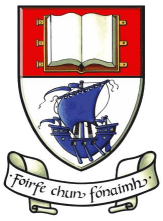

Waterford Institute of Technology

Cloud Infrastructures

Collecting Detailed Metrics from Docker Swarm Hosts

Stephen Coady

20064122

BSc in Applied Computing

April 12, 2017

Contents

1 Introduction

The aim of this research paper is to demonstrate how, using several different technologies, a swarm host can be dynamically polled and the collected metrics displayed graphically. It will show this method of data collection can be turned into a template so that it can be re-used for each new host which is added to the swarm.

It will also show how this process can automated so that no action is required from the administrator each time a new host is added to the swarm or a new service is created.

This paper will also examine the options available for real-time or live feedback on these hosts, for example an alternative to Amazon Web Service's CloudWatch. It will then compare these tools and processes to those available natively by using Docker. It will evaluate the strengths and weaknesses of both.

2 Problem Statement

As of July 2016 Docker usage had exploded in popularity, with some sources stating the container based technology's usage had risen by over 3000 percent in the two years since it left beta release (Arijs, 2016). With this in mind, it is natural to conclude that the amount of 'Dockerised' applications running in production has also grown at a fast rate.

Since an application running in production is assumed to be serving real customers or users downtime should be minimised and the performance of the application should be maximised (i.e. response time, latency, caching). The first part of this is taken care of by Docker Swarm, which provides scaling of applications across multiple hosts. The second, however, is not an easy problem to solve. Since Docker containers are supposed to be ephemeral it is difficult to monitor them as their short lifespan is not conducive to gathering longterm meaningful statistics (Young, 2017). This problem is only exacerbated by the fact that Docker Swarm may create multiple versions of the same container running across several different hosts.

While the Docker remote API does provide some metrics from containers natively it is less than ideal to need to keep track of each individual host and then to poll these hosts for metrics relating to each container on the host - especially since we may not know which service or indeed even how many services are running on each of our hosts. If we are then running our application in an elastic computing fashion such as Amazon's EC2 then these hosts may also be ephemeral and so the monitoring problem quickly becomes unmanageable.

We will now look at some of the technologies involved in the problem and the technologies we are proposing to use to solve the problem.

3 Technology Background

Since the aim of this paper is to implement a monitoring system which is not tied to any one vendor such as Amazon Web Services or Microsoft Azure it will aim to use only open source and community driven software to deliver this solution. This will mean that all findings should be easily reproducible and applicable to any cloud provider.

3.1 Containers

A container uses Linux abstraction techniques such as namespaces and cgroups to isolate and 'contain' a software process from those running on the same host. It does this limit resources available to that process while also making it much more manageable for the user. The container itself does not know that it is a container but instead is presented with a local resource and made to believe it has access to a global resource (Kerrisk, 2013). Containers and their workings are outside the scope of this paper, however it is important to understand that they are simply a process running on a host.

3.2 Docker

Docker is a piece of software which manages the orchestration of containers. It allows the user to tell Docker what they would like Docker to do and it leaves the underlying Linux translations to the Docker engine (Docker, 2016*b*). There are many alternatives such as Kubernetes and Amazon's Container Service, both of which can also manage Docker containers if the user wishes.

Docker receives its instructions from the user in the form of a 'Dockerfile' which is essentially a set of instructions specifying details such as the base image to use and what commands should be run on the container once it starts. This image is then built and can be used as a template for many containers to run from. It is this property which results in a single Docker host having multiple containers all based on the same image - much in the same way a Docker swarm host may contain many containers running a single service as previously discussed in Section 2.

3.3 Docker Swarm

Docker Swarm is a mode of Docker whereby a master/slave relationship is enforced across multiple internet-connected hosts with the express intent that the applications started on these hosts would be spread across all hosts in the *swarm* (Docker, 2016*a*). For example, in a swarm of say 5 hosts, the user could issue a command on the manager node to start a service which is made up of 10 containers. These 10 containers would then (more than likely) be spread across all 5 hosts with each running 2 containers. This means that the application now has 5 different routes by which it can be reached with a fault tolerance of an extra container on each host. While this example is trivial and not very practical it is intended to give a basic understanding of Docker swarm mode.

