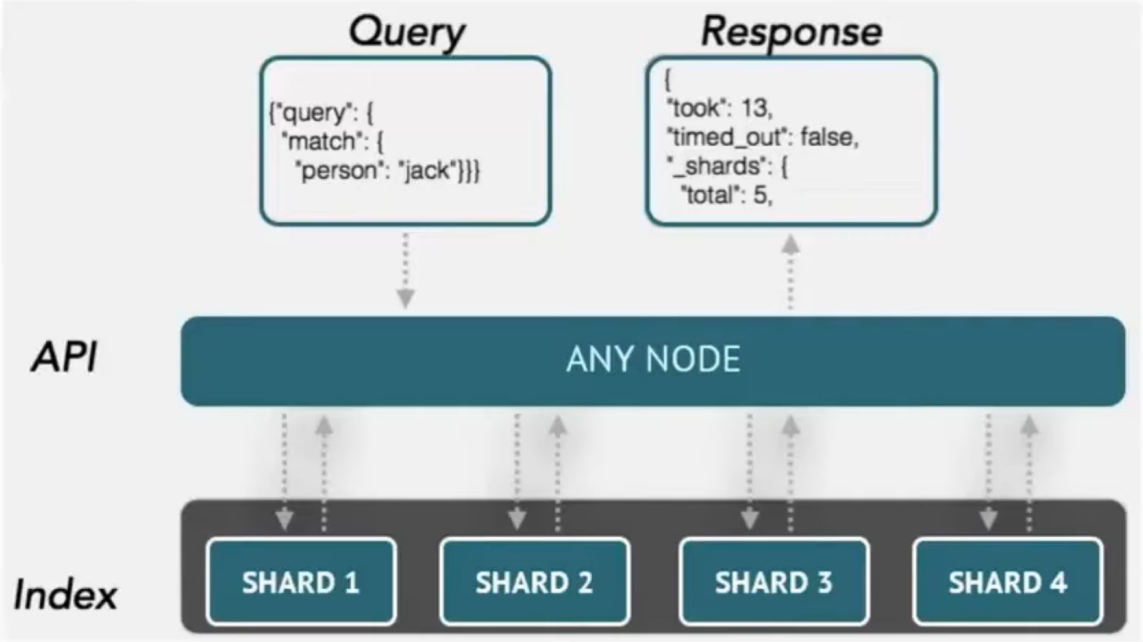
Elasticsearch is a database that stores, retrieves, and manages document-oriented and semi-structured data.

How does Elasticsearch work?

Raw data flows into Elasticsearch from a variety of sources, including logs, system metrics, and web applications.

Data ingestion is the process by which this raw data is parsed, normalized, and enriched before it is indexed in Elasticsearch.

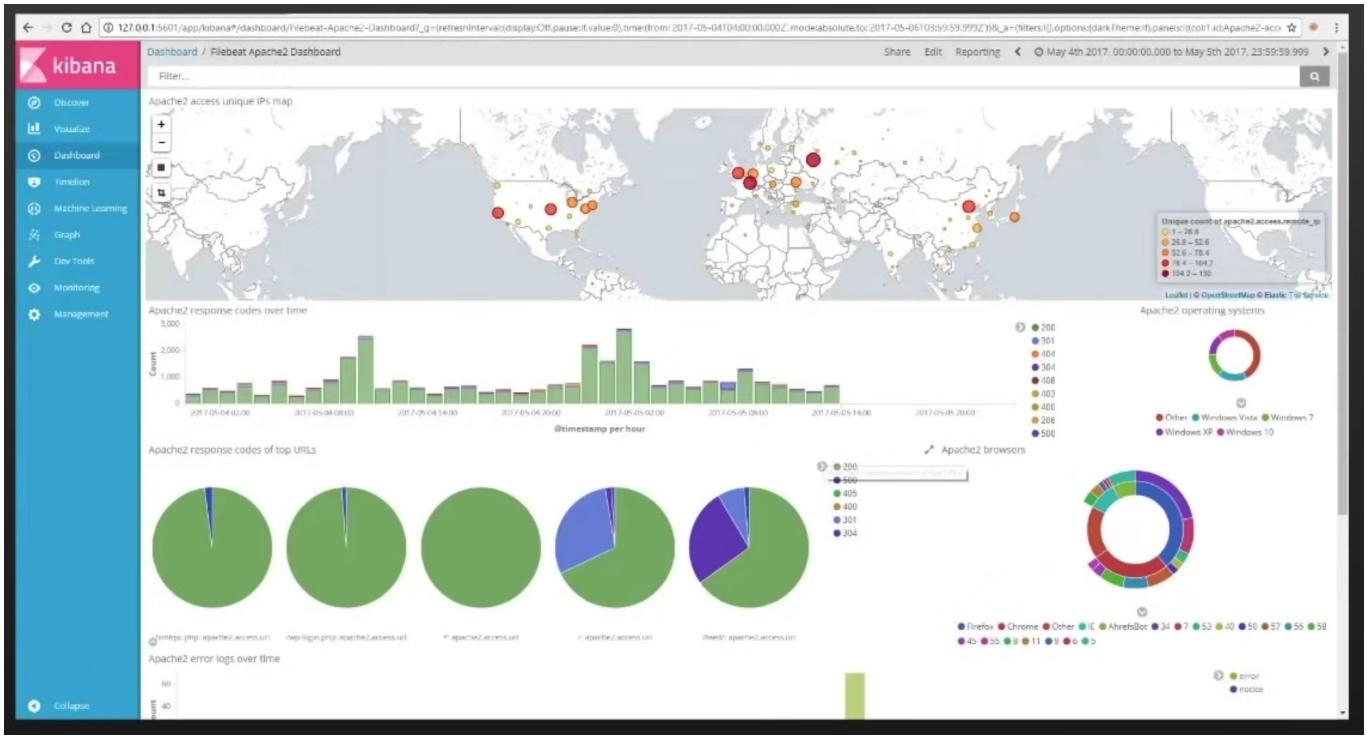


Cluster

A cluster is a collection of one or more servers that together hold entire data and give federated indexing and search capabilities across all servers. For relational databases, the node is DB Instance. There can be N nodes with the same cluster name.

Kibana

- Web UI for searching and visualizing
- Complex aggregations, graphs, charts
- Often used for log analysis



