Architectural Decisions:

1. Overview of Elastic Search architecture
2. Organizing data in elastic Search: indices and shards
3. resilient elastic search cluster
4. Sizing an elastic search cluster

Administration/Operational

1. Provisioning elasticsearch and kibana using ECK
2. Scaling elasticsearch cluster
3. ILM
4. Monitoring Elasticsearch cluster
5. Snapshot and restore

Architectural Decisions:

1. Apache Lucene:

open source search engine written in java

uses the concept of inverted index and edit distance for fuzzy search

harder to scale horizontally because of overhead

2. Shard:

Individual lucene instance running on its own

stores and index data: primary and replica

3. Replication:

Data is written first to primary shard and then replicated to one or more replica shards

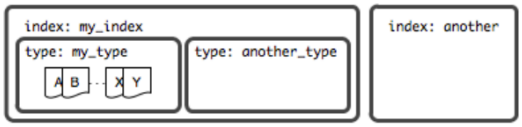
Elasticsearc takes care of promoting replica to primary when the primary hard fails

4. Schema:

A *schema* is a description of one or more fields that describes the document type and how to handle the different fields of a document.

The schema in Elasticsearch is a mapping that describes the the fields in the JSON documents along with their data type, as well as how they should be indexed in the Lucene indexes that lie under the hood. Because of this, in Elasticsearch terms, we usually call this schema a "mapping".

Conceptually, an Elasticsearch server contains zero or more indexes. An index is a container for zero or more types, which in turn has zero or more documents. To put it another way: a document has an identifier, belongs to a type, which belongs to an index. The figure below shows the documents `A`, `B`, `X` and `Y` of the type `my_type`, inside the index `my_index`.



5. Template:
   A template will be defined with a name pattern and some configuration in it. If the name of the index matches the template's naming pattern, the new index will be created with the configuration defined in the template. Elasticsearch has upgraded its template functionality in version 7.8 with composable templates

6. Mapping:
   Mapping is the process of defining how a document, and the fields it contains, are stored and indexed.
   Each document is a collection of fields, which each have their own data type. When mapping your data, you create a mapping definition, which contains a list of fields that are pertinent to the document. A mapping definition also includes metadata fields, like the _source field, which customize how a document's associated metadata is handled.

An example of mapping creation using the Mapping API:

```
PUT 'Server_URL/Index_Name/_mapping/Mapping_Name'
{
            "type_1" : {
                        "properties" : {
                                    "field1" : {"type" : "string"}
                        }
            }
}
```

In the above code:

- Index_Name: Provides the index name to be created
- Mapping_Name: Provides the mapping name
- type_1 : Defines the mapping type
- Properties: Defines the various properties and document fields
- {"type"}: Defines the data type of the property or field

Below is an example of mapping creation using an index API:

```
PUT /index_name

{
  "mappings":{
            "type_1":{
                        "_all" : {"enabled" : true},
                        "properties":{
                                    "field_1":{ "type":"string"},
                                    "field_2":{ "type":"long"}
                        }
            },
            "type_2":{
                        "properties":{
                                    "field_3":{ "type":"string"},
                                    "field_4":{ "type":"date"}
                        }
            }
  }
}
```

In the above code:

- ***Index_Name*** : The name of the index to be created
- ***type_1*** : Defines the mapping type
- ***_all*** : The configuration metafield parameter. If " ***true*** ," it will concatenate all strings and search values
- ***Properties*** : Defines the various properties and document fields
- ***{"type"}*** : Defines the data type of the property or field

7. index:

collection of json documents

a logical grouping of all the shards for a specific type, including primaries and replicas

e.g. orders in ecommerce website are placed into different shards part of an index

8. Alias:

Secondary name for a group of indices

Allows for reindexing and rollover without a downtime and code-change

e.g. indices created by month

9. Node:

Single elasticsearch instance: CPU + memory

A VM or container

A node can assume multiple roles at the same time: master, data, ingest, transform, voting-only, and more

node.roles: [master,data,ingest]