3.4 Grafana

Grafana is a graphical dashboard which can be used to display information from many different types of database. It allows very fine-grained control over the metrics displayed and also how they are displayed. It is a fully-functional web application which supports user login and different views depending on the logged in user (Grafana, 2017). It is not core to this project and there are many alternatives, however it was decided that Grafana was the best option as it has a large online community and support for a large number of third party software packages.

3.5 Prometheus

Prometheus is an open source and community driven monitoring solution which provides a multi-dimensional data model under which any metric can be stored efficiently. It also provides its own query language which can be used to extract information from these metrics (Prometheus, 2017).

Some of the major benefits of Prometheus over other metric storage solutions are:

- A package called 'Node Exporter' which is built by the same community has native compatibility with Prometheus and can be used to collect metrics from the host it is running on. This is extremely useful when monitoring a multi-host system such as a Docker Swarm.
- It has native support for Grafana.
- It also has native support for tools such as cAdvisor, which we will discuss further in Section 3.6.
- It has an active and large community.

3.6 cAdvisor

Container Advisor (cAdvisor) is a standalone application which provides detailed feedback about all containers running on a host. The metrics it provides include performance characteristics such as CPU usage, memory usage and network activity (Google, 2017).

cAdvisor has native support for many different types of containers, including Docker, so it is the ideal choice for this paper.

While it does provide a web app which can graphically display the information its real power lies in the fact that it also exposes a remote API which can be scraped for the metrics. We will take advantage of this fact in Section 4.

3.7 Alert Manager

Alert manager is a tool developed by Prometheus to monitor and create alerts based on metrics collected by Prometheus database (Prometheus, 2017). It can be used in a Docker container to push alerts via email, slack, hipchat and most other popular forms of instant communication. It aims to provide an industry grade alert mechanism as open source.

3.8 Infrastructure Overview

Now that we have seen each technology individually it is not difficult to see how each will fit together to provide a cohesive and modular application which will monitor both our Docker host and the containers which it is running. We can see a diagram giving a visual guide in Figure 1 below.

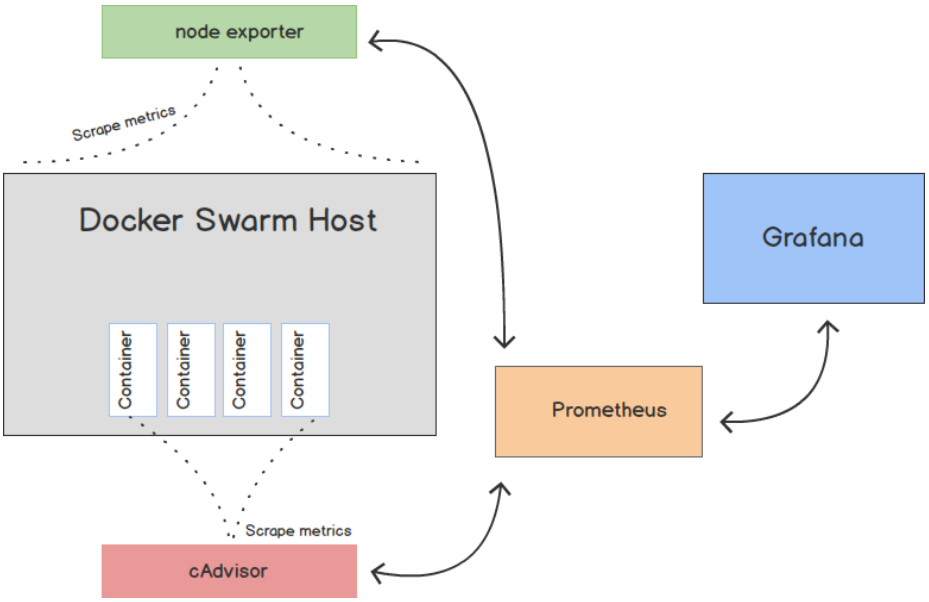

Figure 1: *Infrastructure Overview*

- The node exporter will read metrics based on the Docker Swarm host
- The cAdvisor component will read metrics based on the containers running on the host
- Both cAdvisor and the node exporter will be scraped by Prometheus which will in turn store these metrics for later usage
- Grafana will then be configured to periodically poll Prometheus and display the metrics visually.

3.9 Ansible

Ansible is a tool which can be used to provision and manipulate infrastructure such as AWS. By using modular roles to carry out a single task and then using these roles in a pre-defined order we can easily carry out what would typically be large and difficult tasks (Ansible, 2017). All infrastructure in this paper will be set up using Ansible and any configuration or modification will also be performed using Ansible.

