Td-aggregator Replacement Design Doc

As we aim to deprecate td-aggregator and aws msk, we should have a basic knowledge of what td-aggregator is and which role does it play in our log ingestion pipeline and summarize the pros and cons of current architecture. Then we can reach a consensus on our objectives of the deprecating plan. At the end, we will give a migration plan.

What does td-aggregator do?

td-aggregator is a fluentd service, which usually acts as a log-collector in a data pipeline. **td-aggregator** is distributed and reliable and it moves logs from wish services to amazon msk(aka. kafka), following some elaborate routing rules. It should be noticed that we also have a fluentd service, named **td-agent**, which is responsible for collecting logs as an agent in each edge node with wish services and sending the logs to **td-aggregator** service. **td-agent** is not in the scope we are going to deprecate. For a better understanding of this architecture, you can refer to this slide Tahoe Log Ingestion Pipeline 2.0.

How is td-aggregator deployed?

td-aggregator is deployed as k8s pods in these clusters

env	cluster	kind	replicas	pod size	listening port
prod	app-01-prod-k8s.local	StatefulSet	3	4xlarge	8889
prod	app-02-prod-k8s.local	StatefulSet	3	4xlarge	8889
stage	app-03-stage-k8s.local	StatefulSet	1	2xlarge	8889
stage	app-04-stage-k8s.local	StatefulSet	1	2xlarge	8889
dev	app-05-dev-k8s.local	StatefulSet	1	2xlarge	8889
prod	app-06-prod-k8s.local	StatefulSet	3	4xlarge	8889
prod	app-07-prod-k8s.local	StatefulSet	3	4xlarge	8889

Replace td-aggregator with kafka

td-aggregator plays a tremendous role in log collecting because it supplies some flexible and comprehensive features(with various plugins), including but not limited to:

- It supports heterogeneous community-contributed plugins connecting dozens of data sources and data outputs. If we want to change our data sources or outputs, we don't need to replace the fluentd service, but only need to use another suitable plugin.
- It supports multiple workers in one fluentd instance in a node. Each worker corresponds to a process in this physical machine and can be configured to be guarded by a supervisor process. This feature utilizes more computing resources and brings up high performance for high traffic.
- It provides the **At-Least-Once** semantics, we don't worry about data loss problems in most cases.

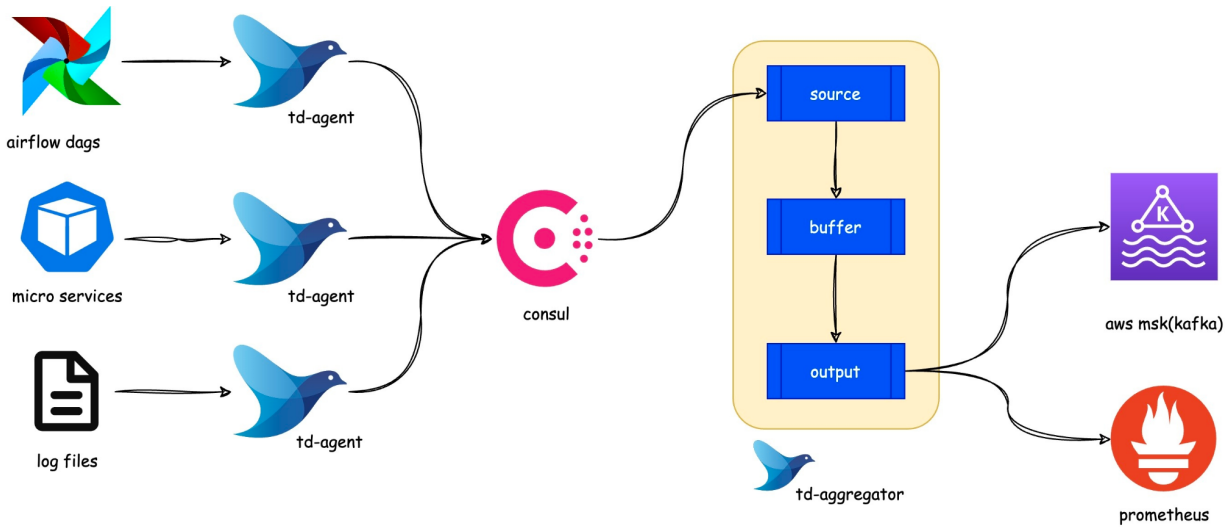
Though td-aggregator provides these good features, it also has some drawbacks that motivate us to replace it

- If our platform is binded to apache kafka, we don't need such a sophisticated service to support our log collecting.
- At-Least-Once semantics is not good enough. Not only do we want to prevent data loss, but also we want to reduce log duplication.
- **td-aggregator** service is stateful and has a SPOF problem as it relies on a disk space to buffer events before delivery in case of downstream failure. If a td-aggregator instance fails to start or a disk failure has occurred, the data it buffered will be lost.