10. Cluster:

A collection of connected elasticsearch nodes

Identified by its cluster name

nodes communicate over tcp and from outside its a HTTP/restful api

Each node in a cluster knows about the other nodes

each node can accept a client request and forward that to the appropriate node

You can put external load balancer in front of elastic saerch cluster but internally if you even send request (search or indexing) to any one node, it will automatically figure out how to deal with othe rnodes internally

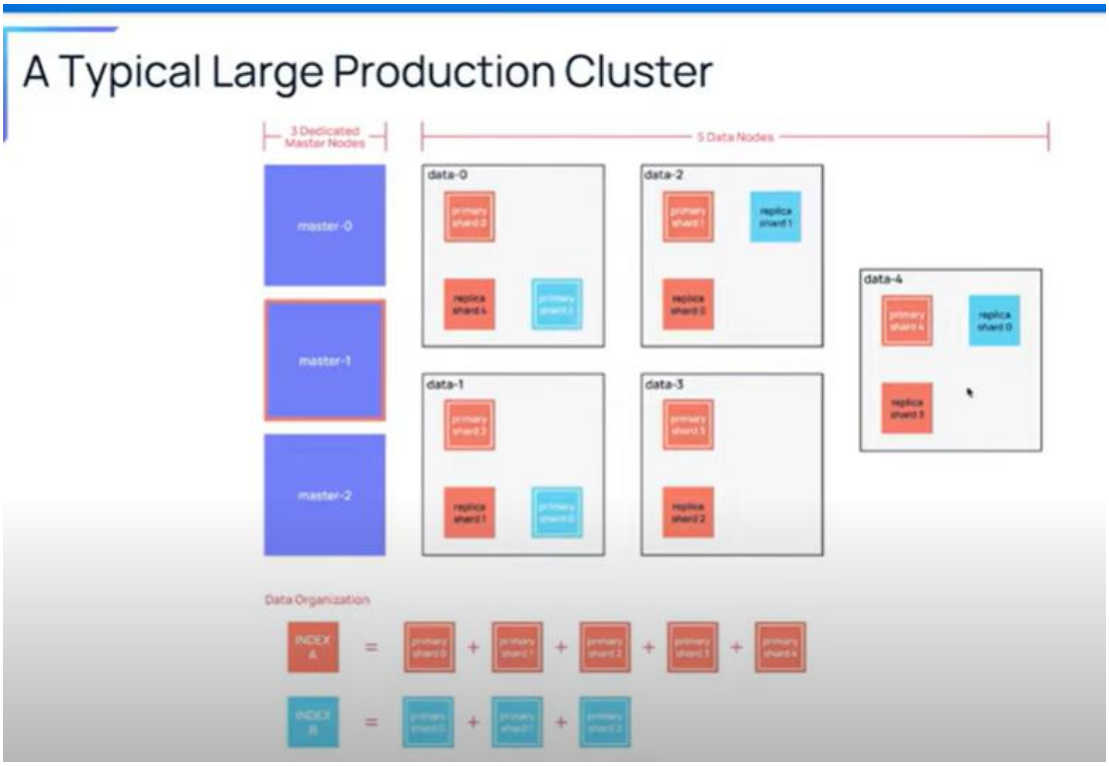
Cluster State:

- Node configuration
- Cluster setting
- Indices, mapping, settings
- Location of all shards

Cluster size:

- 1<= number of nodes <=no limit
- Large cluster (> 100 nodes)
  - More searchable data
  - Larger cluster state
  - Harder for master to manage
  - More operational issues
- Cross-cluster replication and cross cluster search are now available

A typical large production cluster



Master node:

- Only one master at a time
- In-charge of light weight cluster wide actions:
    - Cluster settings
    - Deleting or creating indices and settings
    - Adding or removing nodes
    - Shard allocation to the nodes
- A stable master node is required for the cluster health
- Since, an elasticsearch node can assume multiple node roles simultaneously
    - Small clusters: you can have same nodes assigned as data and master roles (and others)
    - Large cluster: you should have dedicated master and data nodes (separation of concerns)
- Dedicated master node don't need to have same compute resources as data nodes

Electing a master among master eligible nodes:

- There is a single elected master node within a cluster at a time
- Master election happens at
    - Cluster setup
    - When existing master node fails
- Elasticsearch uses Quorum based decision making for electing master node (avoiding a split-brain scenario)
- n/2 + 1 master eligible nodes are required to respond during the master election process.

- You can add or remove master eligible nodes to a running cluster. Elasticsearch will maintain voting configurations (responses from the master eligible nodes) automatically.
- You must not stop half or more of the nodes in the voting config at the same time, otherwise cluster will become unavailable.

Resilient Elasticsearch cluster:

- A resilient cluster requires redundancy for every required cluster component
- This means:
    - At least 3 master eligible nodes
    - At least 2 nodes of each role
    - At least 2 copies of each shard (one primary and one more replica)
- Small clusters
    - You can start with nodes having shared roles (data + master)
    - Minimum: 2 master-eligible nodes + 1 voting only master eligible tiebreaker node
    - Better: 3 master-eligible nodes
- Large clusters
    - Dedicated master eligible nodes = 3
    - Zonal redundancy with one master eligible node in each of the three zones.
    - Cross cluster replication

Sizing ElasticSearch Cluster:

- Sharding Strategy
    - Each elasticsearch index has one primary shard by default. Set at the index creation time.
    - You can increase the number of shards per index (max 1024) by changing index.number_of_shards. Requires reindexing.
    - Total shards = primary shards + replica shards
    - More primary shards
        - More parallelism
        - Faster indexing performance
        - A bit slower search performance
    - Aim for shard size between 10 GB and 50 GB. Better = 25 GB for search use cases.
    - Larger shards makes the cluster less likely to recover from failures (harder to move to other nodes when a node is going under maintenance)
    - Aim for 20 shards or fewer per GB of heap memory
    - For example, a node with 30 GB of heap memory should have at most 600 shards.
- How many shards?
    - Number of shards= (source data + room to grow) x (1 + 10 % indexing overhead)/desired shard size
    - Minimum storage required= source data x (1 + number of replicas) x 1.45
    - For a cluster handling complex aggs, search queries high throughput, frequent updates.

Node config = 2 vcpu cores and 8 GB of memory for every 100 GB of storage
    - Set ES node JVM heap size to no more than 50 % (32 GB max) of the physical memory of the backing node.

Q. Let's assume you have 150 GB of existing data to be indexed

- Number of primary shards (starting from scratch or no growth) = (150 + 0) x 1.10 / 25 = 7 shards
- Number of primary shards (with 100% growth in a year) = (150 + 150) x 1.10 / 25 = 13 shards
- Remember you cannot change the number of primary shards without reindexing, so pre plan.
- Going with minimum 3 nodes (master + data). Assuming, we would have 1 replica per shard.
    - Minimum storage required= 150 x (1+1) x 1.45 = 435 GB
    - Compute and memory= 435 x (2 cpu core and 8 GB memory)/ 100 = 9 vcpu and 35 GB of memory
    - 3 nodes each having 3 vCPUs, 12 GB memory and 150 GB disk storage.
    - With 12 GB of RAM you can set the JVM heap size to 6 GB(==max 120 shards per node, 360 shards in total)

Multi tenancy:

- A cluster per tenant (better when you have < 5 large tenants)
    - Best isolation of network, compute, and storage
    - Needs high degree of automation to create and manage these many clusters
    - Monitoring and operations are going to be difficult
- An index per tenant
    - Logical isolation with access control
    - But quickly runs into the over sharding problem: too many smaller shards -> large cluster state
- A document per tenant (better when you have 10s – 100s of tenants where onboarding is self service)
    - Makes scaling super easy and best suite for search use cases
    - Need to have a tenant ID field per document
    - Need to scope search and indexing requests to the tenant
    - Need to have strong access control mechanism as there is no network, compute or storage isolation