4 Practical

4.1 Creating the Infrastructure

Since all infrastructure in this project will be created using Ansible we must first create the Docker Swarm which will be monitoring. While this is not the core objective of this research paper it is important to highlight how it is done as all tasks are automated. To do this a playbook which carries out the following tasks is required:

- Provision all necessary EC2 infrastructure.
- Install Docker.
- Create a Swarm and assign all necessary roles within the Swarm.
- Start our applications on the Swarm. These are purely for testing and perform no real logic.

By running this playbook we can easily have a large Docker Swarm running ready to be monitored. We will now explore in detail the process of starting the monitoring applications we require.

4.2 Starting the Monitoring Applications

The playbook discussed in the previous section is also responsible for starting the monitoring applications on each Swarm host. As discussed in Section 3 the Prometheus exporter and cAdvisor Docker containers will be started on each host. An overview of what each Swarm host will resemble once this playbook has been run can be seen in Figure 2.

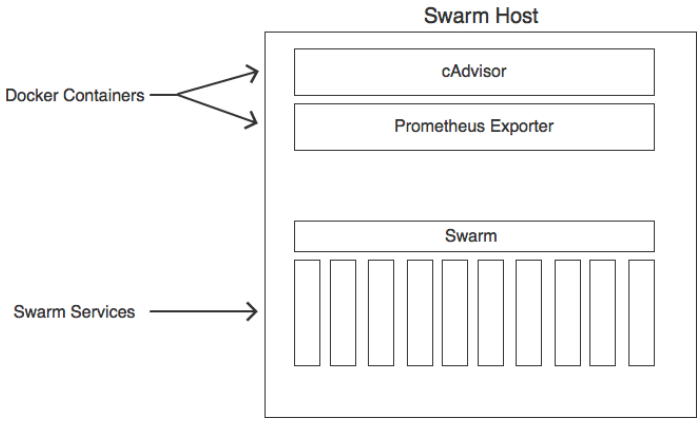

Figure 2: *Swarm Host Overview*

To start both of these applications on each Swarm host the playbook performs the following steps:

Git Clone

The playbook first clones the git repository available in Appendix A to each host. Storing the required code on Github is an easy way to share the code to each instance as required instead of using Ansible to push the code. This Ansible task can be seen below in Figure 3.

```
- hosts: managers:nodes
  become: true
  tasks:
    - name: clone git repository
      shell: >
        git clone https://github.com/StephenCoady/docker-statistics.git
```

Figure 3: *Using Ansible to Git Clone*

Starting the Applications

Using Docker compose we can easily start multiple Docker containers at once. To do this, we use the docker-compose.yml file found in the repository previously mentioned. This Docker compose file can be seen below in Figure 4.

```
1   version: '2'
2
3   services:
4
5     nodeexporter:
6       image: prom/node-exporter
7       container_name: nodeexporter
8       restart: unless-stopped
9       ports:
10        - 9100:9100
11      network_mode: host
12      labels:
13        org.label-schema.group: "monitoring"
14
15    cadvisor:
16      image: google/cadvisor:v0.24.1
17      container_name: cadvisor
18      volumes:
19        - /:/rootfs:ro
20        - /var/run:/var/run:rw
21        - /sys:/sys:ro
22        - /var/lib/docker/:/var/lib/docker:ro
23      restart: unless-stopped
24      ports:
25        - 8080:8080
26      network_mode: host
27      labels:
28        org.label-schema.group: "monitoring"
```

Figure 4: *Creating Exporters using Docker Compose*

We can see in Figure 4 above that both containers will then be started on each host and will expose their services over their respective default ports.

Now that each host has been provisioned and has the monitoring applications running we can now use these applications and display their metrics in a central location.

4.3 Creating the Monitoring Hub

We can now use another Ansible playbook which will create our monitoring hub and automatically configure it. To do this the playbook first follows the same steps as the previous playbook, by provisioning the instance which will be used.

It should be noted that in a production system this monitoring hub would be distributed and not

hosted on one AWS instance. This could be easily achieved by using different AWS availability zones.

This playbook first creates the instance and then installs Docker. Docker will again be used here to run the various containers needed for the monitoring hub. It then also clones the git repository from Appendix A.

Dynamically Scraping the Swarm Hosts

Once the instance has been created we can begin to put configuration in place for the containers which will eventually be run on it. The first of these is the prometheus config file. We will use a feature of Ansible called templates to dynamically populate this file with the endpoints which will need to be scraped for data.

