between the api-server and web-server



(https://cloud.vmware.com/community/wp-content/uploads/9/sites/9/2018/08/image5-2.png)
fitcycle application

The application outputs the following metrics:
api-server (flask), and the web-server (Django) output statsd to port 8125 in each pod (internally)
mysql collects metrics can be accessed logging in and polling for the right tables.
How do we collect and expose the stats?

Creating a statsd collector using telegraf

Telegraf has a wide variety of inputs/outputs. In deploying telegraf to collect the application stats for fitcycle, I created a statsd container with the following configuration:

statsd input plugin polling port 8125 against the main container in the pod for the api-server pod and web-server pod. FULL LIST of Telegraf inputs (https://github.com/influxdata/telegraf/tree/master/plugins/inputs)

wavefront output plugin to send the output to the Wavefront proxy service running in the cluster. FULL LIST of Telegraf outputs (https://github.com/influxdata/telegraf/tree/master/plugins/outputs)

(I'll write another blog about using telegraf with Prometheus)

Detailed repo for building the container is located here. (https://github.com/bshetti/telegraf-wavefront)

The container uses the *alpine* version of telegraf but changes the standard telegraf.conf file with the following:

telegraf.conf

```
# Global tags can be specified here in key="value" format.
 [global_tags]
 pod_name = "$POD_NAME"

# Configuration for telegraf agent
 [agent]
 ## Default data collection interval for all inputs
 interval = "$INTERVAL"
 ## Rounds collection interval to 'interval'
 ## ie, if interval="10s" then always collect on :00, :10, :20, etc.
 round_interval = true

## Telegraf will send metrics to outputs in batches of at
 ## most metric_batch_size metrics.
 metric_batch_size = 1000
 ## For failed writes, telegraf will cache metric_buffer_limit metrics for each
 ## output, and will flush this buffer on a successful write. Oldest metrics
 ## are dropped first when this buffer fills.
 metric_buffer_limit = 10000

## Collection jitter is used to jitter the collection by a random amount.
 ## Each plugin will sleep for a random time within jitter before collecting.
 ## This can be used to avoid many plugins querying things like sysfs at the
 ## same time, which can have a measurable effect on the system.
 collection_jitter = "0s"

## Default flushing interval for all outputs. You shouldn't set this below
 ## interval. Maximum flush_interval will be flush_interval + flush_jitter
 flush_interval = "$INTERVAL"
 ## Jitter the flush interval by a random amount. This is primarily to avoid
 ## large write spikes for users running a large number of telegraf instances.
 ## ie, a jitter of 5s and interval 10s means flushes will happen every 10-15s
 flush_jitter = "0s"

## By default, precision will be set to the same timestamp order as the
 ## collection interval, with the maximum being 1s.
 ## Precision will NOT be used for service inputs, such as logparser and statsd.
 ## Valid values are "Nns", "Nus" (or "Nµs"), "Nms", "Ns".
 precision = ""
 ## Run telegraf in debug mode
 debug = false
 ## Run telegraf in quiet mode
 quiet = false
 ## Override default hostname, if empty use os.Hostname()
 hostname = "$NODE_HOSTNAME"
 ## If set to true, do no set the "host" tag in the telegraf agent.
 omit_hostname = false

# Statsd Server
 [[inputs.statsd]]
 ## Protocol, must be "tcp", "udp4", "udp6" or "udp" (default=udp)
 protocol = "udp"

## MaxTCPConnection - applicable when protocol is set to tcp (default=250)
 max_tcp_connections = 250

## Enable TCP keep alive probes (default=false)
 tcp_keep_alive = false

## Specifies the keep-alive period for an active network connection.
 ## Only applies to TCP sockets and will be ignored if tcp_keep_alive is false.
 ## Defaults to the OS configuration.
```

```
 # tcp_keep_alive_period = "2h"

## Address and port to host UDP listener on
 service_address = ":8125"

## The following configuration options control when telegraf clears it's cache
 ## of previous values. If set to false, then telegraf will only clear it's
 ## cache when the daemon is restarted.
 ## Reset gauges every interval (default=true)
 delete_gauges = true
 ## Reset counters every interval (default=true)
 delete_counters = true
 ## Reset sets every interval (default=true)
 delete_sets = true
 ## Reset timings & histograms every interval (default=true)
 delete_timings = true

## Percentiles to calculate for timing & histogram stats
 percentiles = [90]

## separator to use between elements of a statsd metric
 metric_separator = "_"

## Parses tags in the datadog statsd format
 ## http://docs.datadoghq.com/guides/dogstatsd/ (http://docs.datadoghq.com/guides/dogstatsd/)
 parse_data_dog_tags = false

## Statsd data translation templates, more info can be read here:
 ## https://github.com/influxdata/telegraf/blob/master/docs/DATA_FORMATS_INPUT.md#graphite (https://github.com/influxda
 # templates = [
 # "cpu.* measurement*"
 # ]

## Number of UDP messages allowed to queue up, once filled,
 ## the statsd server will start dropping packets
 allowed_pending_messages = 10000

## Number of timing/histogram values to track per-measurement in the
 ## calculation of percentiles. Raising this limit increases the accuracy
 ## of percentiles but also increases the memory usage and cpu time.
 percentile_limit = 1000

# Specify optional tags to be applied to all metrics for this plugin
 # NOTE: Order matters, this needs to be at the end of the plugin definition
 # [[inputs.statsd.tags]]
 # tag1 = "foo"
 # tag2 = "bar"

# Configuration for Wavefront proxy to send metrics to
 [[outputs.wavefront]]
 host = "$WAVEFRONT_PROXY"
 port = 2878
 metric_separator = "."
 source_override = ["hostname", "nodename"]
 convert_paths = true
 use_regex = false
```