Logstash / Beats

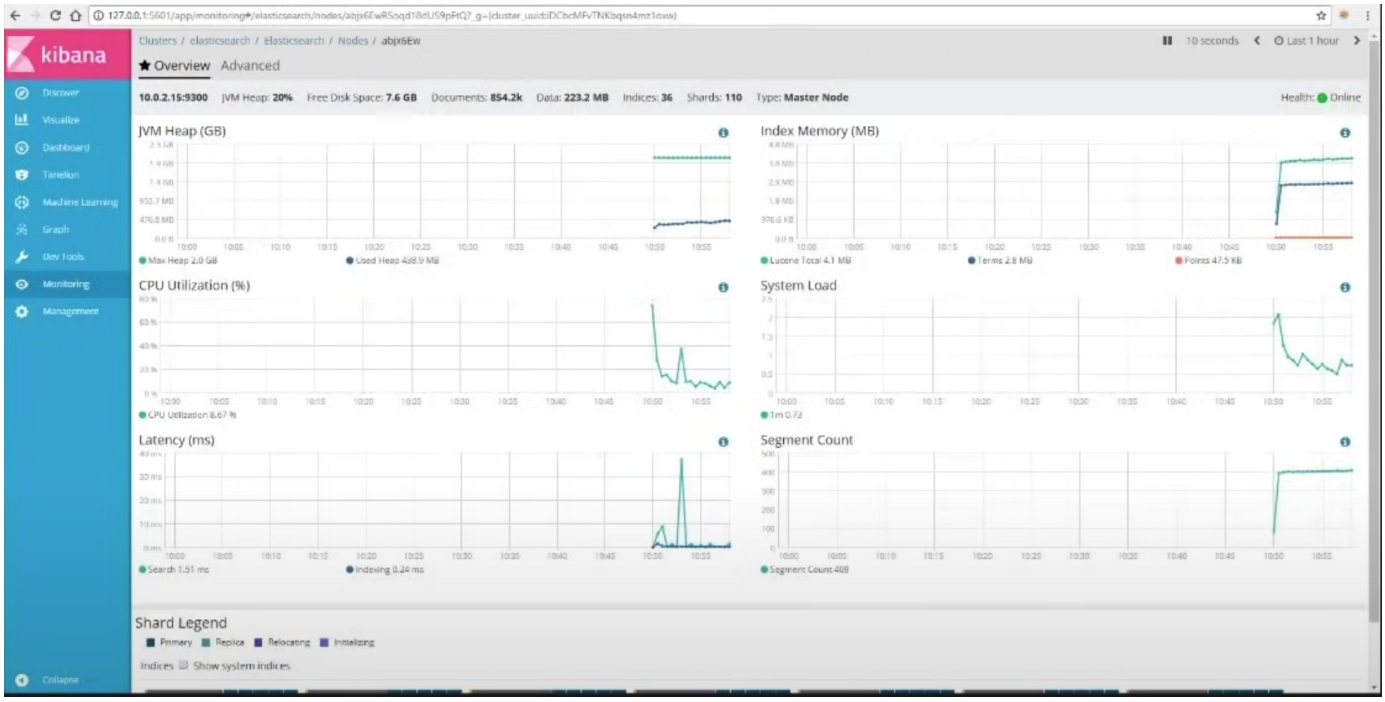
- Ways to feed data into Elasticsearch
- FileBeat can monitor log files, parse them, and import into Elasticsearch in near-real-time
- Logstash also pushes data into Elasticsearch from many machines
- Not just log files

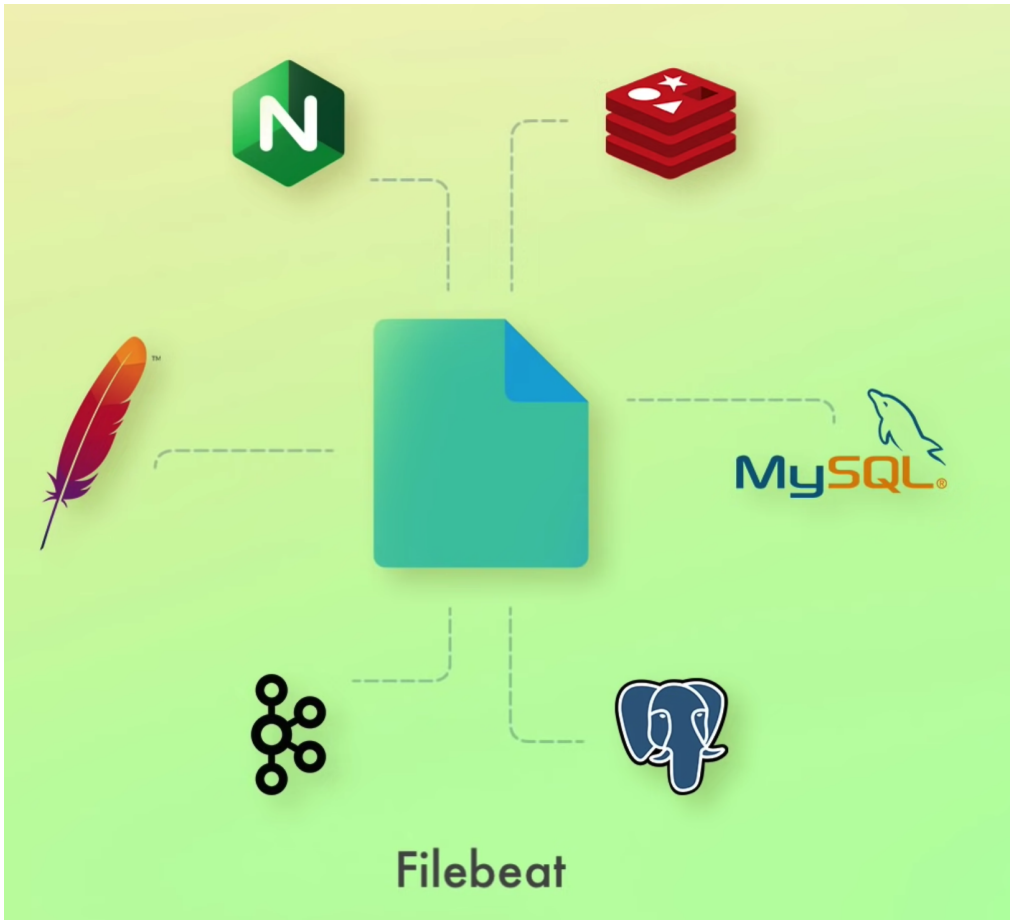
X-Pack

- Security
- Alerting
- Monitoring
- Reporting
- Machine Learning
- Graph Exploration

* X-Pack can integrate with Kibana/ElasticSearch for authentication via LDAP.
* Can monitor and view the performance of ELK stack, cpu, memory, disk space,...

* setup alerting for the ELK stack based on metrics
* generate reports based on ELK performance CSV, PDF, spreadsheet,..)
* enabled machine learning on Kibana (anomaly detection, forecasting,..)
* allowing querying ELK data directly with SQL





FILEBEAT

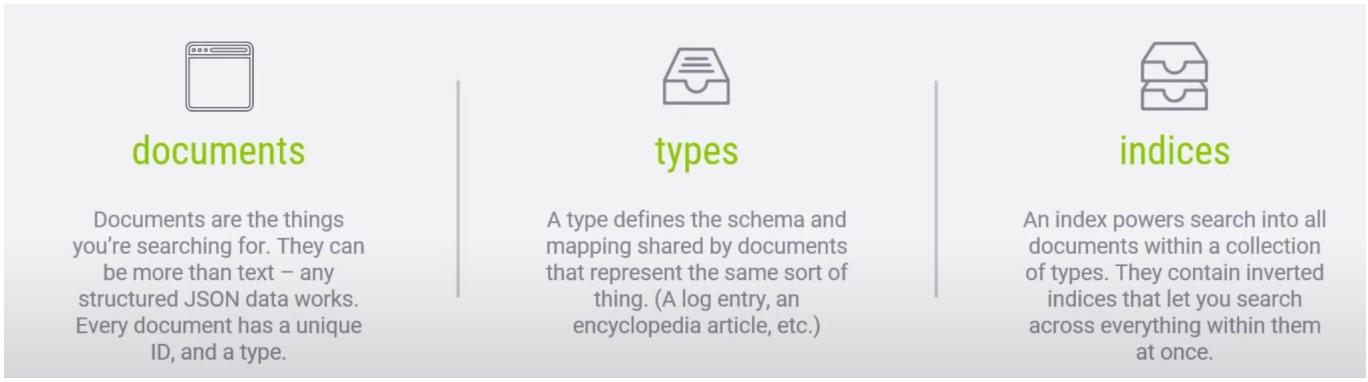
- Collects log files
- Includes modules for common log types (e.g. nginx and MySQL)

METRICBEAT

- Collects system and service metrics
- E.g. memory and CPU usage
- Includes modules for common services (e.g. nginx and MySQL)

PACKETBEAT

- Collects network data
- E.g. HTTP requests or database transactions



Index => database
Type => table
Document => row

Index

The index is a collection of documents that have similar characteristics.