We can see in Figure 5 below what this template file looks like before the dynamic hosts have been added.

```
scrape_configs:
  - job_name: 'nodeexporter'
    scrape_interval: 5s
    static_configs:
      - targets:
{% for host in groups['worker']%}
        - '{{ hostvars[host].inventory_hostname }}:9100'
{% endfor %}
{% for host in groups['master']%}
        - '{{ hostvars[host].inventory_hostname }}:9100'
{% endfor %}
```

Figure 5: *Example Prometheus Configuration*

Once all hosts have been added the file then resembles Figure 6. Each of these target endpoints refers to one single Swarm Host.

```
scrape_configs:
  - job_name: 'nodeexporter'
    scrape_interval: 5s
    static_configs:
      - targets:
        - '52.34.124.85:9100'
        - '52.34.85.124:9100'
        - '34.1.84.23:9100'
```

Figure 6: *Final Prometheus Configuration*

A similar process is followed for the cAdvisor scraping which will need to be performed - the only difference is that the port scraped will be 8080.

Updating the Hub's Config

Once all templates have been dynamically populated with hosts they are then copied to the monitoring hub instance by Ansible. This task can be seen below in Figure 7.

```
- hosts: monitors
  become: true
  tasks:
    - name: Update Prometheus config
      template:
        src=~/dev/docker-statistics/ansible/templates/prometheus.yml
        dest=docker-statistics/prometheus/prometheus.yml
        backup=yes
```

Figure 7: *Ansible Copying Prometheus Config to the Hub*

Starting the Monitoring Hub

Now that Prometheus has been given the endpoints to scrape we can start the 3 necessary containers which will combine to provide a full monitoring solution. To recap, these containers are:

Grafana This will be used to display the information in a web application. Contains full user login functionality and multiple plugins. Natively compatible with Prometheus.

Prometheus The database which will store all collected metrics. As soon as the Prometheus container is started it starts collecting metrics from the endpoints discussed in the previous section.

AlertManager Will be used to push alerts upon certain events triggering. Has native support for Prometheus and so can be used to monitor any metrics collected by Prometheus.

All of these containers are started by Ansible on the Monitoring Hub using the Docker compose file shown below in Figure 8.

```
grafana:
  image: grafana/grafana
  container_name: grafana
  volumes:
    - grafana_data:/var/lib/grafana
  env_file:
    - user.config
  restart: unless-stopped
  expose:
    - 3000
  ports:
    - 3000:3000
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

