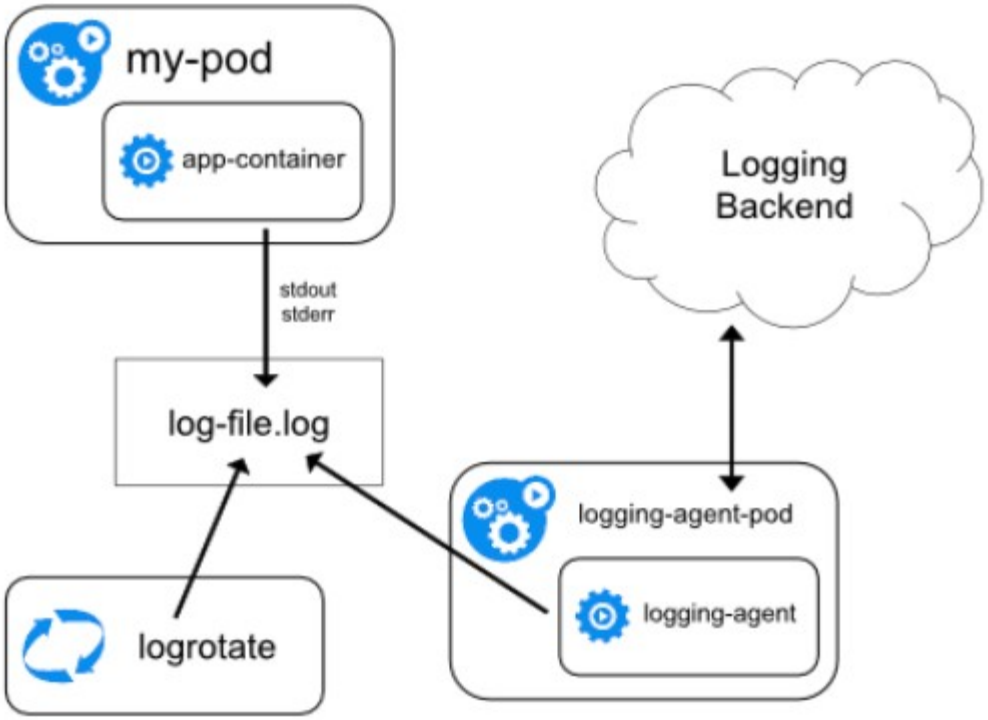
Cluster and Node Logging

Logging in Kubernetes

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

- Use a node-level logging agent that runs on every node.
- Include a dedicated sidecar container for logging in an application pod.
- Push logs directly to a backend from within an application.



The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Node-level logging creates only one agent per node and doesn't require any changes to the applications running on the node.

Containers write to stdout and stderr, but with no agreed format. A node-level agent collects these logs and forwards them for aggregation.