Node

A node is a single server that holds some data and participates on the cluster's indexing and querying. A node can be configured to join a specific cluster by the particular cluster name. A single cluster can have as many nodes as we want. A node is simply one Elasticsearch instance.

Shards

A shard is a subset of documents of an index. An index can be divided into many shards.

Document 1:
Space: The final frontier. These are the voyages…

Document 2:
He's bad, he's number one. He's the space cowboy with the laser gun!

Inverted index

space:	1, 2
the:	1, 2
final:	1
frontier:	1
he:	2
bad:	2
…	

The data you store in Elasticsearch generally falls into one of two categories:

- Content: a collection of items you want to search, such as a catalog of products
- Time series data: a stream of continuously-generated timestamped data, such as log entries

RESTful API

Elasticsearch fundamenatally works via HTTP requests and JSON data. Any language or tool that can handle HTTP can use Elasticsearch.

client API's

Most languages have specialized Elasticsearch libraries to make it even easier.

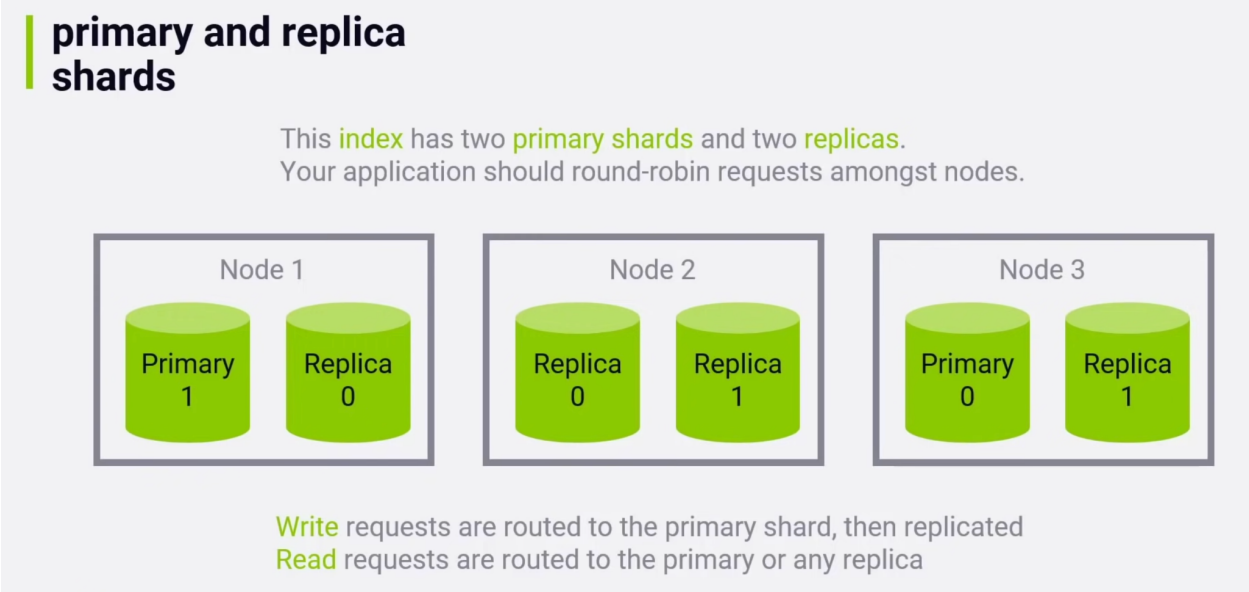
analytic tools

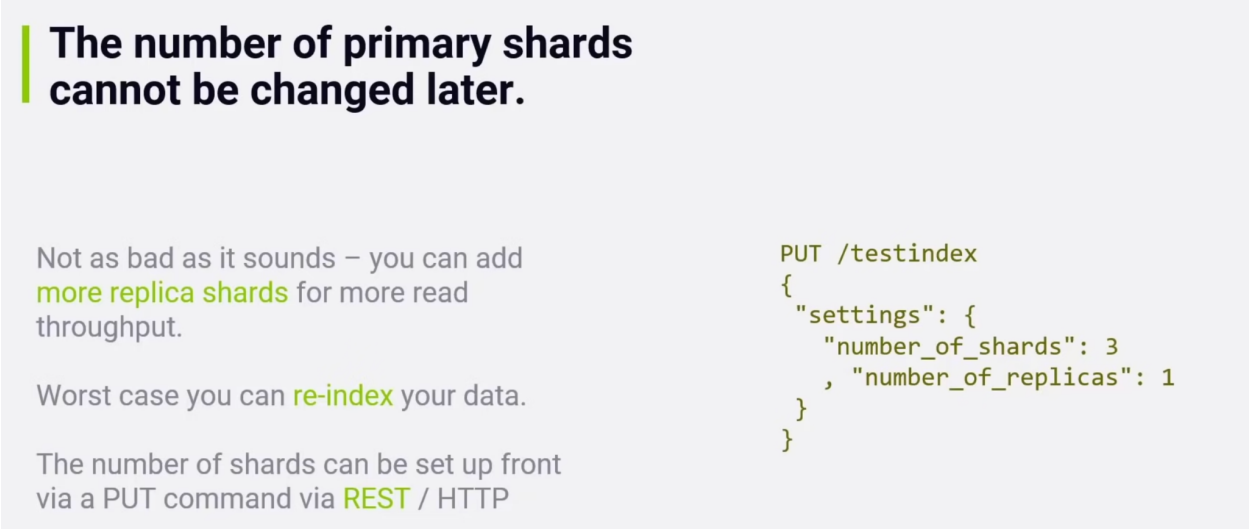
Web-based graphical UI's such as Kibana let you interact with your indices and explore them without writing code.

an index is split into shards.

Documents are hashed to a particular shard.

Each shard may be on a different node in a cluster.
Every shard is a self-contained Lucene index of its own.





The number of shards you can hold on a node will be proportional to the amount of heap you have available, but there is no fixed limit enforced by Elasticsearch. A good rule-of-thumb is to ensure you keep the number of shards per node below 20 per GB heap it has configured. A node with a 30GB heap should therefore have a maximum of 600 shards, but the further below this limit you can keep it the better. This will generally help the cluster stay in good health.

Elastisearch saves the information into indexes and each index has one or more shards. Each shard is an instance of a Lucene Index and on top of that, some of the shards can be replicas. Due to the nature of elasticsearch, each index should have multiples shards and replicas.

Replicas provide redundant copies of your data, to protect against any future problems, like hardware failures.

Create an index

```
curl -X PUT "localhost:9200/bank?pretty"

curl --user elastic -X PUT "localhost:9200/bank?pretty"
```

Create an index, for the index have 3 shards and 2 replicas of each shard

```
curl --user elastic:medium -X PUT "localhost:9200/bank?pretty"
-H 'Content-Type: application/json' -d'
{
  "settings": {
    "index": {
      "number_of_shards": 3,
      "number_of_replicas": 2
    }
  }
}
```

Check the existing indexes

```
curl --user elastic "localhost:9200/_cat/indices?v"
```

Add aliases to an index

```
curl --user elastic:medium -X POST "localhost:9200/_aliases" -H
'Content-Type: application/json' -d'
{
  "actions": [
    {
      "add": {
        "index": "bank",
        "alias": "banco"
      }
    }
  ]
}
```

Delete an index

```
curl --user elastic -X DELETE "localhost:9200/bank?pretty"
```

Add mapping when creating index

```
curl --user elastic:medium -X PUT
"localhost:9200/twitter?pretty"  -H 'Content-Type:
application/json' -d'
{
  "settings": {
    "number_of_shards": 2,"number_of_replicas": 3},
  "mappings": {
      "properties": {
        "content": {
          "type": "text"
        },
        "user_name": {
          "type": "keyword"
        },
        "tweeted_at": {
          "type": "date"
        }
      }
  }
}
```

see the mappings for the index

```
curl --user elastic -X GET "localhost:9200/twitter/_mapping"
```

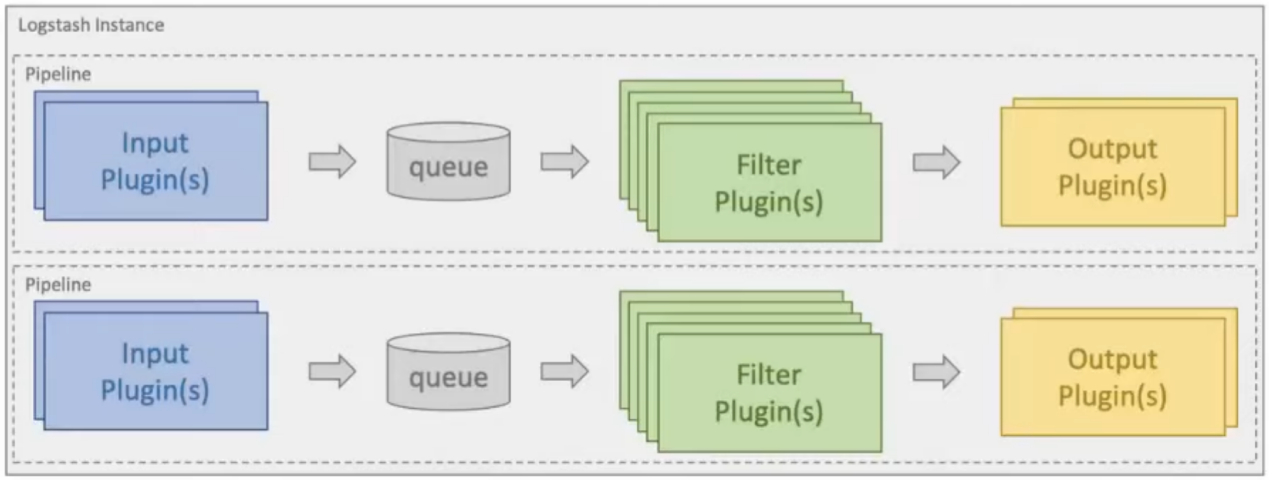
add documents in bulk operation to index, from the file tweets.json

```
curl --user elastic -H "Content-Type: application/json" -XPOST
"localhost:9200/twitter/_bulk?pretty&refresh" --data-binary
"@tweets.json"
```

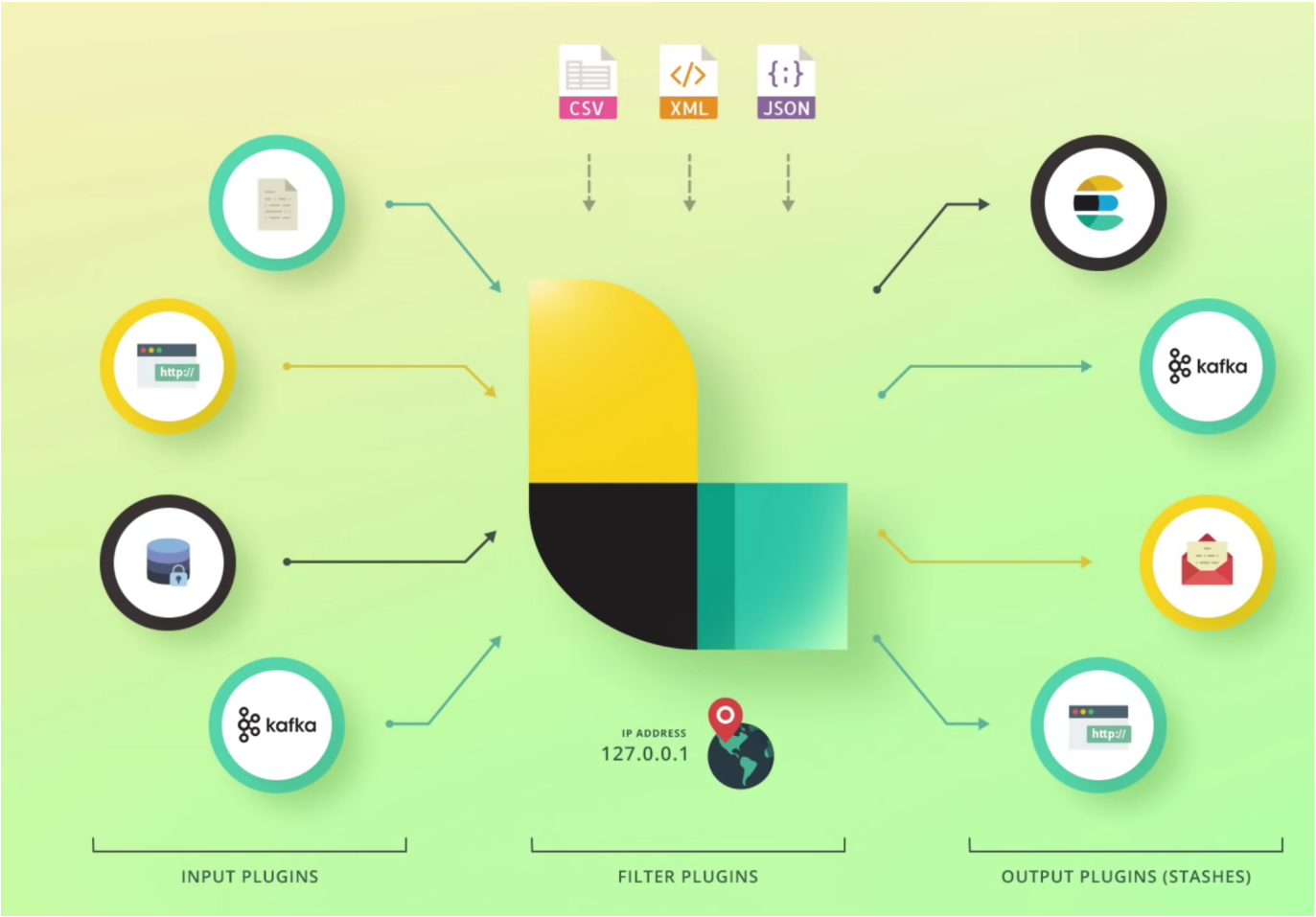
re-indexing data

```
curl --user elastic:medium -X POST "localhost:9200/_reindex" -H
'Content-Type: application/json' -d'
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  } }
```

Logstash is a tool based on the filter/pipes patterns for gathering, processing and generating the logs or events. It helps in centralizing and making real time analysis of logs and events from different sources.



- Logstash can collect data from different sources and send to multiple destinations.
- Logstash can also handle http requests and response data.
- Logstash can handle all types of logging data like Apache Logs, Windows Event Logs, Data over Network Protocols, Data from Standard Input and many more.
- Logstash provides a variety of filters, which helps the user to find more meaning in the data by parsing and transforming it.
- Logstash can also be used for handling sensors data in internet of things.


CSV
XML
JSON
http://
kafka
kafka
IP ADDRESS
127.0.0.1
http://
INPUT PLUGINS
FILTER PLUGINS
OUTPUT PLUGINS (STASHES)

Example pipeline configuration

```
input {
    file {
        path => "/path/to/apache_access.log"
    }
}

filter {
    if [request] in ["/robots.txt", "/favicon.ico"] {
        drop { }
    }
}

output {
    file {
        path => "%{type}_%{+yyyy_MM_dd}.log"
    }
}
```

Event Object

It is the main object in Logstash, which encapsulates the data flow in the Logstash pipeline. Logstash uses this object to store the input data and add extra fields created during the filter stage. Logstash offers an Event API to developers to manipulate events.

Pipeline

It comprises of data flow stages in Logstash from input to output. The input data is entered in the pipeline and is processed in the form of an event. Then sends to an output destination in the user or end system's desirable format.

Input

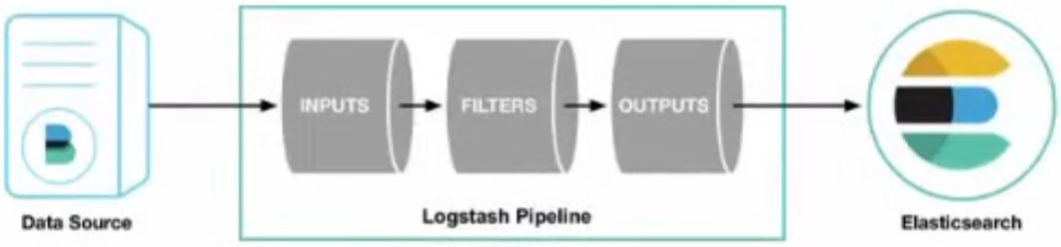
This is the first stage in the Logstash pipeline, which is used to get the data in Logstash for further processing. Logstash offers various plugins to get data from different platforms. Some of the most commonly used plugins are – File, Syslog, Redis and Beats.

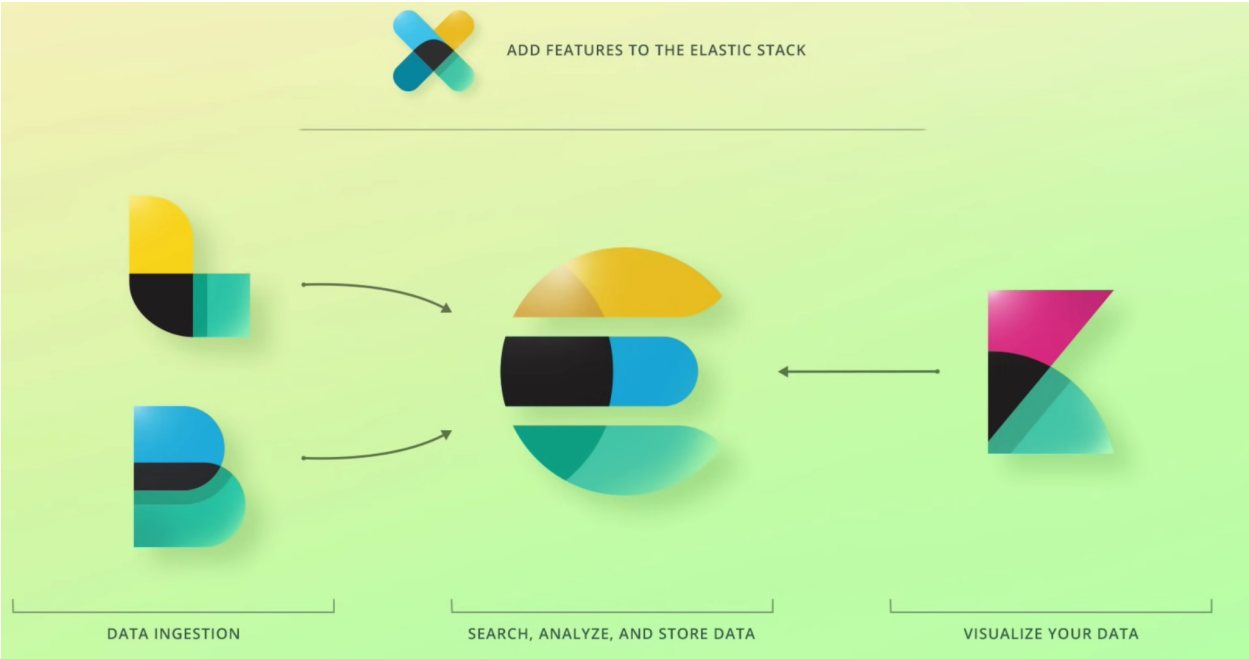
Filter

This is the middle stage of Logstash, where the actual processing of events take place. A developer can use pre-defined Regex Patterns by Logstash to create sequences for differentiating between the fields in the events and criteria for accepted input events.

Output

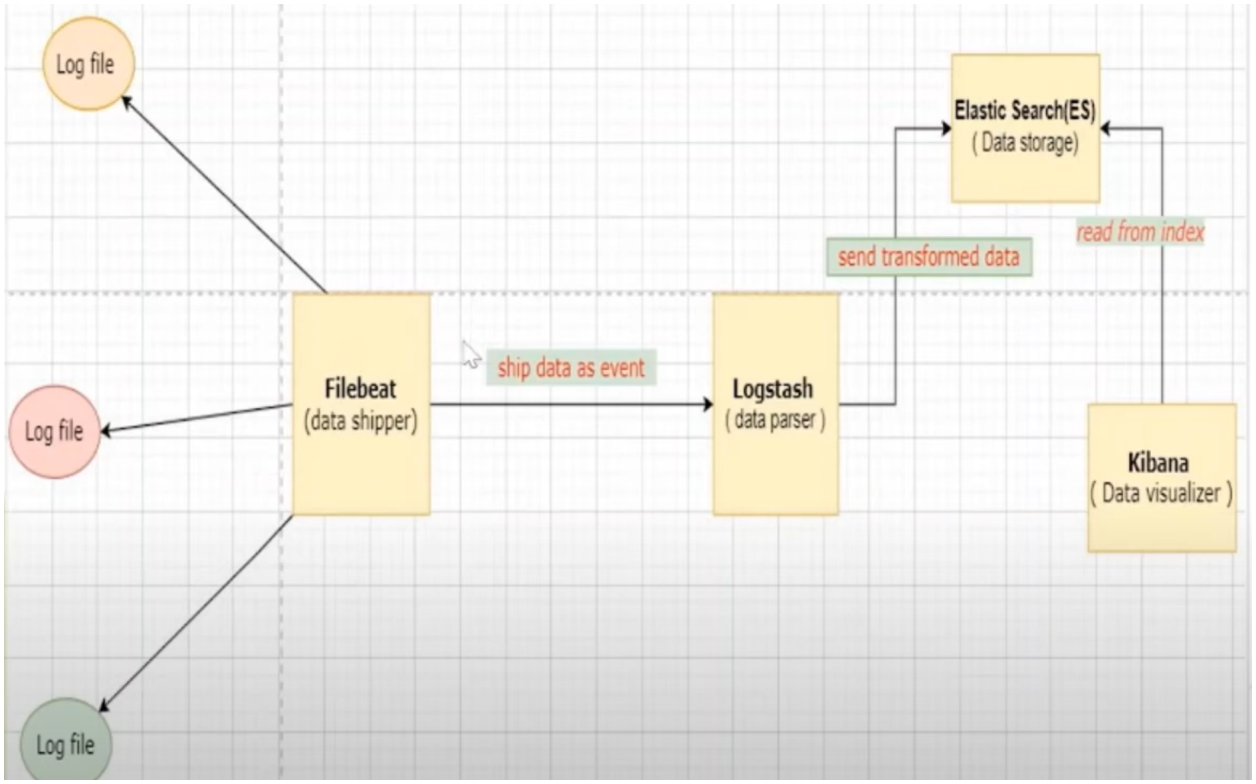
This is the last stage in the Logstash pipeline, where the output events can be formatted into the structure required by the destination systems. Lastly, it sends the output event after complete processing to the destination by using plugins. Some of the most commonly used plugins are – Elasticsearch, File, Graphite, Statsd, etc.



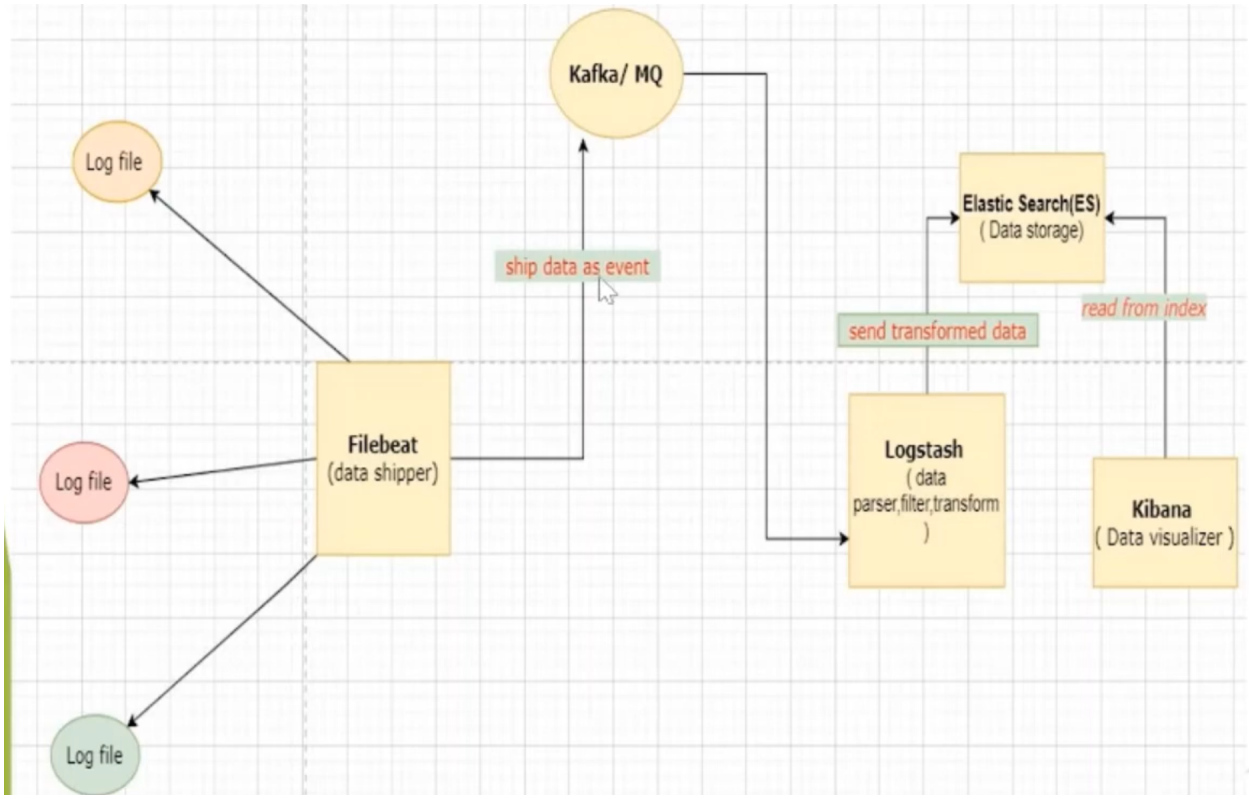


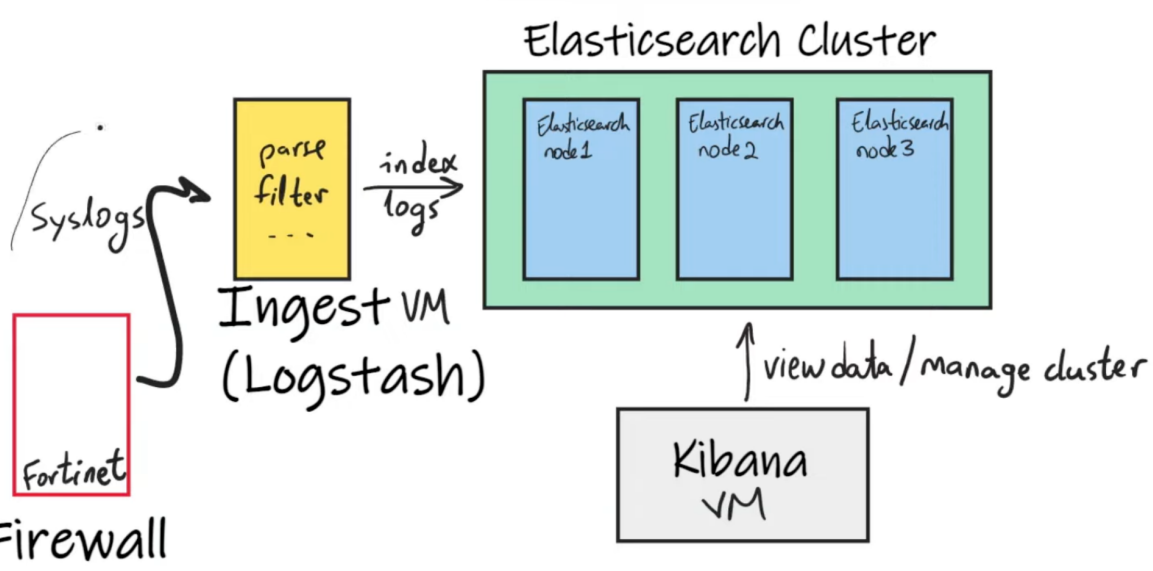
Data ingestion => FileBeat+ LogStash
FileBeat directly working with Elastic API
Search, analyze and save => ElasticSearch
Data Visualization => Kibana
Security, Monitoring (add-on) => X-pack

a. FileBeat reads log data from application logs and sends it to LogStash
b. LogStash reads data, formats and filters it and sends it in JSON format to ElasticSearch
c. ElasticSearch stores data, can query this data,...
d. Kibana uses Elastic to visualize data into charts



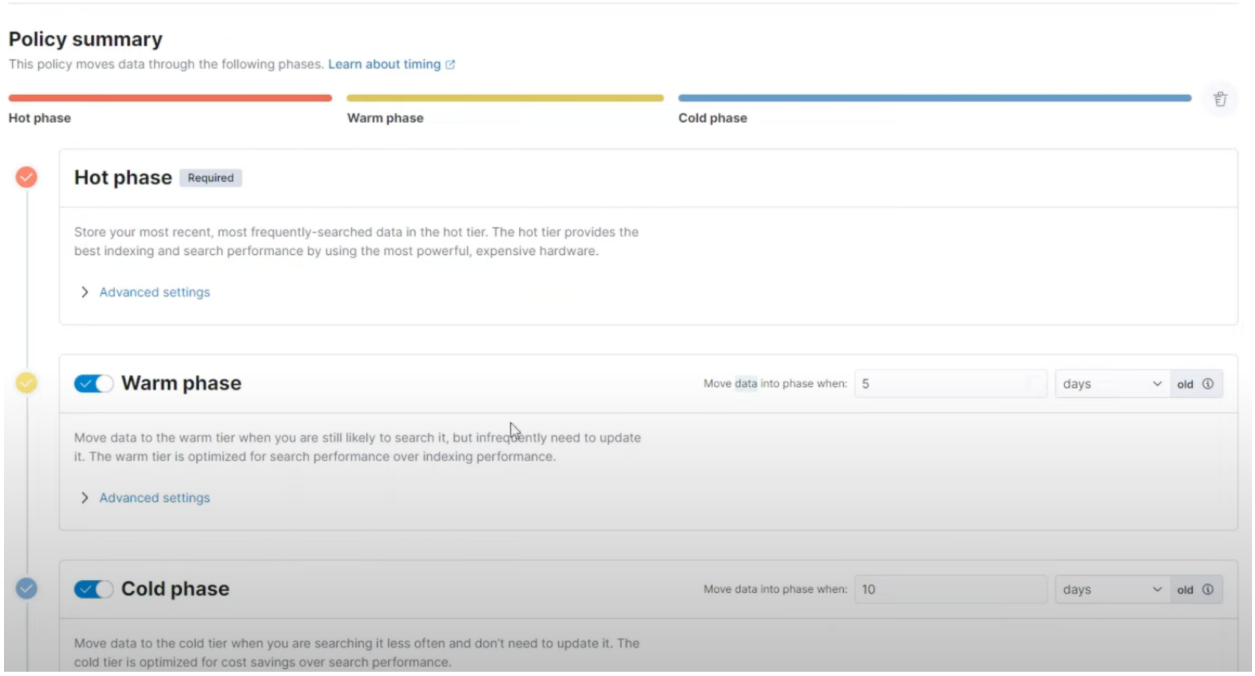
We use Kafka when we have too much data to process, so we use kafka as buffer.