Based on the considerations above, we want to find a better middleware to replace **td-aggregator** and we think **Apache Kafka + MirrorMaker** is a good candidate option.

For the limit of td-agent, we need to set up a new kafka cluster to persist log events and then send these data to the destination kafka clusters. But, what is the difference between the new architecture and the original td-aggregator architecture?

First let us see the original data flow diagram with td-aggregator:



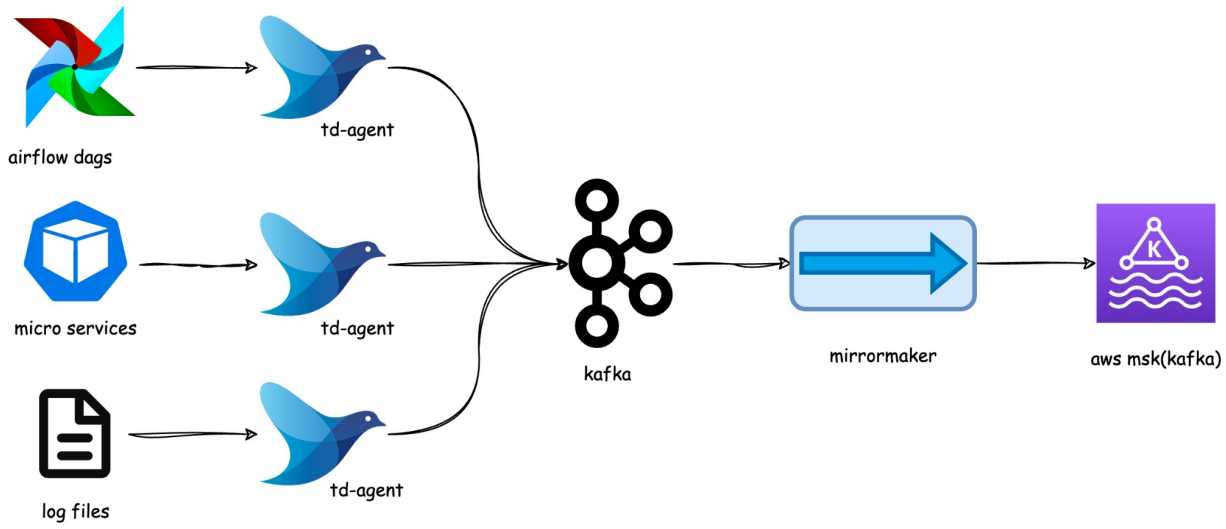
Each td-aggregator service registers itself at a service discovery service called consul and consul routes requests from td-agent to td-aggregator. The td-aggregator service takes responsibility for

- receive data through its source, which is a listening server socket
- persist the log data to the disk
- send the log to the destination, which is always a kafka cluster in our scenario

Further, we split this process into two sub steps

- receiving log
- sending log

A kafka cluster and a kafka mirrormaker can assume these two responsibilities respectively. Thus, the new designed data flow diagram can be designed like this:



A kafka cluster is specifically utilized to receive log data and persist the data in a partition reliably. If one kafka broker is down, the data won't be lost as long as it is replicated in the cluster with a replication factor larger than 1.

A kafka mirrormaker is utilized to maintain a replica of an existing kafka cluster. It uses a kafka consumer to consume messages from the source cluster, and re-publishes those to the target cluster. A kafka mirrormaker corresponds to the output plugin in td-aggregator.

Above is the design idea for refactoring. Next we discuss the feasibility of this change. We need to delve a little deeper into the working logic of td-agent and td-aggregator.

Functional feasibility

Below are snippets from configurations of td-agent and td-aggregator.

td-agent

```
Unset
<match td.**> //when matches an event then just forwards it
  @type forward
  @id tdlog-agent

  expire_dns_cache 0
  dns_round_robin true
  heartbeat_type none
  connect_timeout 20
  <server>
    host %s.service.consul
    port 8889
  </server>

  <buffer tag>
    @type file_single
    chunk_limit_size 32m
    flush_at_shutdown true
    flush_interval 20s
    flush_thread_count %d
    retry_max_interval 3600s
    retry_forever true
    retry_randomize true
    total_limit_size 12g
  </buffer>
```

```
</match>
```

We can see that td-agent does nothing but forwards events to the downstream topological node, which is a service discovery service called consul. If we use a kafka cluster as the downstream, the service discovery service is still required to locate the bootstrap broker though kafka brokers can be found through zookeeper by a controller broker internally. So for td-agent, we need to replace the forward plugin with a kafka plugin.

td-aggregator

td-aggregator uses the match directive to define output logic. Some match directives deliver the events directly, and some redirect events to other match sections. In fluentd, relabel output plugin is used to route the matched events and then the respective label directive takes care of these events. Let's look at the match directives and relabel directives and their actions one by one.