Log Aggregation

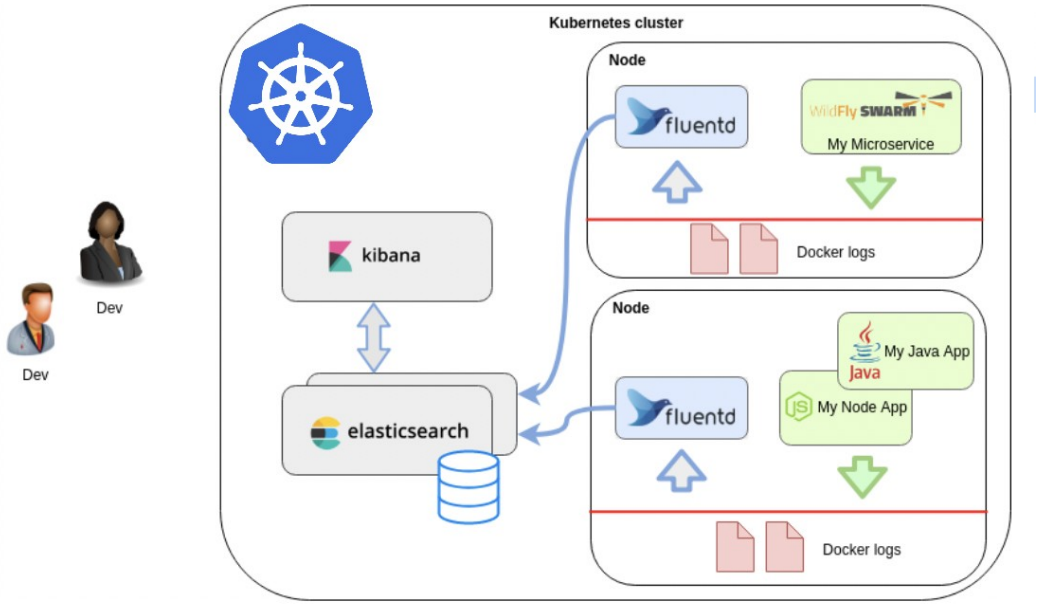
Kubernetes doesn't provide log aggregation of its own. However, Kubernetes release contains optional logging agents for Elasticsearch and for Stackdriver Logging (for use with Google Cloud Platform), and Fluentd as node agent.

The general architecture for cluster log aggregation is to have a local agent (such as Fluentd or Filebeat which are discussed below) to gather the data and send it to the central log management.

Fluentd

Fluentd is a popular open-source log aggregator that allows you to collect various logs from your Kubernetes cluster, process them, and then ship them to a data storage backend of your choice.

Kubernetes-native, Fluentd integrates seamlessly with Kubernetes deployments. The most common method for deploying Fluentd is as a daemonset which ensures a Fluentd pod runs on each pod. Similar to other log forwarders and aggregators, Fluentd appends useful metadata fields to logs such as the pod name and Kubernetes namespace, which helps provide more context.



ELK Stack

The ELK Stack (Elasticsearch, Logstash and Kibana) is another very popular open-source tool used for logging Kubernetes, and is actually comprised of four components:

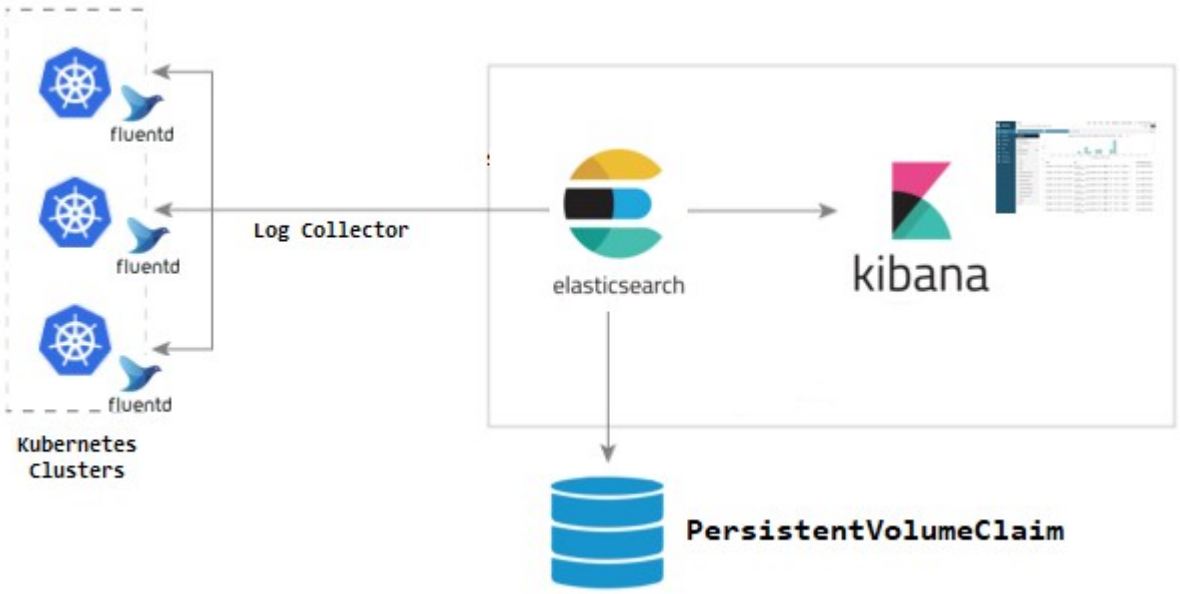
-Elasticsearch – provides a scalable, RESTful search and analytics engine for storing Kubernetes logs