Administration/Operational

Provisioning Elasticsearch

- Managed service by elastic.co

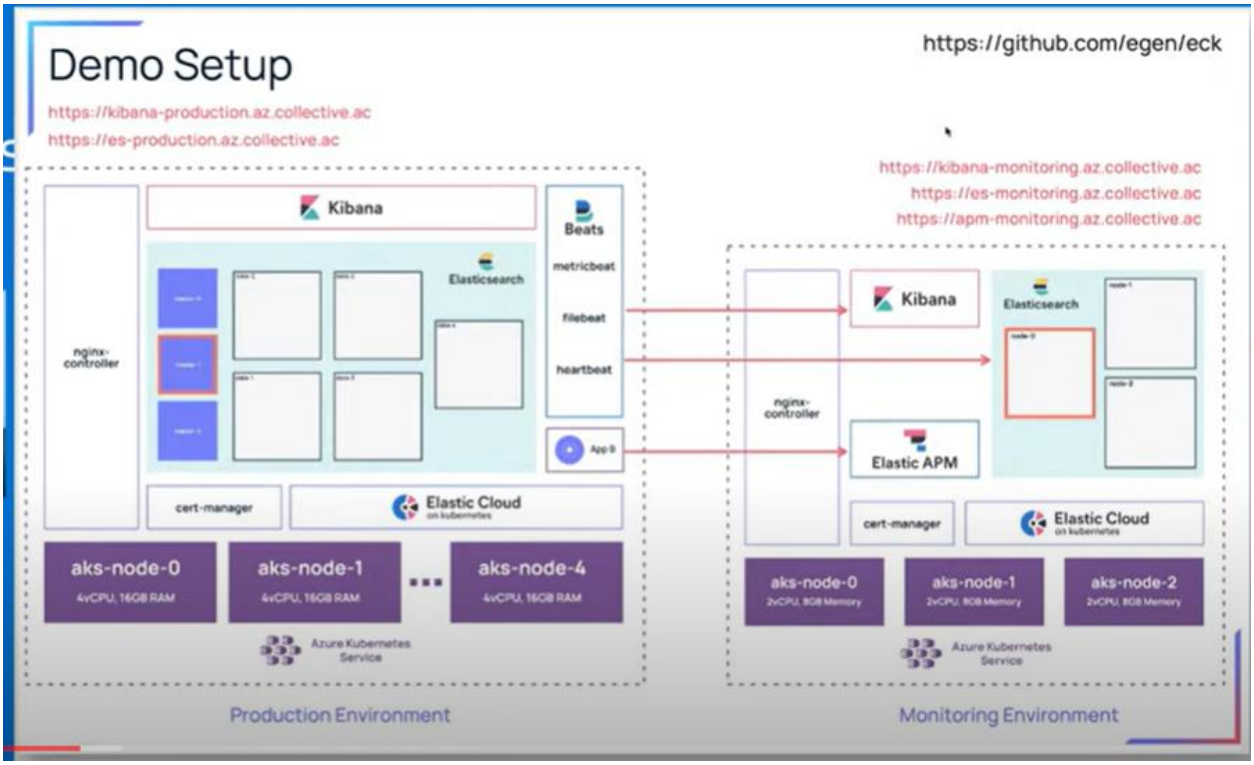
  Elastic cloud -> AWS, GCP

- Self hosted in Kubernetes un=sing ECK operator
- Self hosted on your cloud provider (virtual machine cluster) -> AWS, GCP

Monitoring Elastic Search

- Important metrics to monitor
    - CPU, memory and disk IO of the nodes
    - JVM metrics: heap, GC, JVM pool size
    - Cluster availability and allocation states
    - Node availability
    - Total number of shards
    - Shard size and availability
    - Request rate and latency

- Options:
    - Elastic observability with metricbeat, filebeat (ofcourse a separate monitoring ES cluster)
    - Prometheus + Grafana
    - Splunk
    - Datadog



Handling data growth

- 1. Aim for shard size between 10 GB and 50 GB. Better 25 GB for search use cases.

Options:

- Increase primary shards by updating index settings. Reindexing will be required.
- Rollover the index using ILM.
  - Start with the default setting of 1 primary shard per index.
  - Use the rollover index API to create new index when the current index goes past a
    - Age (fixed time interval)
    - Size
    - Count

    e.g. orders-2021-01, orders-2021-02 and so on

- 2. Aim for 20 shards or fewer per GB of heap memory
  Options:
  - Data retention: if possible, delete/archive the older shards as per a retention policy. Goes best with ILM.
  - Use shrink index API to shrink an existing index into a new index with fewer primary shards:
    - Requested number of primary shards in the target index must be a factor of the number of shards in the source index
    - E.g. an index with 8 primary shards can be shrunk into 4,2 or 1 primary shards or an index with 15 primary shards can be shrunk into 5,3 or 1.
  - Adding new data nodes to the cluster
    - Scale up by adding master-inelligible nodes (data, ingest, transform, etc) only

ILM: Index Lifecycle Management:

Managing indices:

- Automatically manage indices according to performance, resiliency, and retention requirements using simple ILM policies.
- Index lifecycle phases:
  - Hot: actively being updated and queried
  - Warm: No longer being updated but actively being queried (logs and similar immutable objects)
  - Cold: no longer being updated and is queried infrequently. Okay for searches to be slower
  - Frozen: no longer being updated and is queried rarely. Okay for searches to be extremely slow.
  - Delete: no longer needed and can be safely removed
- ILM policies specifies which phases are applicable, what actions are performed in each phase and when it transitions between phases.

Data Tiers:

- Specialized node.roles roles to tie data nodes to a specific data tiers:
    - data_hot
    - data_warm
    - data_cold
    - data_frozen
- you can attach best performing hardware to the data_hot nodes with SSD while using slower and cheaper hardware for the other nides in the list.
- Helps in tremendous cost saving

ILM Policy Examples:

Q. Why re-indexing is required?

When you want to update the number of shards in an already created index, you cannot change it, you need to reindex. Steps that is followed: Create new index with more shards -> move the data from old index to new index -> updated the alias to point to new index -> deleted the old index

But this is cumbersome, so other way to do this is using rollover.

Rollover will create a new index where new data will be written, the old index is still there but no new data will be written in that.

Now this also needs manual task, so other better approach is to use ILM.

```
#ilm
PUT _ilm/policy/order_archive_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {
          "rollover": {
            "max_age": "30s"
          }
        }
      },
      "cold": {
        "min_age": "0ms",
        "actions": {
          "allocate": {
            "number_of_replicas": 0
          },
          "readonly": {}
        }
      }
    }
  }
}
```

```
PUT _index_template/order-template
{
  "index_patterns": ["order-*"],
  "template": {
    "settings": {
      "number_of_shards": 1,
      "number_of_replicas": 1,
      "index.lifecycle.name": "order_archive_policy",
      "index.lifecycle.rollover_alias": "orders"
    },
    "mappings":{
      "properties":{
        "vendor":{
          "type":"keyword"
        }
      }
    }
  }
}
```

```
#important setting for the purpose of the demo. default
  poll_interval is 10minutes.
PUT _cluster/settings
{
  "transient": {
    "indices.lifecycle.poll_interval":"1s"
  }
}


PUT order-000001
{
  "aliases": {
    "orders": {
      "is_write_index":true
    }
  }
}



PUT orders/_doc/1
{
  "vendor":"amazon"
}
```