```
prometheus:
  image: prom/prometheus
  container_name: prometheus
  volumes:
    - ./prometheus/:/etc/prometheus/
    - prometheus_data:/prometheus
  command:
    - '-config.file=/etc/prometheus/prometheus.yml'
    - '-storage.local.path=/prometheus'
    - '-alertmanager.url=http://alertmanager:9093'
    - '-storage.local.memory-chunks=100000'
  restart: unless-stopped
  expose:
    - 9090
  ports:
    - 9090:9090
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

```
alertmanager:
  image: prom/alertmanager
  container_name: alertmanager
  volumes:
    - ./alertmanager/:/etc/alertmanager/
  command:
    - '-config.file=/etc/alertmanager/config.yml'
    - '-storage.path=/alertmanager'
  restart: unless-stopped
  expose:
    - 9093
  ports:
    - 9093:9093
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

Figure 8: *Docker Compose File*

4.4 Configuring the Monitoring Hub

Now that we have started all monitoring applications on the monitoring hub we can log into Grafana and begin setting up the dashboards. This is a one time process where we are essentially telling Grafana about the local Prometheus Docker container and where it can be accessed.

Once the Grafana container is running we can browse to the instance's address as seen below in Figure 9.

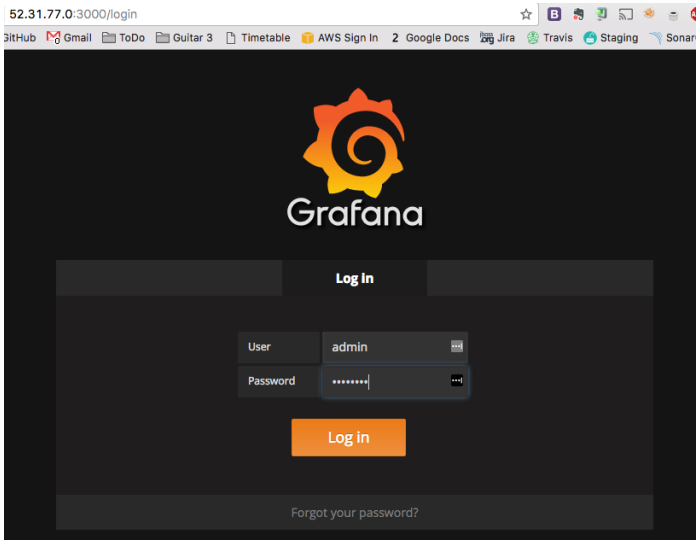

Figure 9: *Grafana Login Page*

The next step is informing Grafana where it can find a datasource. This is easily done by using Grafana's quick start menu. This can be seen below in Figure 10.

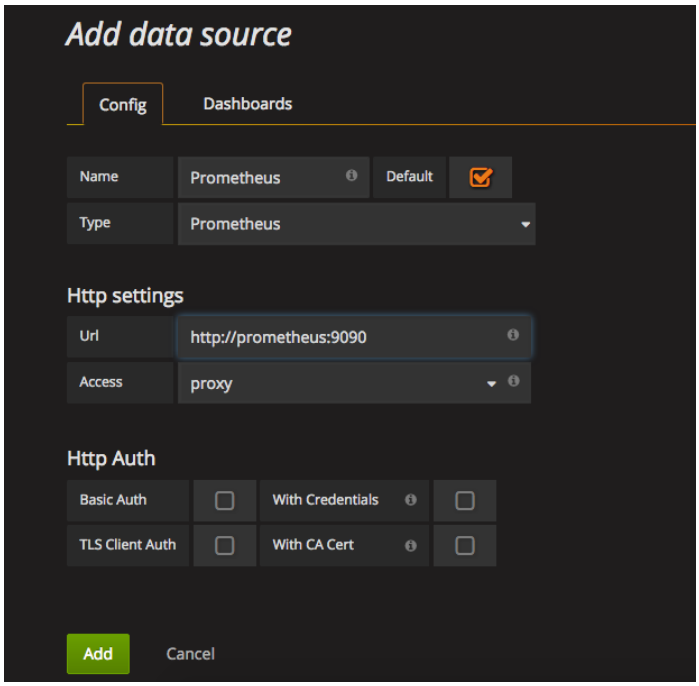

Figure 10: *Pointing Grafana at Prometheus*

Notice how we use the Docker networking name for the prometheus container here. This could easily

be localhost or the instance address but using the Docker network enforces uniformity.

After completing this step Grafana now has access to all metrics stored by Prometheus. Since Prometheus was started with a config which pointed to all instances in the Docker Swarm it is collecting metrics in the background, even if we are not currently displaying them. To display the metrics we must now create Grafana dashboards. This is a JSON file which defines all metric expressions to be displayed and also the format in which they will be displayed. A snippet of one of these files can be seen below in Figure 11. This particular snippet creates a machine cpu cores box on the dashboard which is created from the sum of all machine cores stored in Prometheus.

```
      "targets": [
        {
          "expr": "sum(machine_cpu_cores)",
          "interval": "",
          "intervalFactor": 2,
          "legendFormat": "",
          "metric": "machine_cpu_cores",
          "refId": "A",
          "step": 20
        }
      ],
      "thresholds": "",
      "title": "CPU Cores",
      "type": "singlestat",
      "valueFontSize": "80%",
      "valueMaps": [
        {
          "op": "=",
          "text": "N/A",
          "value": "null"
        }
      ],
      "valueName": "avg"
    },
```

Figure 11: *JSON Template for a Dashboard*

Once all relevant dashboards have been created using separate JSON templates we can browse to Grafana's dashboards and view the metrics we have decided to display. In Figure 12 we can see metrics which relate to all hosts in the Docker Swarm.

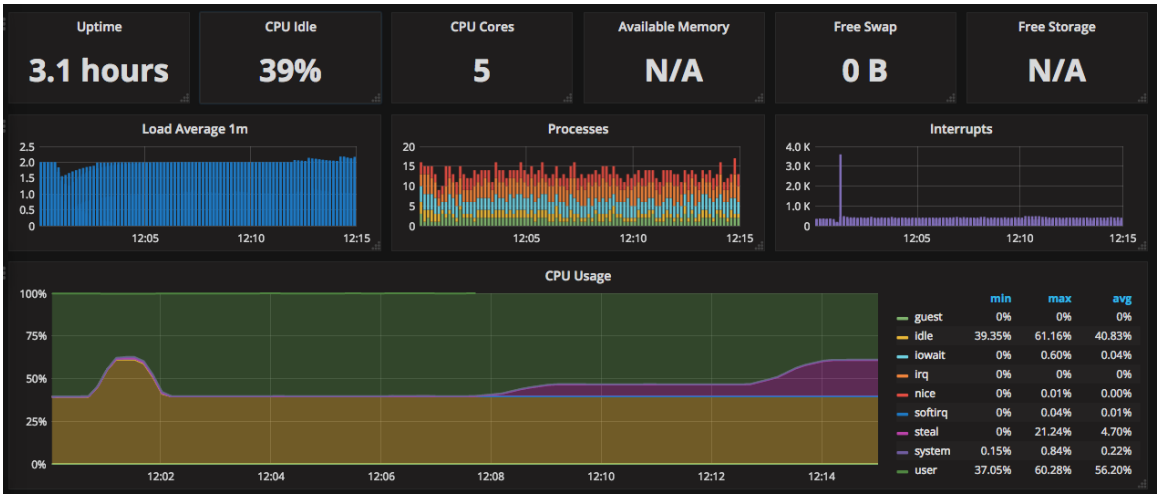

Figure 12: *Docker Swarm Host Monitoring*

We can also view metrics collected on each Docker container running in our Swarm. This can be seen in Figure 13.

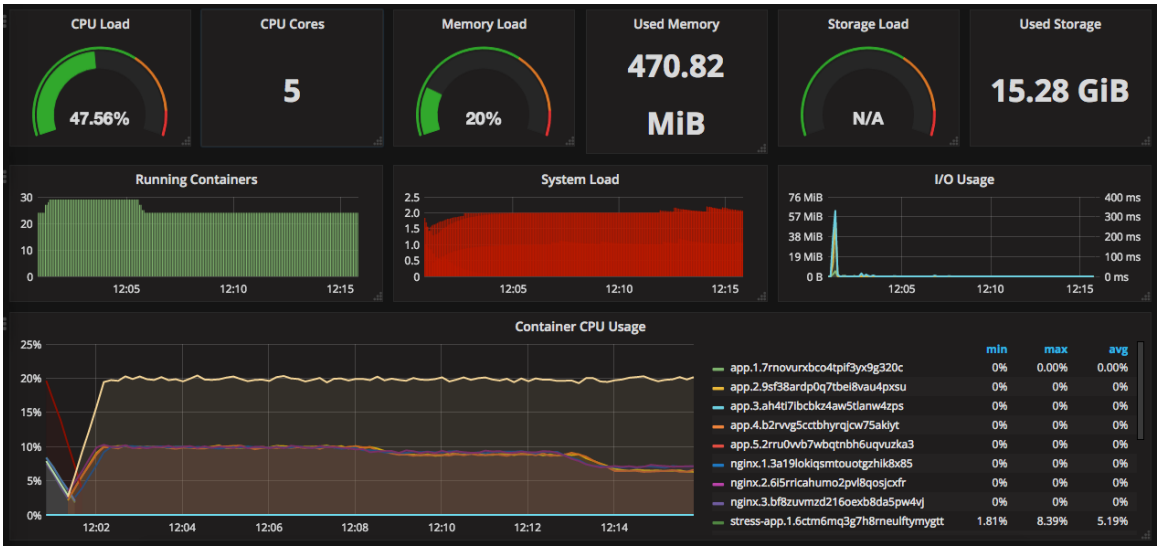

Figure 13: *Docker Swarm Container Monitoring*

If it was required we could also create metrics on each host in the swarm individually. However for the purposes of this report it is sufficient to monitor the Swarm as a whole.

4.5 Creating Alerts

Now that we are monitoring metrics from each host in the Swarm we can use Alert Manager to create alerts based on these metrics. Some sample alerts created for this report are:

Container Down

In Figure 14 we can see that if the container with the 'visualiser' is down for more than ten seconds then an alert will be triggered.