As noted in bold above two plugins are configured for telegraf:
input section—for statsd
output section—for wavefront (this can be replaced with prometheus)
There are several ENV variables in BOLD above that are important to note:

$POD_NAME—used to note the name of the pod if you want to particularly distinguish the pod (I will pass this in when using the container in Kubernetes as a sidecar)
$NODE_HOSTNAME—used to note the node where the pod is running (I will get this via a global spec variable from kubernetes when creating the sidecar container)
$INTERVAL—to note the collection interval time
$WAVEFRONT_PROXY—this is the kubernetes service name, DNS or IP of the wavefront proxy
This telegraf.conf is used in the Dockerfile to create the container
Dockerfile

```
# Telegraf agent configured for Wavefront output intended to be used in a sidecar config

FROM telegraf:alpine

ENV WAVEFRONT_PROXY="wavefront-proxy"
ENV INTERVAL="60s"

COPY telegraf.conf /etc/telegraf/telegraf.conf

CMD ["telegraf", "--config", "/etc/telegraf/telegraf.conf",
"--config-directory", "/etc/telegraf/telegraf.d"]
```

Now simply run:

```
docker build -t telegraf-statsd
```

And save the container to your favorite repo.
My version of the telgraf based statsd container is available via google registry.

```
gcr.io/learning-containers-187204/telegraf-statsd-sc
```

Kubernetes configuration using Telegraf-statsd container

Now that the statsd collector container is built and saved, I added it in a several kubernetes deployment yaml files. (api-server pod and the web-server pod)
I'll walk through the api-server (flask server) kubernetes deployment file showing how to configure the statsd collector as a side car. The Django and mysql configurations are similar, and details are found in my git repo.

```
https://github.com/bshetti/container-fitcycle/tree/master/wavefront (https://github.com/bshetti/container-fitcycle/tree
```

Here (https://github.com/bshetti/container-fitcycle/blob/master/wavefront/flask/api-server-deployment.yaml) is the deployment yaml for the api-server

```
apiVersion: apps/v1beta1 # for versions before 1.8.0 use apps/v1beta1
kind: Deployment
metadata:
 name: api-server
 labels:
 app: fitcycle
spec:
 selector:
 matchLabels:
 app: fitcycle
 tier: api
 strategy:
 type: Recreate
 replicas: 3
 template:
 metadata:
 labels:
 app: fitcycle
 tier: api
 spec:
 volumes:
- name: "fitcycle-apiserver-data"
emptyDir: {}
containers:
- image: gcr.io/learning-containers-187204/api-server-ml:latest
name: api-server
env:
- name: MYSQL_ID
value: "root"
- name: MYSQL_PASSWORD
valueFrom:
secretKeyRef:
name: mysql-pass
key: password
- name: MYSQL_SERVER
value: fitcycle-mysql
ports:
- containerPort: 5000
name: api-server
volumeMounts:
- mountPath: "/data"
name: "fitcycle-apiserver-data"
resources:
 resources:
 requests:
 memory: "64Mi"
 cpu: "100m"
 limits:
 memory: "256Mi"
 cpu: "500m"

- image: gcr.io/learning-containers-187204/telegraf-statsd-sc:latest
 name: telegraf-statsd
 ports:
 - name: udp-statsd
 containerPort: 8125
 protocol: UDP
 - name: udp-8092
 containerPort: 8092
 - name: tcp-8094
 containerPort: 8094
 env:
 - name: WAVEFRONT_PROXY
```

```
value: wavefront-proxy
- name: INTERVAL
value: 60s
- name: METRIC_SOURCE_NAME
# This can be change to use the Deployment / Statefulset name instead as a simple value
# The metric source name should not be an ephemeral value
valueFrom:
fieldRef:
fieldPath: spec.nodeName
- name: POD_NAME
valueFrom:
fieldRef:
fieldPath: metadata.name
- name: NAMESPACE
valueFrom:
fieldRef:
fieldPath: metadata.namespace
- name: NODE_HOSTNAME
valueFrom:
fieldRef:
fieldPath: spec.nodeName
resources:
requests:
memory: 30Mi
cpu: 100m
limits:
memory: 50Mi
cpu: 200m
```

Note the sections in bold. Key items to note in the configuration are:
Use the pre-built statsd collector container:

```
gcr.io/learning-containers-187204/telegraf-statsd-sc:latest
```

NODE_HOSTNAME variable uses a value from Kubernetes

```
- name: NODE_HOSTNAME
 valueFrom:
 fieldRef:
 fieldPath: spec.nodeName
```

spec.nodeName will return the node name this deployment is being deployed in.
Collection INTERVAL set to 60s for Wavefront
WAVEFRONT_PROXY is set to the service name of Wavefront proxy running in the kubernetes cluster. Installation Notes Here. (https://docs.wavefront.com/proxies_installing.html)
Enabling port 8125—which will listen to the output from the api-server
In order to run:

```
kubectl create -f api-server-deployment.yaml
```

Follow the instructions in the github repo for django and mysql configurations

---

Sample Application (Fitcycle) with Telegraf sidecars

Now that I have deployed the sidecars, we need to also
Deploy the Wavefront proxy (see instructions in the github repo)
Deploy the Wavefront heapster deployment
The application with sidecars now looks like follows: