Monitoring and Logging

- Prometheus
- Grafana
- Monitoring Java Applications
- Centralized Logging

What We will be covering

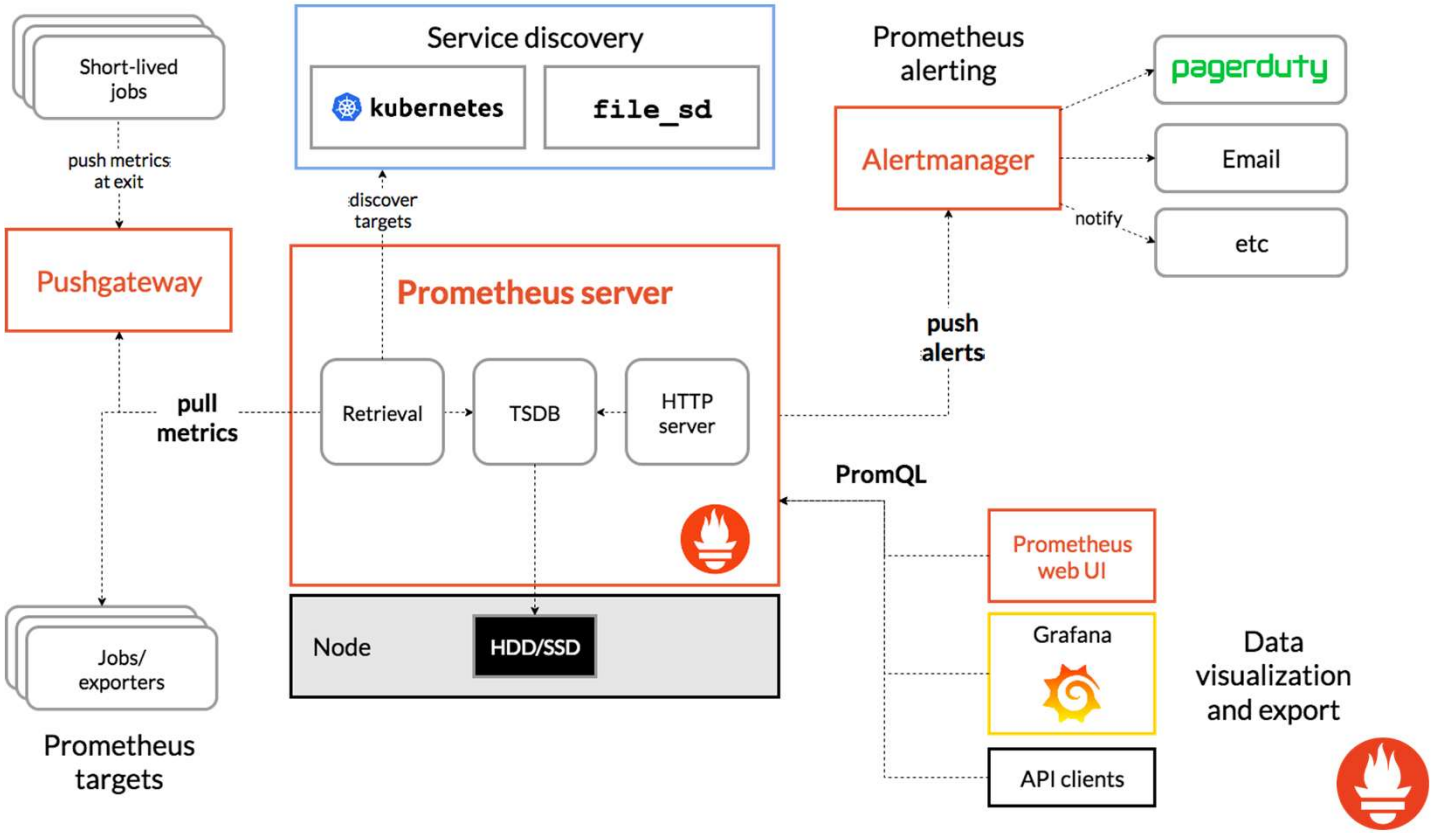
- Session 5 – Monitoring and Logging
  - Introduction to Grafana, Prometheus, and ELK stack
  - Centralized Logging
  - Metrics and Metrics Scrapping
  - Monitoring Kubernetes Pods with Prometheus/Grafana/ELK

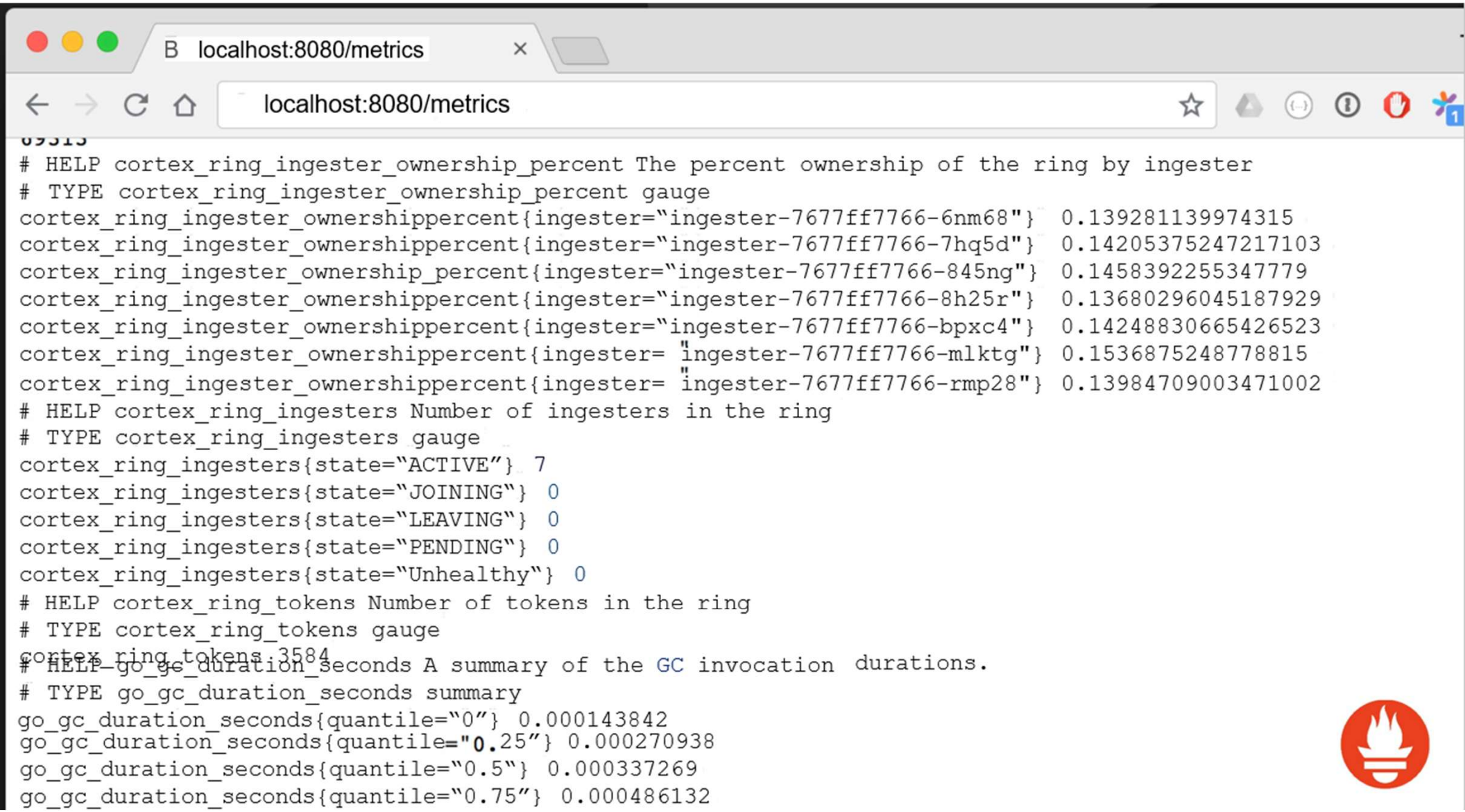
Prometheus

- A monitoring & alerting system.
- Inspired by Google's BorgMon
- Originally built by SoundCloud in 2012
- Open Source, now part of the CNCF

- Simple text-based metrics format
- Multidimensional datamodel
- Rich, concise query language

Prometheus




localhost:8080/metrics
localhost:8080/metrics
HELP cortex_ring_ingester_ownership_percent The percent ownership of the ring by ingester
TYPE cortex_ring_ingester_ownership_percent gauge
cortex_ring_ingester_ownershippercent{ingester="ingester-7677ff7766-6nm68"} 0.139281139974315
cortex_ring_ingester_ownershippercent{ingester="ingester-7677ff7766-7hq5d"} 0.14205375247217103
cortex_ring_ingester_ownership_percent{ingester="ingester-7677ff7766-845ng"} 0.1458392255347779
cortex_ring_ingester_ownershippercent{ingester="ingester-7677ff7766-8h25r"} 0.13680296045187929
cortex_ring_ingester_ownershippercent{ingester="ingester-7677ff7766-bpxc4"} 0.14248830665426523
cortex_ring_ingester_ownershippercent{ingester="ingester-7677ff7766-mlktg"} 0.1536875248778815
cortex_ring_ingester_ownershippercent{ingester="ingester-7677ff7766-rmp28"} 0.13984709003471002
HELP cortex_ring_ingesters Number of ingesters in the ring
TYPE cortex_ring_ingesters gauge
cortex_ring_ingesters{state="ACTIVE"} 7
cortex_ring_ingesters{state="JOINING"} 0
cortex_ring_ingesters{state="LEAVING"} 0
cortex_ring_ingesters{state="PENDING"} 0
cortex_ring_ingesters{state="Unhealthy"} 0
HELP cortex_ring_tokens Number of tokens in the ring
TYPE cortex_ring_tokens gauge
cortex_ring_tokens 3584
HELP go_gc_duration_seconds A summary of the GC invocation durations.
TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0.000143842
go_gc_duration_seconds{quantile="0.25"} 0.000270938
go_gc_duration_seconds{quantile="0.5"} 0.000337269
go_gc_duration_seconds{quantile="0.75"} 0.000486132

Prometheus

- Simple Data Model
  - <identifier> -> (t_0,v_0), (t_1, v_1), ... (t_n,v_n)
- Essentially a time series data
- Timestamps are in milliseconds
- Examples of these include: (See the series selectors below)

```
http_requests_total{job="nginx", instances="1.2.3.4:80", path="/home", status="200"}
http_requests_total{job="nginx", instances="1.2.3.4:80", path="/home", status="500"}
http_requests_total{job="nginx", instances="1.2.3.4:80", path="/settings", status="200"}
http_requests_total{job="nginx", instances="1.2.3.4:80", path="/settings", status="502"}
```

Prometheus

- Queries start from a selector (usually)
  - PromQL: `http_requests_total{job="nginx", status=~"5.."}`

```
{job="nginx", instances="1.2.3.4:80", path="/home", status="500"}        34
{job="nginx", instances="1.2.3.4:80", path="/settings", status="502"}    56
{job="nginx", instances="2.3.4.5:80", path="/home", status="500"}        76
{job="nginx", instances="2.3.4.5:80", path="/settings", status="502"}    96
...
```

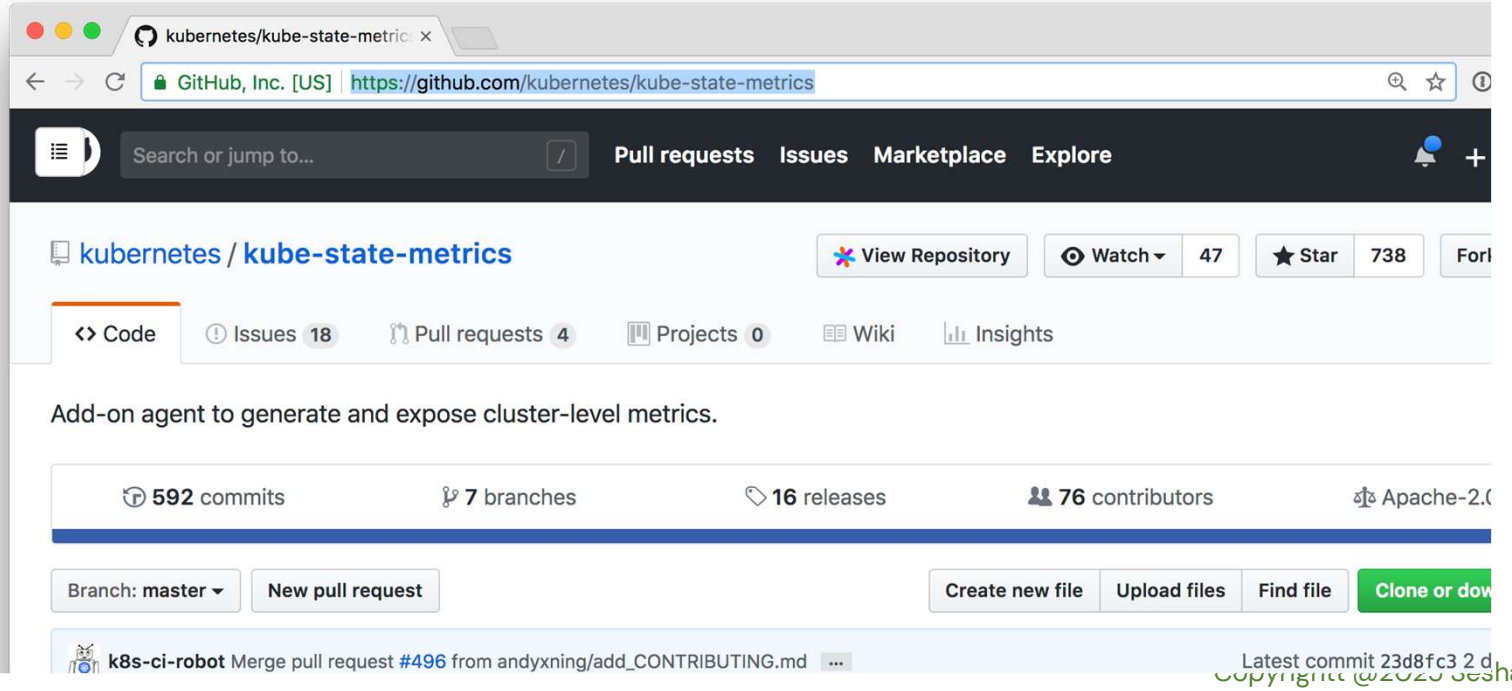
Prometheus

- PromQL has a number of features.
- It can select a vector of values, use functions and
- Aggregrate by dimension e.g.
- *sum by (path) (rate(http_requests_total{job="nginx",status =~ "5.."}[1m]))*
- And do binary operations e.g.
  - *sum by (path) (rate(http_requests_total{job="nginx",status =~ "5.."}[1m])) / sum by (path) (rate(http_requests_total{job="nginx"}[1m])*

- Full Reference – please refer to the documentation

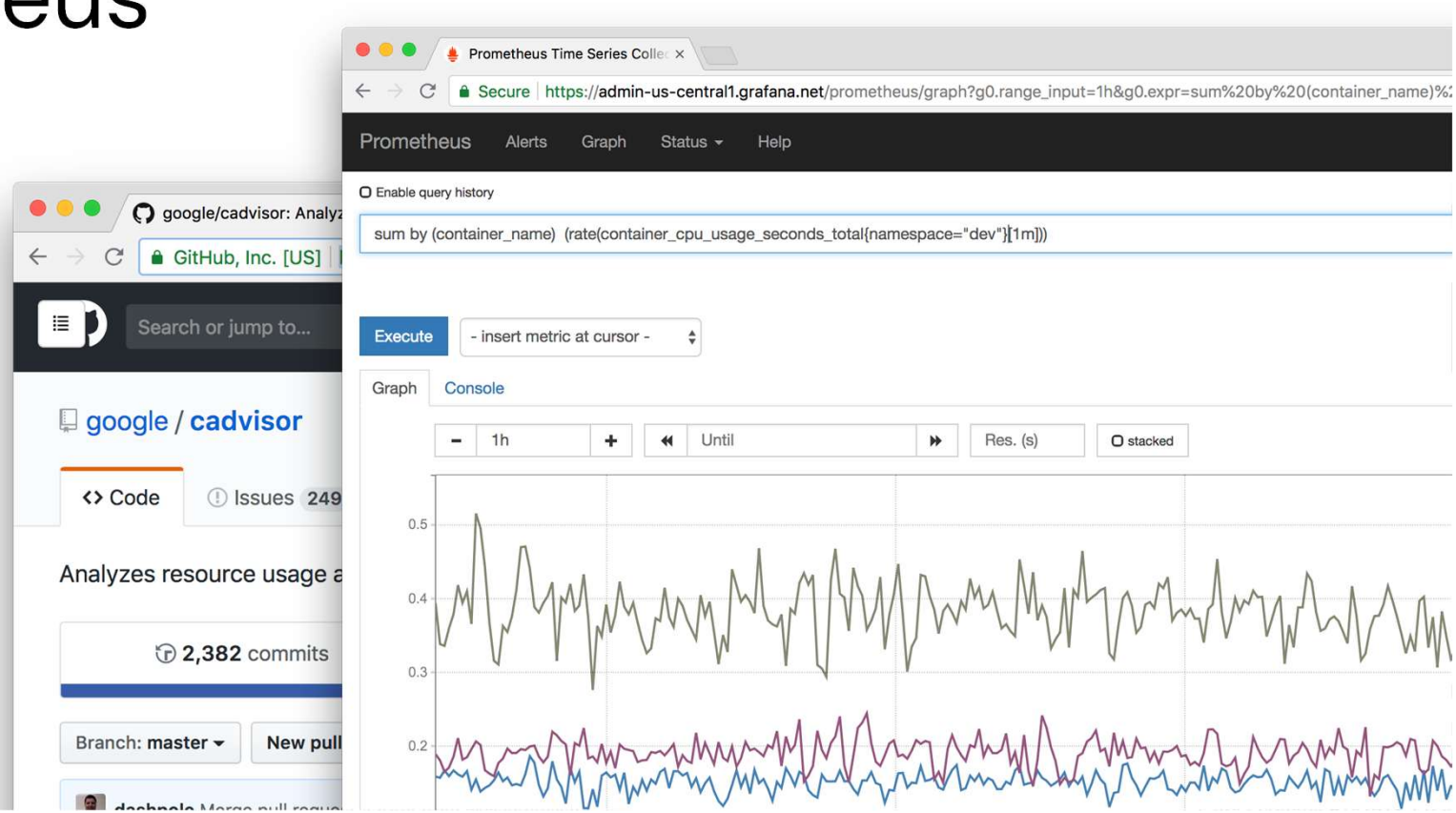
Prometheus

- The term metrics is often used.
- E.g. Refer to the site: https://github.com/kubernetes/kube-state-metrics



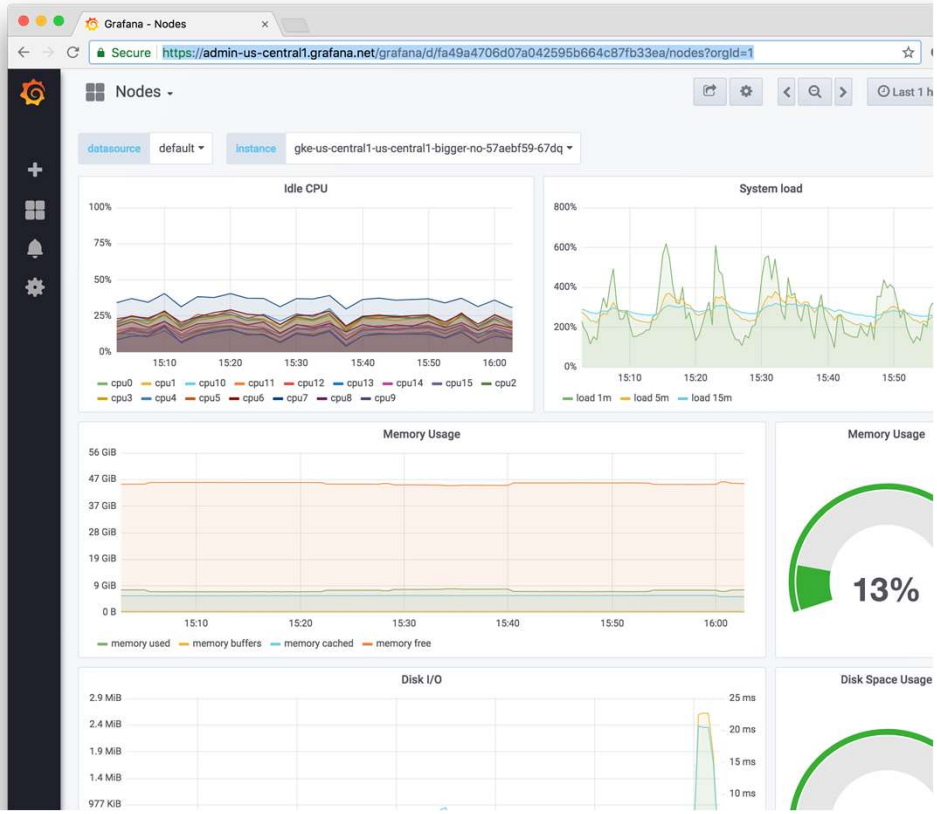
Prometheus

- In addition to the Prometheus UI, you can use the UI provided by Kubernetes state metrics or Cadvisor from Google



USE

- Cluster and Node Level Metrics
- node_exporter runs as a daemonset

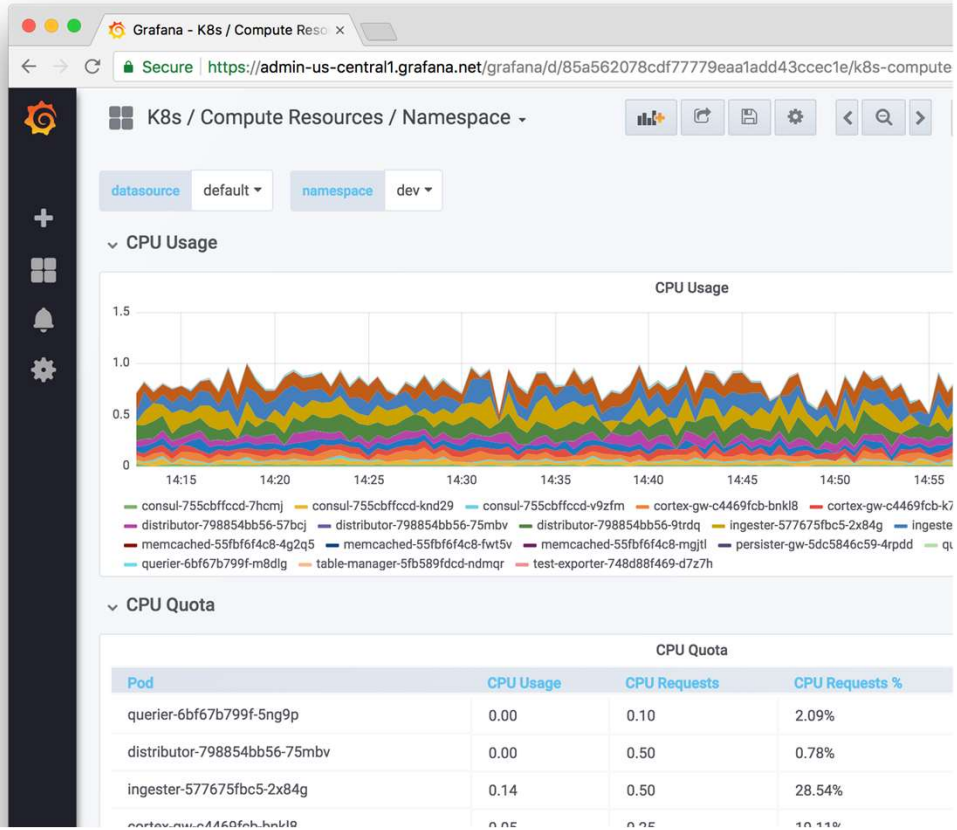


USE

- CPU Utilization
  - 1 – avg(rate(node_cpu{mode="idle"}[1m]))
- CPU Saturation
  - sum(node_load1) / (sum(node:node_num_cpu:sum)

USE Method

- cAdvisor also provides container level metrics
- That can be combined with the metrics from kube-state-metrics



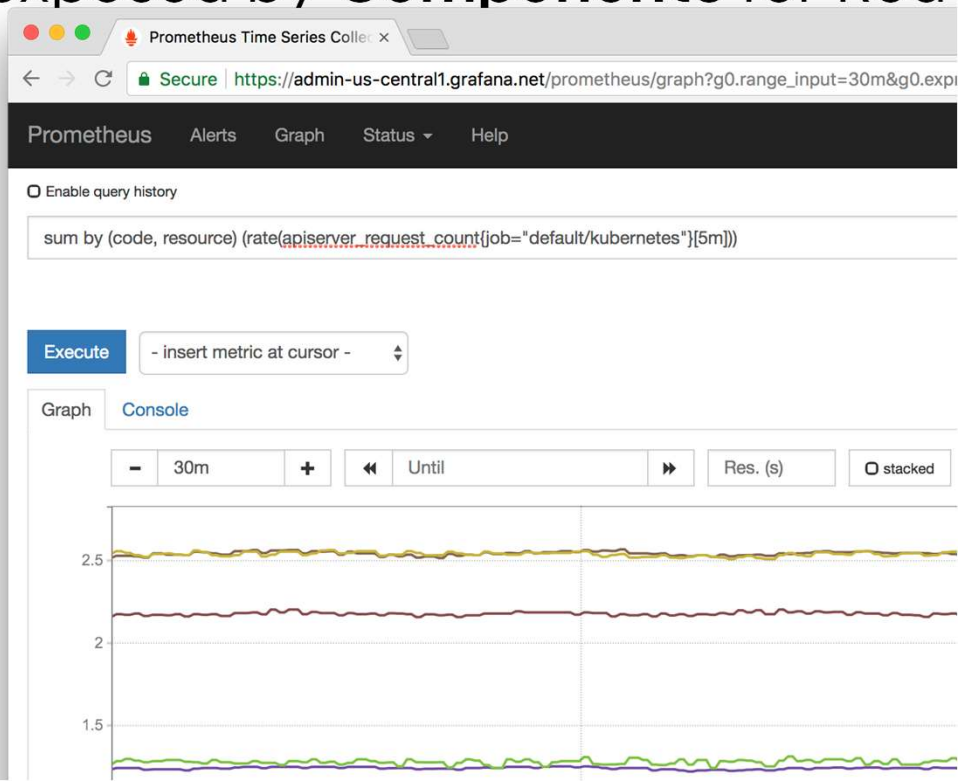
USE Method

- Container CPU usage by “app” label

```
sum by  (namespace,label_name)    ( sum by
       (pod_name, namespace(
     rate(container_cpu_usage_seconds_total[5m])
 )
 *     on  (pod_name) group_left(label_name)
 label_join(kube_pod_labels, "pod_name", ",“ , "pod"
)
```

RED Method

- Metrics exposed by **Components** for Red Style Montitoring



RED Method

- Some Functions

```
  100 * sum by(instance, job) (
  rate(rest_client_requests_total{code!~"2.."}[5m])
)
/
  sum by(instance, job)
  (rate(rest_client_requests_total[5m])
)
```

Ad-Hoc Methods

- *Alert expressions are invariants that describe a healthy system*
- Examples include:

- *kube_deployment_spec_replicas != kube_deployment_status_replicas_available*
- *rate(kube_pod_container_status_restarts_total [15m])>0*

Some Links to get started on Prometheus/Grafana

- github.com/coreos/prometheus-operator Job to look after running Prometheus on Kubernetes
- github.com/coreos/kube-prometheus Set of configs for running all there other things you need.
- github.com/grafana/jsonnet-libs/tree/master/prometheus-ksonnet – Sample configs for running Prometheus, Alertmanager, Grafana etc
- github.com/kubernetes-monitoring/kubernetes-mixin Joint project to unify and improve common alerts for Kubernetes.

Demo Time

Getting Started with Monitoring and Logging

Enabling Monitoring for Java Componens

- Refer: https://github.com/jreock/monitoring-java- apps-prometheus-grafana

What will be covered

- Expose JMX metrics from Java Application to Prometheus
- Import these metrics into Prometheus
- Visualize in Grafana
- Set up Alert Manager
- Create Thresholds and Alerts

Why??

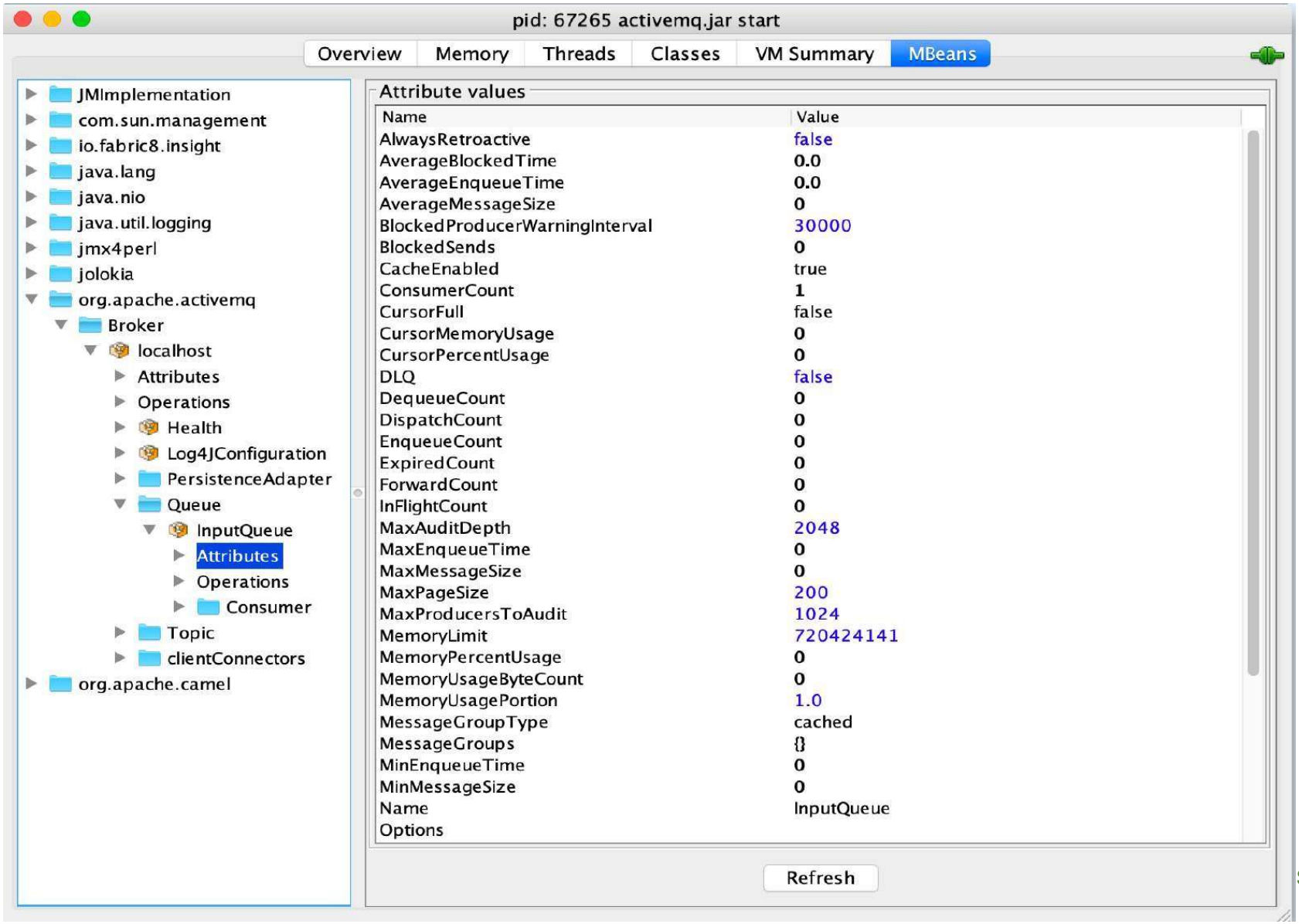
- It is one thing to monitor overall Kubernetes performance
- It is another to dig down into root cause of issues (usually at app level ☹)
- With Microservices, the amount of logging and monitoring needs have increased.

What is JMX

- JMX, or the Java Management eXtensions, is a Java-native specification that allows developers to
- expose application metrics in a standard, object-oriented way
- A full overview can be read here: https://docs.oracle.com/javase/tutorial/jmx/overview/index.html
- Create Java Objects called mBeans and exposed via TCP Port exposed by JVM
- JMX clients can connect on this port and collect metrics related to Application health.

JMX in the .NET World

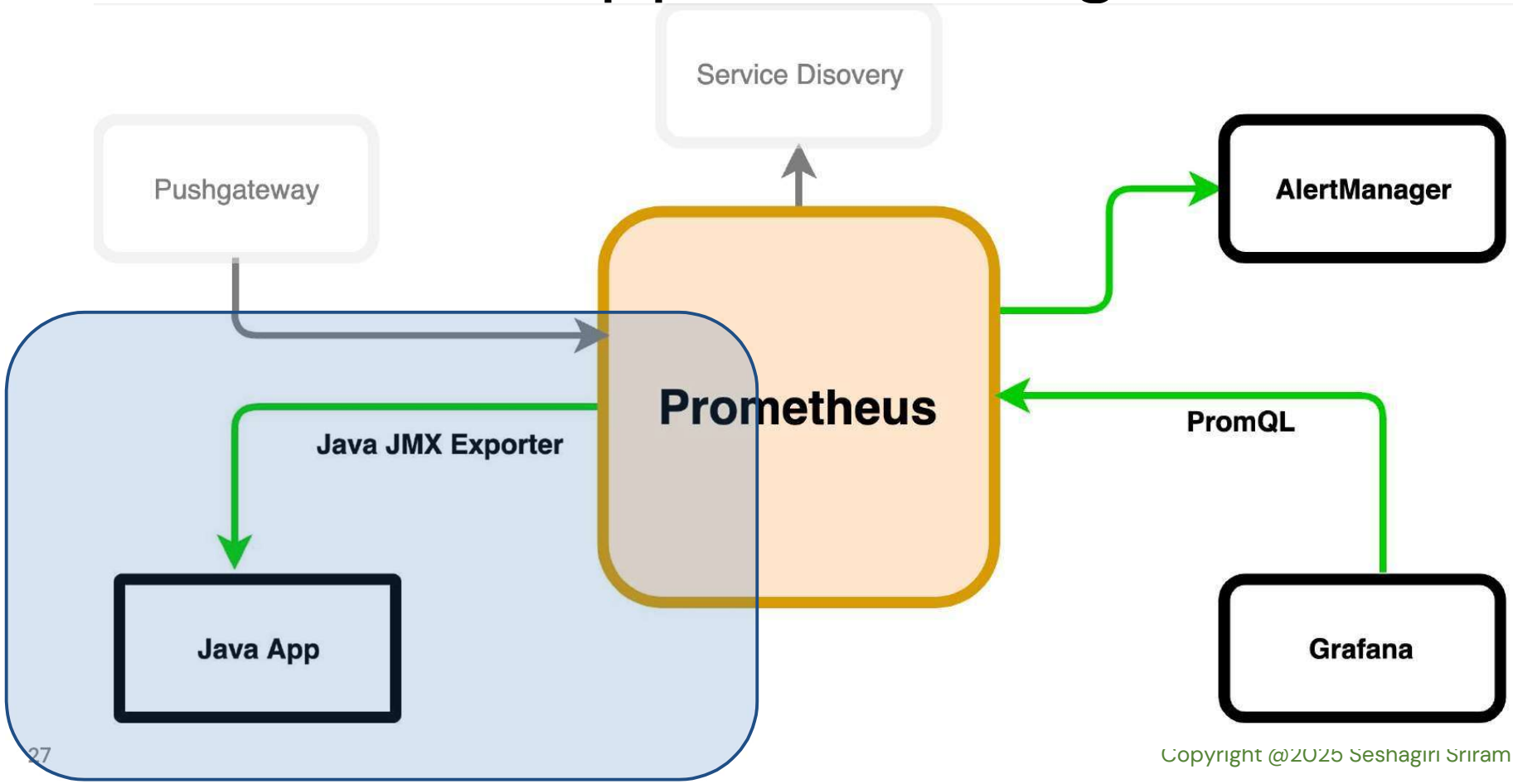
- Use WMI
- .NET profiling API
- Performance Counters


pid: 67265 activemq.jar start
Overview
Memory
Threads
Classes
VM Summary
MBeans
JMImplementation
com.sun.management
io.fabric8.insight
java.lang
java.nio
java.util.logging
jmx4perl
jolokia
org.apache.activemq
Broker
localhost
Attributes
Operations
Health
Log4JConfiguration
PersistenceAdapter
Queue
InputQueue
Attributes
Operations
Consumer
Topic
clientConnectors
org.apache.camel
Attribute values
Name
Value
AlwaysRetroactive false
AverageBlockedTime 0.0
AverageEnqueueTime 0.0
AverageMessageSize 0
BlockedProducerWarningInterval 30000
BlockedSends 0
CacheEnabled true
ConsumerCount 1
CursorFull false
CursorMemoryUsage 0
CursorPercentUsage 0
DLQ false
DequeueCount 0
DispatchCount 0
EnqueueCount 0
ExpiredCount 0
ForwardCount 0
InFlightCount 0
MaxAuditDepth 2048
MaxEnqueueTime 0
MaxMessageSize 0
MaxPageSize 200
MaxProducersToAudit 1024
MemoryLimit 720424141
MemoryPercentUsage 0
MemoryUsageByteCount 0
MemoryUsagePortion 1.0
MessageGroupType cached
MessageGroups {}
MinEnqueueTime 0
MinMessageSize 0
Name InputQueue
Options
Refresh

JMX and Java app Monitoring

- Prometheus depends on a number of exporters.
- These exporters expose metrics that are scraped by Prometheus
- All that’s need is a Prometheus agent Jar and our apps can use it

JMX and Java app Monitoring



Java and JMX

- In essence, all you need to do is
  - Include the Prometheus Agent Jar
  - And include our applications to start with a reference to the same
    - E.g. -javaagent:$APP_BASE/jmx_prometheus_javaagent-0.12.0.jar=8080:/$APP_BASE/conf/jmx-export-config.yml
  - We do need to provide a config yml (you can take on from Prometheus git hub site to start with)
  - And of course provide the Mbeans
  - Configure Prometheus to scrape t

Prometheus Configuration

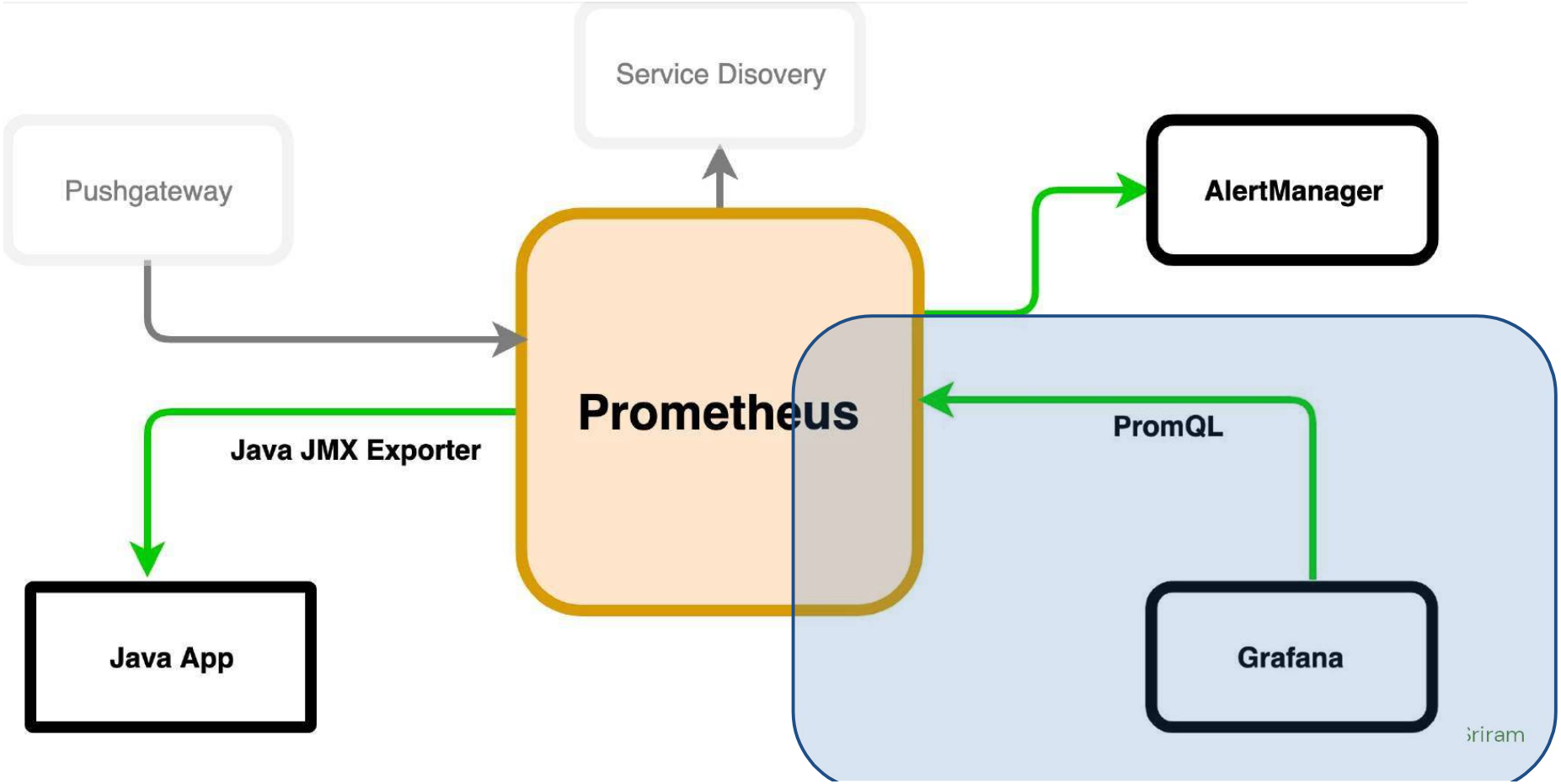
- Notice the rule_files section

```yaml
global:
  scrape_interval:     15s
  external_labels:
    monitor: 'activemq'

rule_files:
  - ./rules-amq.yml

scrape_configs:
  - job_name: 'activemq'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:8080']

alerting:
  alertmanagers:
    - scheme: http
      static_configs:
        - targets: ['localhost:9093']
```

Wrapping up

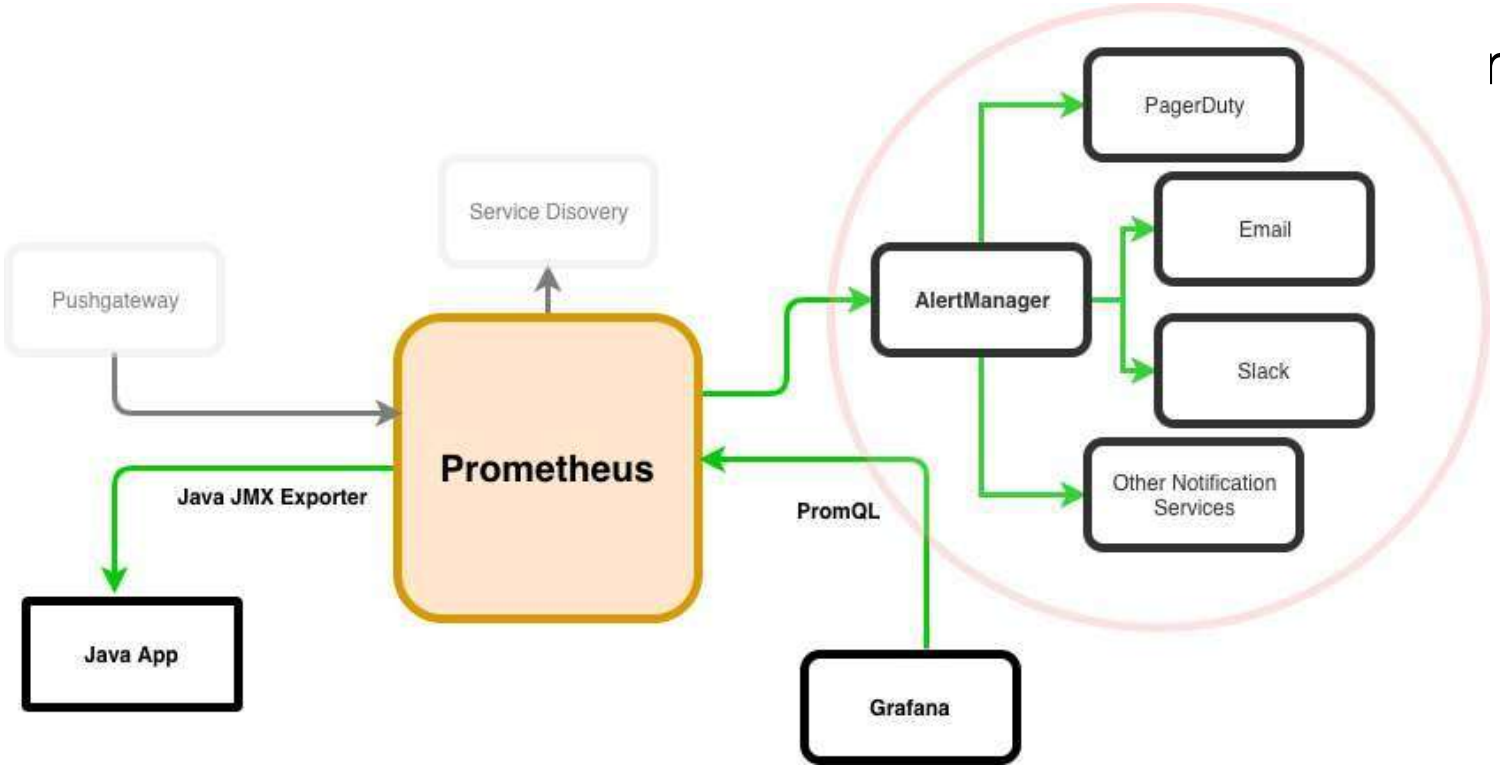
- Include a Grafana Dashboard to consume data from Prometheus for better visualization.

Wrapping up


Service Disovery
Pushgateway
AlertManager
Prometheus
Java JMX Exporter
PromQL
Java App
Grafana

Wrapping up

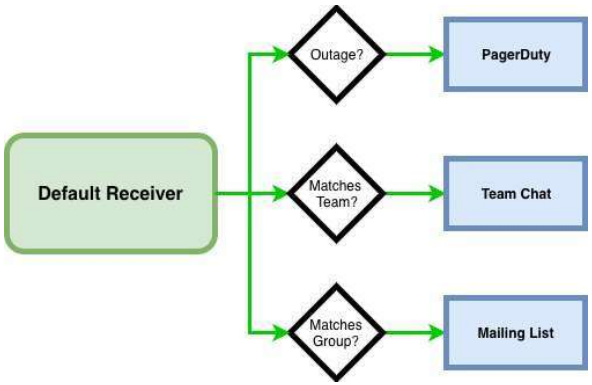
m



Wrapping up

- You do need an instance of an Alert Manager
- Configure Prometheus to use Alert Manager
- Creating Alert Rules in Prometheus

Alert Manager Rules



- AlertManager **rules are conceptualized as routes**, giving you the ability to write sophisticated sets of rules to determine where notifications should end up
- A **default receiver should be configured** for every notification, and then additional services can be configured through child routes which will match certain conditions

- A full configuration reference is available here: https://prometheus.io/docs/alerting/configuration

Alert Manager Rules

- Our config YAML file will be responsible for **setting up routing rules** that will determine how events are triaged
- As mentioned before, all events should **start with a default receiver**, called default-receiver, which will be the starting point for any route
- From there, any number **of sub-receivers can be configured**
- **Sample Configuration one called 'slack'** which will be invoked when the ”service” tag of the event that has been triggered matches “activemq”
- Next, **configure our receivers**
- Sample Slack receiver config will **contain WebHook** into Slack

```
global:
  smtp_smarthost: 'localhost:25'
  smtp_from: 'alertmanager@monitoring.com'

route:
  receiver: 'default-receiver'
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  group_by: [cluster, alertname]
  routes:
  - receiver: 'slack'
    group_wait: 10s
    match_re:
      service: activemq

receivers:
  - name: 'default-receiver'
    email_configs:
    - to: 'justin.reock@roguewave.com'

  - name: 'slack'
    slack_configs:
    - api_url: https://hooks.slack.com/services/
      channel: '#general'
```

Alert Manager Rules

- Configure Sample Rules
  - **two simple events**, but, events can be created out of a **huge range of possible query configurations**

```
groups:
- name: activemq
  rules:
  - alert: DLQ
    expr: org_apache_activemq_Broker_DLQ > 1
    for: 1m
    labels:
       severity: minor
       service: activemq
    annotations:
       summary: A message has gone into the DLQ
       dashboard: http://192.168.40.120:3000/dashboard/db/activemq-broker
       impact: A message has been misfired
       runbook:  http://activemq.apache.org
  - alert: Broker Down
    expr: up{job="activemq"} == 0
    labels:
       severity: major
       service: activemq
    annotations:
       summary: The broker has crashed
       dashboard: http://192.168.40.120:3000/dashboard/db/activemq-broker
       impact: Broker is down
       runbook:  http://activemq.apache.org
```

Integrating with Prometheus

- **configure Prometheus to push** alert events into AlertManager
- **Update prom- amq.yml** configuration file from earlier to integrate with our newly configured AlertManager instance
- Upon restarting Prometheus, we **should see our alerts** in the Prometheus dashboard

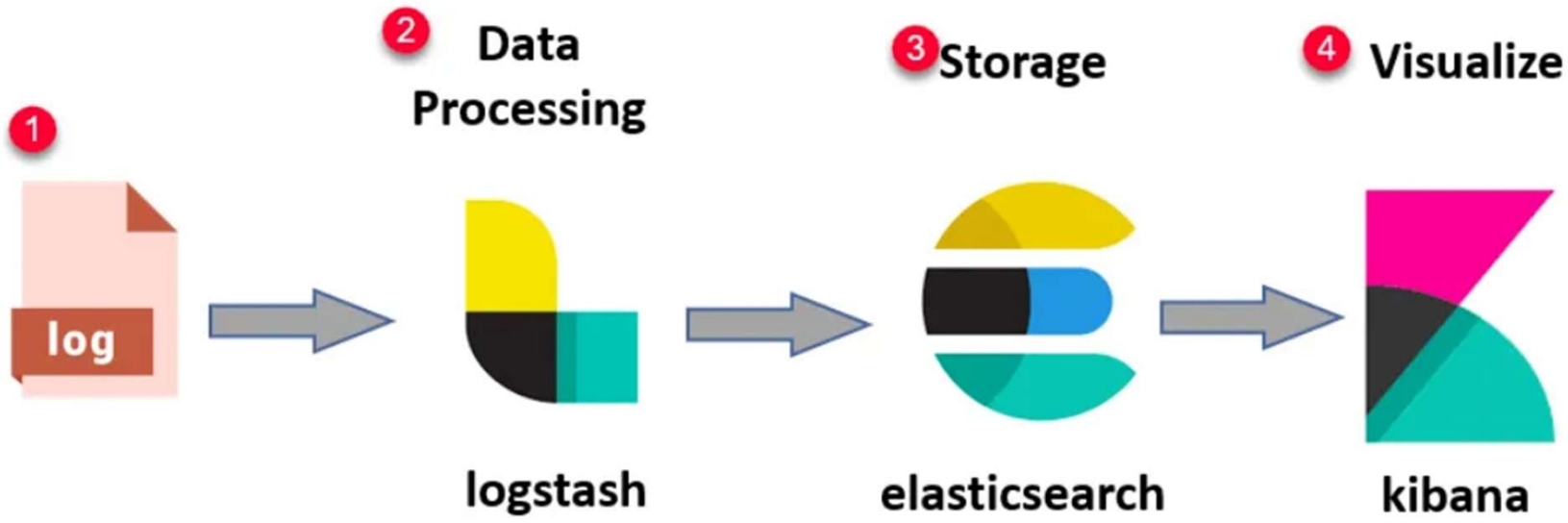
```
global:
  scrape_interval:      15s
  external_labels:
    monitor: 'activemq'

          -amq.yml
rule_files:
  - ./rules

scrape_configs:
  - job_name: 'activemq'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:8080']

alerting:
  alertmanagers:
    - scheme: http
      static_configs:
        - targets: ['localhost:9093']
```

Logging with ELK

Logging with ELK

- ELK Stack is the leading open-source IT log management solution for companies who want the benefits of a centralized logging solution without the enterprise software price.
- ELK is one of the most widely used stacks for processing log files and storing them as JSON documents. It is extremely configurable, versable, and scalable. It can be simple to use or complex, as it supports both simple and advanced operations.

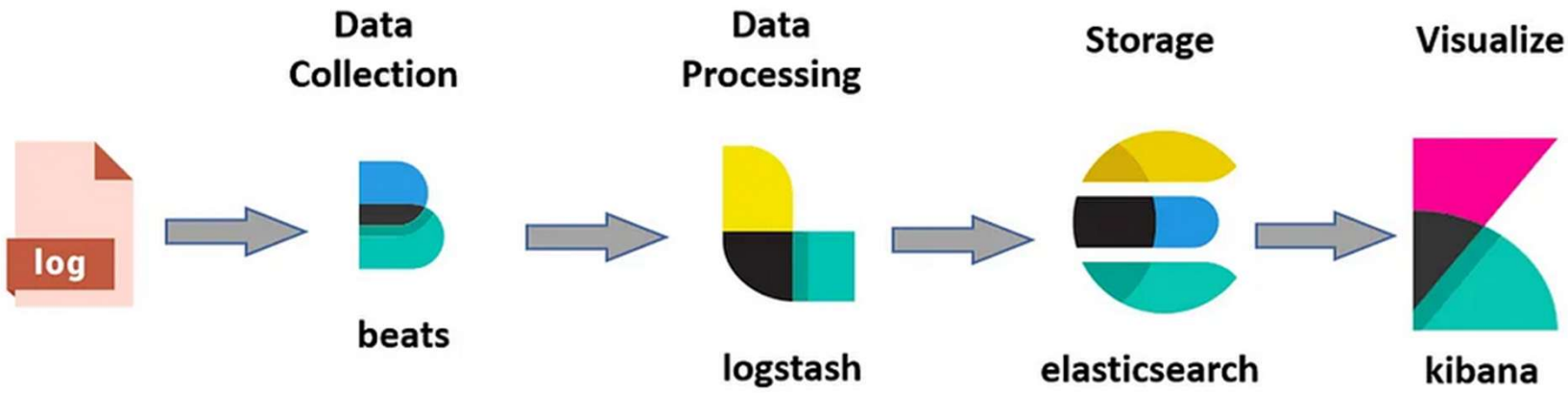
Logging with ELK



Logging with ELK

- **Logs:** Server logs that need to be analyzed are identified
- **Logstash:** Collect logs and events data. It even parses and transforms data
- **ElasticSearch:** The transformed data from Logstash is Store, Search, and indexed.
- **Kibana:** Kibana uses Elasticsearch DB to Explore, Visualize, and Share
- **NOT SHOWN:** BEATS

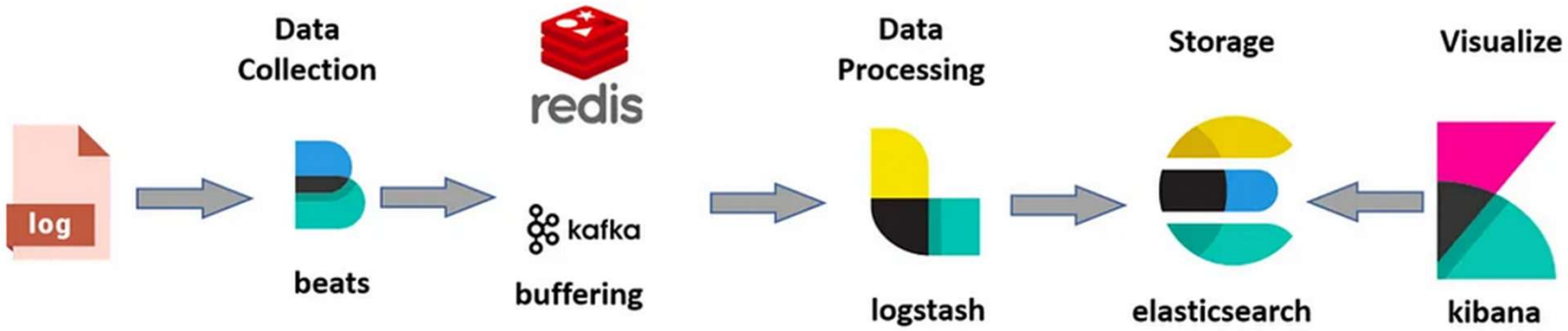
Logging with ELK



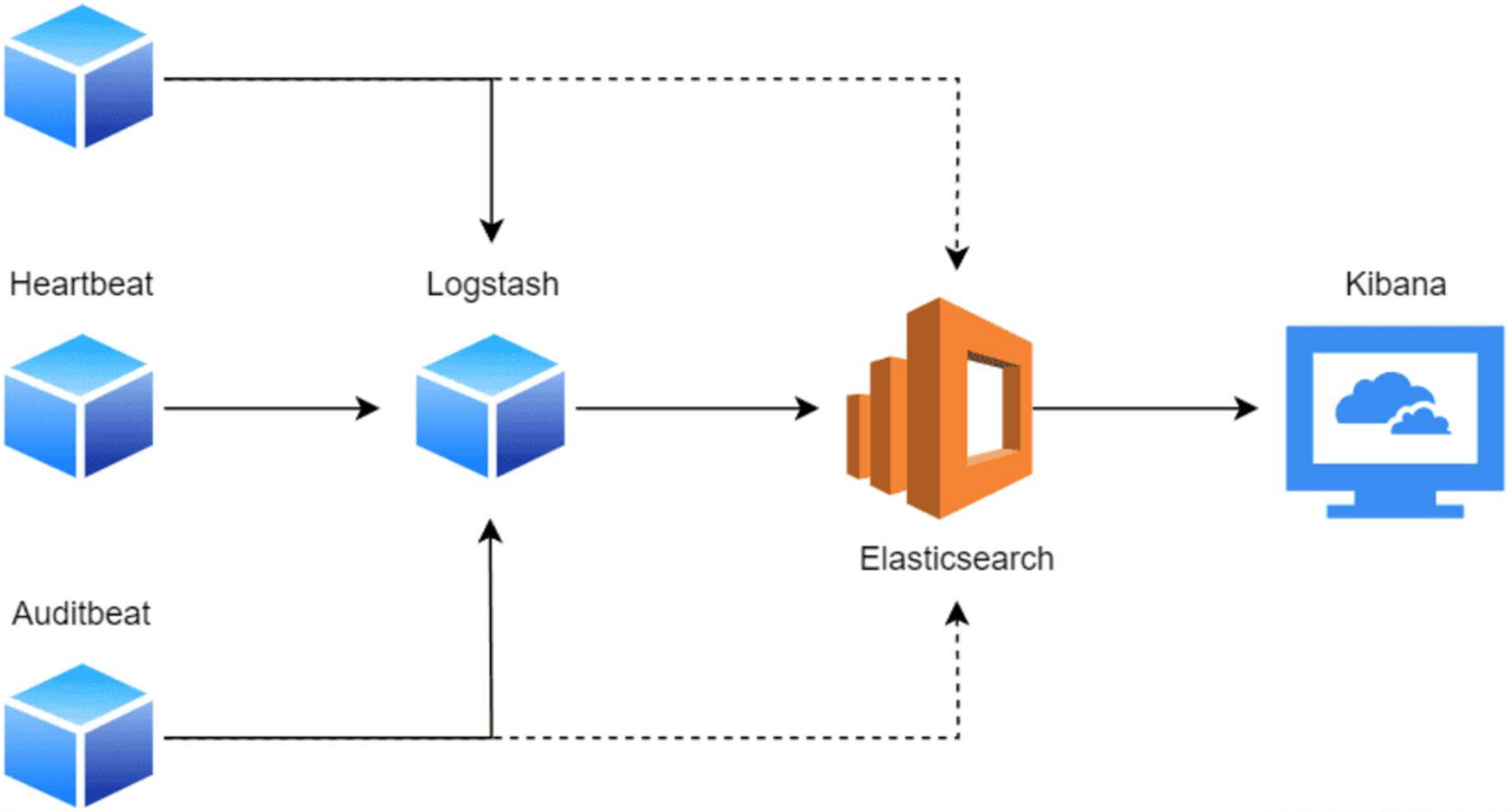
Logging with ELK

- As data size increases and there is need for securing, we use other tools like
  - Kafka
  - Redis
  - NGINX (for securing access)

Logging with ELK



Logging with ELK – Summary



Logging with ELK – Summary

- **Generally, a single Elasticsearch cluster aggregates data from several Logstash instances**. For example, multiple Logstash instances deployed on different nodes will collect data and feed it to the central Elasticsearch cluster for storage. Usually, these nodes are runtime environments like Kubernetes nodes, VMs, IoT devices, servers, and appliances.

- However, another approach to data collection is through the use of Beats. **Beats are lightweight single-purpose agents that ship data to a central Logstash instance**. Nonetheless, they can also send the data directly to an Elasticsearch cluster, but that's not recommended.

- Afterward, the aggregated data can be accessed through a Kibana instance, which provides a real-time interface for monitoring and data visualizations.

Installation on Kubernetes

- Steps
  - Install Elastic Search and Service
  - Configure Logstash to point to Elastic Search
  - Configure Logstash to get from a Beat Application
  - Install Logstash
  - Install Kibana and Service
- Document and Scripts Available on GitHub
- git@github.com:SeshagiriSriram/Devops.git

Demo Time

Getting Started with Monitoring and Logging

Q &A

We're done!

Thank you for your time and participation.