```
ALERT visualiser_down
  IF absent(container_cpu_user_seconds_total{name="visualiser"})
  FOR 10s
  LABELS { severity = "critical" }
  ANNOTATIONS {
    summary= "Visualiser down",
    description= "Visualiser container is down for more than 10 seconds."
  }
```

Figure 14: *Creating Container Alerts*

Host High Cpu Usage

We can see in Figure 15 that if the combined CPU rate of all hosts is above 30 then an alert is triggered.

Alert Manager can then be configured to use many different communication channels to send these alerts. For the purposes of this report Slack was chosen. To set this up the following config file is

```
ALERT high_cpu_load
  IF sum(rate(node_cpu[1m])) by (mode) * 100 / count_scalar(node_cpu{mode="user"}) > 30
  FOR 30s
  LABELS { severity = "warning" }
  ANNOTATIONS {
      summary = "Server under high load",
      description = "Docker host is under high load, the avg load 1m is at {{ $value}}. Reported by
      instance {{ $labels.instance }} of job {{ $labels.job }}.",
  }
```

Figure 15: *Creating Alerts Related to Swarm Hosts*

loaded in Alert Manager.

```
route:
    receiver: 'slack'

receivers:
    - name: 'slack'
      slack_configs:
          - send_resolved: true
            text: "{{ .CommonAnnotations.description }}"
            username: 'Prometheus'
            channel: '@scoady'
            api_url: 'https://hooks.slack.com/services/T2JQPS52A/
```

Figure 16: *Configuring Alert Manager to use Slack*

Which can then be used to trigger alerts in Slack like those seen in Figure 17.

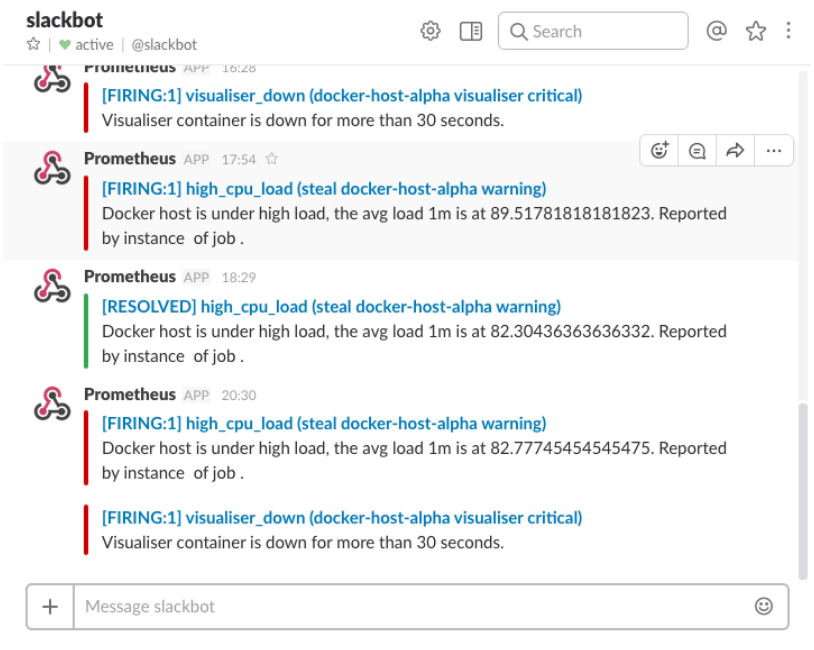

Figure 17: *Slack Alerts*

5 Discussion

5.1 Problems Encountered

Although the monitoring hub was set up successfully and provided a complete view of the Docker Swarm there are several problems with the technology stack. Some of the technologies discussed did not work as one would reasonably expect. While this was not a huge problem for this paper it might make them unsuitable for large enterprise applications. Some examples are:

5.1.1 Authentication

While Grafana supports authentication the same cannot be said for Prometheus or Alert Manager. This means that the services expose themselves by default to the outside world. While this might be a problem easily solved by using reverse proxies or private networks it is not ideal to have a service with no native support for security. This alone may be enough to stop some businesses using these services.

cAdvisor and the Prometheus exporter modules do not support authentication meaning their end-points are also open to the public network. Again this could be mitigated but it may be enough to mean the applications are not yet suitable for production.

5.1.2 Complexity

While Grafana is an extremely powerful tool it also has a steep learning curve. The interface is designed to be accessible by using JSON to configure dashboards but this can be off putting when paired with large data sets. For example, while running this project cAdvisor and the Prometheus exporter modules both created over 150 different metrics in the Prometheus database. While this was a non-trivial amount it is still small compared to what a large enterprise-level application would produce.

5.1.3 Alerts Reliability

While testing the Alert Manager container it was found that alerts did not always fire as expected. After some investigation some possible causes were identified:

Instance Size
Since this project used Amazon's t2.micro instances the memory available on each instance was relatively low. This was necessary as the ansible playbook was often tested by creating upwards of twelve instances at a time. After some investigation of processes running on each instance it was found that running any applications which required high CPU usage alongside the monitoring stack caused unexpected peaks and troughs in usage in general. It is still unclear as to whether this was caused by using low-powered instances or not.

Alert Manager Configuration
During the course of this project it was found that Alert manager can be temperamental during configuration. Any small mistake in the file would result in absolutely no alerts at all being fired. This would make monitoring large systems with huge configuration files extremely difficult.

5.2 Alternatives

While there are several alternative solutions to those proposed in this report there are some differences.

Price
Options like using AWS ECS to orchestrate Docker containers and CloudWatch to create alerts can be costly. If a large application needs to be dispersed across a large infrastructure the cost can quickly add up.

Flexibility
Using proprietary software such as AWS means that the developer/system admin does not have the same level of control over the services they run. For example by creating our own infrastructure we have made it easy to swap any component for an equivalent one. If for instance we needed to remove Prometheus and use a different Database we can do so easily. If we used a large provider like AWS we would have to use whatever metric collection solution they provide with CloudWatch.

Open Source
One of the biggest advantages of using the proposed stack is that everything is open source. If this solution was being used in a large company which needed to change any of the products to suit their own needs then they could do this.

6 Conclusion

6.1 Further Improvements

While we have shown a high level monitoring solution created from scratch using open source software there are several improvements which would bring great value to the project as a whole.

6.1.1 Authentication

As mentioned in section 5 the introduction of authentication at all endpoints increase the value of the project.

6.1.2 Network Monitoring

Perhaps by using another third party tool such as Conntrack to monitor all network interactions of the system (Conntrack, 2017). This would provide an even more detailed overview of the infrastructure and the applications running on it.

6.1.3 Granular Monitoring

Currently the monitoring hub displays metrics for the infrastructure as a whole and also high level metrics for each container running on this infrastructure. To improve this further the dashboard templates could be manipulated further to include the ability to monitor a specific host, container or group of containers. This would allow the system administrator to manage containers based on the application they are running. For instance if a Swarm is running several different applications across dozens of different services then it would make sense to be able to monitor each application and its respective services individually.

6.2 Closing Statement

In summary we have seen how multiple open source tools can be used together to monitor a large application running in a Docker Swarm. We have seen how these tools can also be created and deployed using automation tools such as Ansible. With the current shift in developer responsibilities this automation places more power in the developer's hands while also decreasing the work required to achieve results.

We have shown how these tools can be used to provide alternatives to paid options. This means that monitoring no longer needs a large financial outlay.

Bibliography

Ansible (2017), 'Ansible', https://www.ansible.com. Accessed: 15-03-2017.

Arijs, P. (2016), 'Docker usage statistics: Increased adoption by enterprises and for production use', http://www.coscale.com/blog/docker-usage-statistics-increased-adoption-by-enterprises-and-for-production-use. Accessed: 23-03-2017.

Conntrack (2017), 'Connection tracking tools', http://conntrack-tools.netfilter.org/. Accessed: 12-04-2017.

Docker (2016*a*), 'Docker Swarm', https://docs.docker.com/swarm/overview/. Accessed: 03-10-2016.

Docker (2016*b*), 'What is docker?', https://www.docker.com/what-docker. Accessed: 03-10-2016.

Google (2017), 'cAdvisor', https://github.com/google/cadvisor. Accessed: 24-03-2016.

Grafana (2017), 'Grafana', https://grafana.com/. Accessed: 24-03-2016.

Kerrisk, M. (2013), 'Namespaces in operation, part 1: namespaces overview [LWN.net]', https://lwn.net/Articles/531114/. Accessed: 03-10-2016.

Prometheus (2017), 'Prometheus', https://prometheus.io/docs/introduction/overview/. Accessed: 24-03-2016.

Young, K. (2017), 'The docker monitoring problem', https://www.datadoghq.com/blog/the-docker-monitoring-problem/. Accessed: 24-03-2017.

Appendices

A Application Repository

https://github.com/StephenCoady/docker-statistics