-Kibana – the visualization layer, allowing you with a user interface to query and visualize logs

-Logstash – the log aggregator used to collect and process the logs before sending them into Elasticsearch

-Beats – Filebeat and Metricbeat are ELK-native lightweight data shippers used for shipping log files and metrics into Elasticsearch

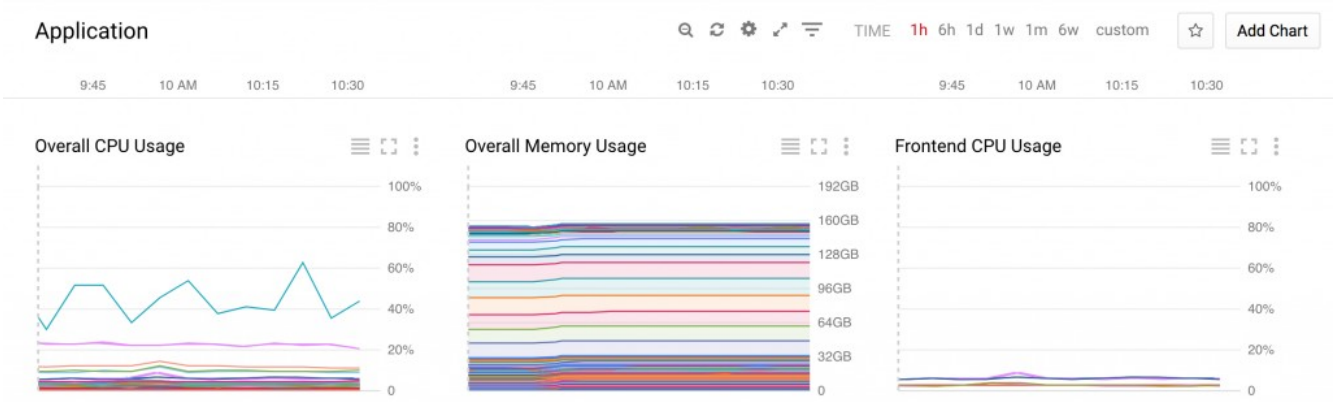
ELK can be deployed on Kubernetes as well, on-prem or in the cloud. While Beats is Elasticsearch's native shipper, a common alternative for Kubernetes installations is to use Fluentd to send logs to Elasticsearch (sometimes referred to as the EFK stack).



Google Stackdriver

And last but not least…Google Stackdriver.

Stackdriver is another Kubernetes-native logging tool that provides users with a centralized logging solution. After acquiring Stackdriver in 2014, Google worked hard to make it the default log aggregation and monitoring solution for Google Cloud Platform (GCP). The feature set of Stackdriver is pretty good for an out of box integrated solution for GCP. It is the equivalent of CloudWatch on AWS. As soon as you start your application on top of GKE, logs going to stdout or stderr from your containers will be pushed to Stackdriver Logs, ready for you to view and filter them. The magic happens when you create a GKE cluster - it will come preconfigured with Fluentd pushing logs to Stackdriver.



Logging Directories

Master

- /var/log/kube-apiserver.log - API Server, responsible for serving the API
- /var/log/kube-scheduler.log - Scheduler, responsible for making scheduling decisions

- /var/log/kube-controller-manager.log - Controller that manages replication controllers

Worker Nodes

- /var/log/kubelet.log - Kubelet, responsible for running containers on the node
- /var/log/kube-proxy.log - Kube Proxy, responsible for service load balancing

Standard Output and Standard Error

The first layer of logs that can be collected from a Kubernetes cluster are those being generated by your containerized applications. The best practice is to write your application logs to the standard output (*stdout*) and standard error (*stderr*) streams. You shouldn't worry about losing these logs, as *kubelet*, Kubernetes' node agent, will collect these streams and write them to a local file behind the scenes, so you can access them with Kubernetes.