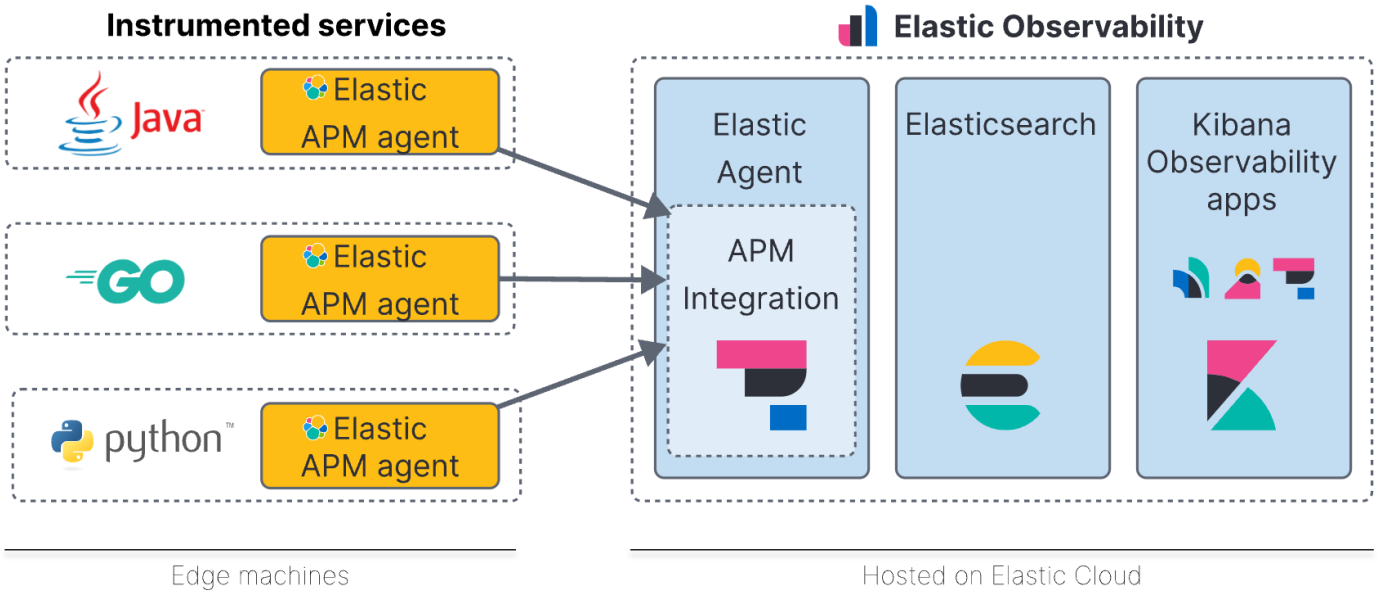

```
]#
]# nano /etc/elasticsearch/elasticsearch.yml
```

```
  GNU nano 2.9.8          /etc/elasticsearch/elasticsearch.yml

cluster.name: Fortified
node.name: chamber1
path.data: /var/lib/elasticsearch
path.logs: /var/log/elasticsearch
network.host: 192.168.25.100
http.port: 9200

node.roles: [ master, data_hot, data_content, ingest, transform, remote_cluster$

# Enable security features
xpack.security.enabled: true
xpack.security.enrollment.enabled: true
# Enable encryption for HTTP API client connections, such as Kibana, Logstash, $
xpack.security.http.ssl:
  enabled: true
  keystore.path: certs/http.p12
# Enable encryption and mutual authentication between cluster nodes
xpack.security.transport.ssl:
  enabled: true
```



Elastic APM

Elastic APM is an application performance monitoring system built on the Elastic Stack. It allows you to monitor software services and applications in real-time, by collecting detailed performance information on response time for incoming requests, database queries, calls to caches, external HTTP requests, and more.



Provisioning Elasticsearch

Managed service by Elastic.co

Self-hosted on Kubernetes using ECK Operator

Self-hosted on your cloud provider (virtual machine cluster)

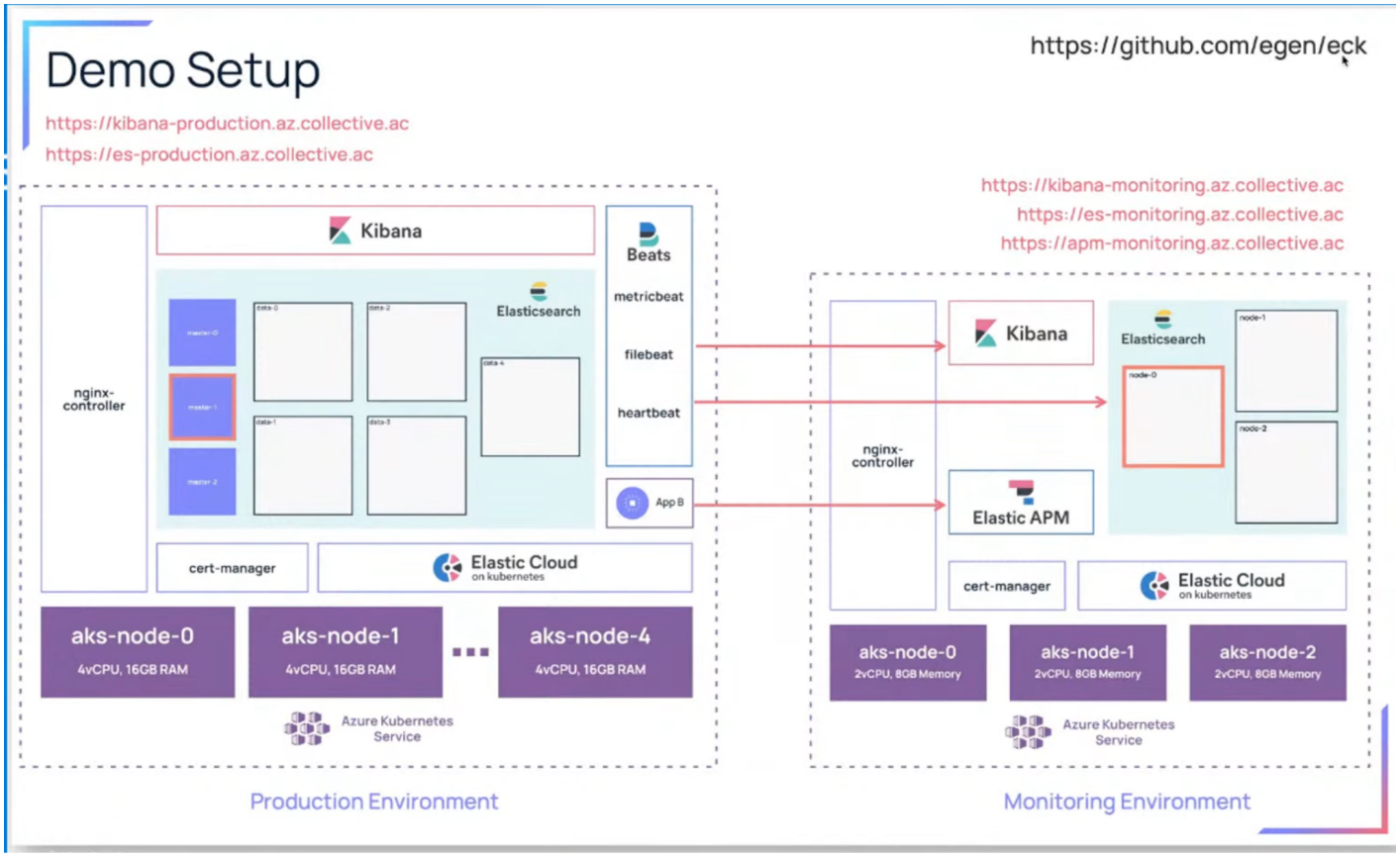
Monitoring Elasticsearch

Important Metrics to Monitor

- CPU , Memory, and Disk IO utilization of the nodes
- JVM Metrics: Heap, GC, and JVM Pool size
- Cluster availability (green, yellow, red) and allocation states
- Node availability (up or down)
- Total number of shards
- Shard size and availability
- Request rate and latency

Options:

1. Elastic Observability with Metricbeat, Filebeat (of course a separate monitoring ES cluster)
2. Prometheus + Grafana
3. Splunk
4. Datadog



Discovery

Discovery is the process by which the cluster formation module finds other nodes with which to form a cluster. This process runs when you start an Elasticsearch node or when a node believes the master node failed and continues until the master node is found or a new master node is elected.

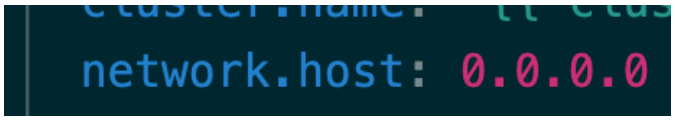
This process starts with a list of seed addresses from one or more seed hosts providers, together with the addresses of any master-eligible nodes that were in the last-known cluster. The process operates in two phases: First, each node probes the seed addresses by connecting to each address and attempting to identify the node to which it is connected and to verify that it is master-eligible. Secondly, if successful, it shares with the remote node a list of all of its known master-eligible peers and the remote node responds with its peers in turn.

Each seed hosts provider yields the IP addresses or hostnames of the seed nodes. If it returns any hostnames then these are resolved to IP addresses using a DNS lookup. If a hostname resolves to multiple IP addresses then Elasticsearch tries to find a seed node at all of these addresses.

```
discovery.seed_hosts:
   - 192.168.1.10:9300
   - 192.168.1.11 (1)
   - seeds.mydomain.com (2)
```

The network.host config is used to tell elasticsearch which IP in the server it will use to bind. Every service running in a server needs to bind to at least one IP, since servers can have multiple IPs, you can use 0.0.0.0 to tell the service to bind to all the IPs available on the server.

You can configure network.host so other machines can access the node. if the IP is for example 172.16.10.50 and you set network.host: 172.16.10.50, the elasticsearch will be accessible from outside the server to every other machine that can reach this IP.



Shard allocation filtering:
have several data cluster nodes, two of them with SSD storage. If we are looking for a fast response over one of our indexes, we can configure that their shards go only to the SSD data nodes. This concept is called shard allocation filtering.

```
./bin/elasticsearch -Enode.attr.size=medium -Enode.attr.disk=ssd
```

Or in elasticsearch.yaml config file:

```
node.attr.size: medium
node.attr.disk: ssd
```

Then we can specify our chosen index, twitter, to save on the data nodes with SSD attribute and size medium:

```
PUT twitter/_settings
{
 "index.routing.allocation.require.size": "medium",
 "index.routing.allocation.require.disk": "ssd"
}
```

specify the rack where each node of the cluster is running:

```
./bin/elasticsearch -Enode.attr.rack_id=rack_one
```
Then we need to specify in our master node that we are going to use rack_id as an attribute:
```
cluster.routing.allocation.awareness.attributes: rack_id
```

Force Awareness
one rack fails and then you might not have sufficient resources to host your primary and replica shards in only one rack. To prevent overloading in case of a failure, we can use force awareness.We can specify that replicas will be allocated only if both racks are available.
In the master node:
```
cluster.routing.allocation.awareness.attributes: rack_id
cluster.routing.allocation.awareness.force.rack_id.values:
rack_one,rack_two
```

Cluster Health:
cluster has several possible health statuses: green, yellow or red.a red status indicates that the specific shard is not allocated, yellow means that the primary shard is allocated but replicas are not, and green means that all shards are allocated.
```
GET _cluster/health
GET /_cluster/health/twitter
GET /_cluster/health/twitter?level=shards

GET /_cluster/allocation/explain
```

If the node is in yellow or red state, perhaps we need to change some configuration.configuring the twitter index specifying that its primary shard cannot remain on the node named “data-node1”.
```
PUT /twitter/_settings
{"index.routing.allocation.exclude._name": "data-node1 }
```

Backup and Restore:
In order to perform backups, elasticsearch offers us a snapshot API. You can take snapshots of a running index or the entire cluster, and then store that information somewhere else. elasticsearch will avoid copying any data that is already stored because of an earlier snapshot. Because of that is it recommended to take snapshots of your cluster quite frequently.

To save snapshots we need to create a repository.specify the possible paths for each repository inside the elasticsearch.yml on each of master and data nodes:

```
path.repo: ["/mount/backups", "/mount/longterm_backups"]
```
Then we can register a repo with a name:
```
PUT /_snapshot/my_backup
{
 "type": "fs",
```

```
 "settings": {
    "location": "/mount/backups"
 }
}
```

Back up data via a snapshot:

```
PUT /_snapshot/my_backup/snapshot_1?wait_for_completion=true
```

or specific indexes:

```
PUT /_snapshot/my_backup/snapshot_2?wait_for_completion=true
{
 "indices": "index_1,index_2",
 "ignore_unavailable": true,
 "include_global_state": false
}
```

check if the snapshot is created:

```
GET /_snapshot/my_backup/snapshot_1
```

restore the whole snapshot:

```
POST /_snapshot/my_backup/snapshot_1/_restore
```

restore specific indices from the snapshot:

```
POST /_snapshot/my_backup/snapshot_1/_restore
{
 "indices": "index_1,index_2",
 "ignore_unavailable": true,
 "include_global_state": true,
 "rename_pattern": "index_(.+)",
 "rename_replacement": "restored_index_$1"
}
```

delete the snapshot:

```
DELETE /_snapshot/my_repository/my_snapshot
```

Cross Cluster Search

To enable it we just need to specify the IP addresses of each cluster:

```
PUT _cluster/settings
```

```
{
  "persistent": {
    "cluster": {
      "remote": {
        "cluster_one": {
          "seeds": [
            "127.0.0.1:9300"
          ]
        },
        "cluster_two": {
          "seeds": [
            "127.0.0.1:9301"
          ]
        }
      }
    }
  }
}
```

Search in specific cluster:

```
GET /cluster_one:twitter/_search
{
 "query": {
   "match": {
     "user.id": "carloslannister"
   }
 }
}
```