Hands On – Test the Default Logging Mechanism

I. Create a manifest that creates a pod sending one message per second to STDOUT and STDERR.

Notice in the *container* field we are running a script when deploying this pod. Essentially, this is sending one message per second to the STDOUT. This way we can easily retrieve the logs.

```
~ $cat doms-counter-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: ubuntu
    args: [/bin/bash, -c,
           'i=0; while true; do echo "STDOUT: $i: $(date)"; my-error; i=$((i+1)); sleep 1; done']
~ $kubectl apply -f doms-counter-pod.yaml
pod/counter unchanged
~ $kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
counter   1/1     Running   0          29s
~ $
```

II. Differentiate between Standard Out and Standard Error Logs.

Verify your log counter is working properly by issuing the kubectl logs counter command for the counter pod. Notice that there are no STDERR logs being generated.

```
~ $kubectl logs counter
STDOUT: 0: Thu Mar 24 17:10:59 UTC 2022
/bin/bash: my-error: command not found
STDOUT: 1: Thu Mar 24 17:11:00 UTC 2022
/bin/bash: my-error: command not found
STDOUT: 2: Thu Mar 24 17:11:01 UTC 2022
/bin/bash: my-error: command not found
STDOUT: 3: Thu Mar 24 17:11:02 UTC 2022
/bin/bash: my-error: command not found
STDOUT: 4: Thu Mar 24 17:11:03 UTC 2022
/bin/bash: my-error: command not found
STDOUT: 5: Thu Mar 24 17:11:04 UTC 2022
/bin/bash: my-error: command not found
STDOUT: 6: Thu Mar 24 17:11:05 UTC 2022
/bin/bash: my-error: command not found
```

Let's edit the original manifest to include STDERR logs as well. Notice, we are now separating the STDOUT from STDERR into separate files. Apply your changes and verify.

```
23  apiVersion: v1
24  kind: Pod
25  metadata:
26    name: counter-stdout-stderr
27  spec:
28    containers:
29    - name: count
30      image: ubuntu
31      args: [/bin/bash, -c,
32             '(i=0; while true; do echo "STDOUT: $i: $(date)"; my-error; i=$((i+1)); sleep 1; done) 1>> STDOUT.file 2>> STDERR.file']
33
34
```

Although we may not see specific error logs, we have now verified that standard output and standard errors are now being logged to different files. Should the pod experience any errors during the deployment, we would be able to issue the necessary commands to determine the issue. If you are able to see each of these files separately, you have viewed the logs and completed the training.

```
~ $kubectl exec -it counter-stdout-stderr -- tail STDOUT.file
STDOUT: 27: Thu Mar 24 17:21:12 UTC 2022
STDOUT: 28: Thu Mar 24 17:21:13 UTC 2022
STDOUT: 29: Thu Mar 24 17:21:14 UTC 2022
STDOUT: 30: Thu Mar 24 17:21:15 UTC 2022
STDOUT: 31: Thu Mar 24 17:21:16 UTC 2022
STDOUT: 32: Thu Mar 24 17:21:17 UTC 2022
STDOUT: 33: Thu Mar 24 17:21:18 UTC 2022
STDOUT: 34: Thu Mar 24 17:21:19 UTC 2022
STDOUT: 35: Thu Mar 24 17:21:20 UTC 2022
STDOUT: 36: Thu Mar 24 17:21:21 UTC 2022
~ $kubectl exec -it counter-stdout-stderr -- tail STDERR.file
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
/bin/bash: my-error: command not found
```

This is a rough diagram of what you have just created from your Pod.