<match td.*.*.{tahoe_only, tahoe_prod}>

→ relabel events to tahoe_only label.

<match td.*.*.tahoe_stage>

→ relabel events to tahoe_stage label.

<match td.{some table names}>

→ relabel events to tahoe_only label.

<match td.**>

copy events, and then

- store in prometheus(for *fluentd_deprecated_status_num_records_total* metric)
- rewrite tag for exceptional events

<match exception.** sherlock.**>

receive exception logs and store them in prometheus(for *fluentd_output_status_num_records_total* metric)

Above are all match sections, we can see some match sections relabel events to other sections, including tahoe_only, tahoe_prod, tahoe_stage, etc. In td-aggregator, these are relabel sections

<label @tahoe_stage>

- **<match td.*.*.tahoe_stage>**
  - rewrite tag to td.{database}.{table}
- **<match "td.#{ENV['CRITICAL_TABLES']}">**
  - store to kafka(stage, one topic per table)
  - store to prometheus(for *fluentd_output_status_num_records_total* metric)
- **<filter td.**>**
  - extract database name and fill it in __db__topic__name field for events combination. The __db__topic__name is like td.${database}.combined
- **<match td.**>**

  for the rest:

  - store to kafka(stage, one topic per db)
  - store to prometheus(for *fluentd_output_status_num_records_total* metric)

<label @tahoe_only>

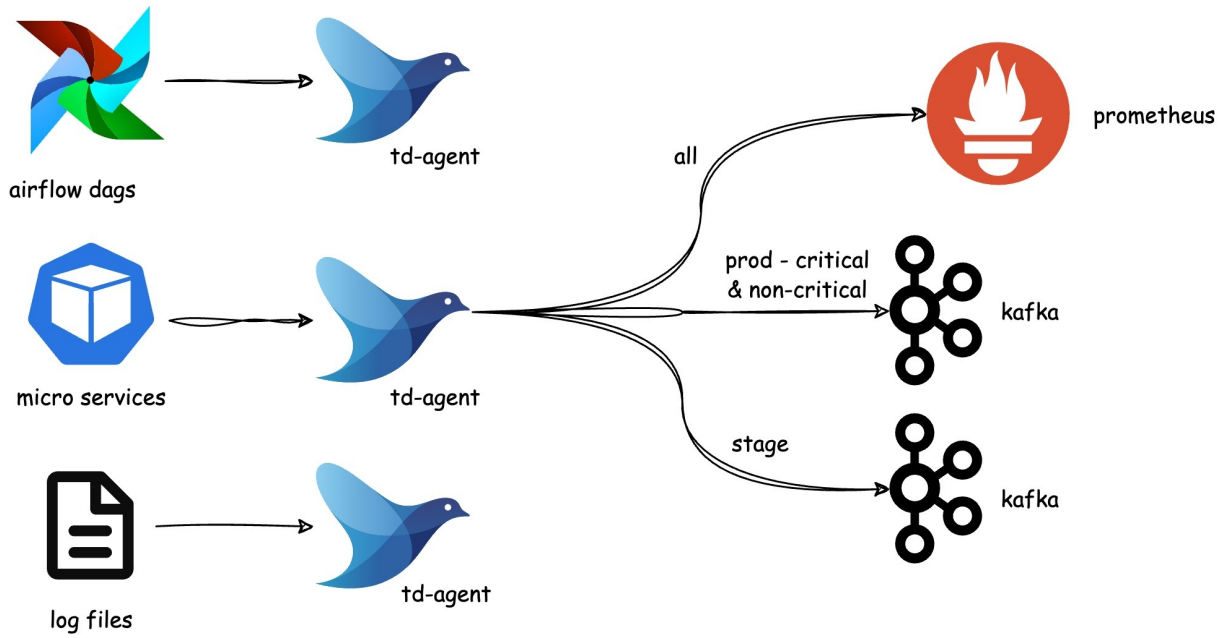
- **<match td.*.*.{tahoe_only,tahoe_prod}>**
  - rewrite tag to td.{database}.{table}
- **<match "td.#{ENV['CRITICAL_TABLES']}">**
  - store to kafka(prod, one topic per table)
  - store to prometheus(for *fluentd_output_status_num_records_total* metric)
- **<filter td.**>**
  - extract database name and fill it in __db__topic__name field for events combination. The __db__topic__name is like td.${database}.combined
- **<match td.**>**
  - store to kafka(prod, one topic per db)
  - store to prometheus(for *fluentd_output_status_num_records_total* metric)

The configuration is lengthiness and complicated, but it can be summed up in the following points

- store in prometheus and kafka
- combine non-critical events into one topic per database

If we want to replace td-aggregator with kafka, those features should be satisfied as well.

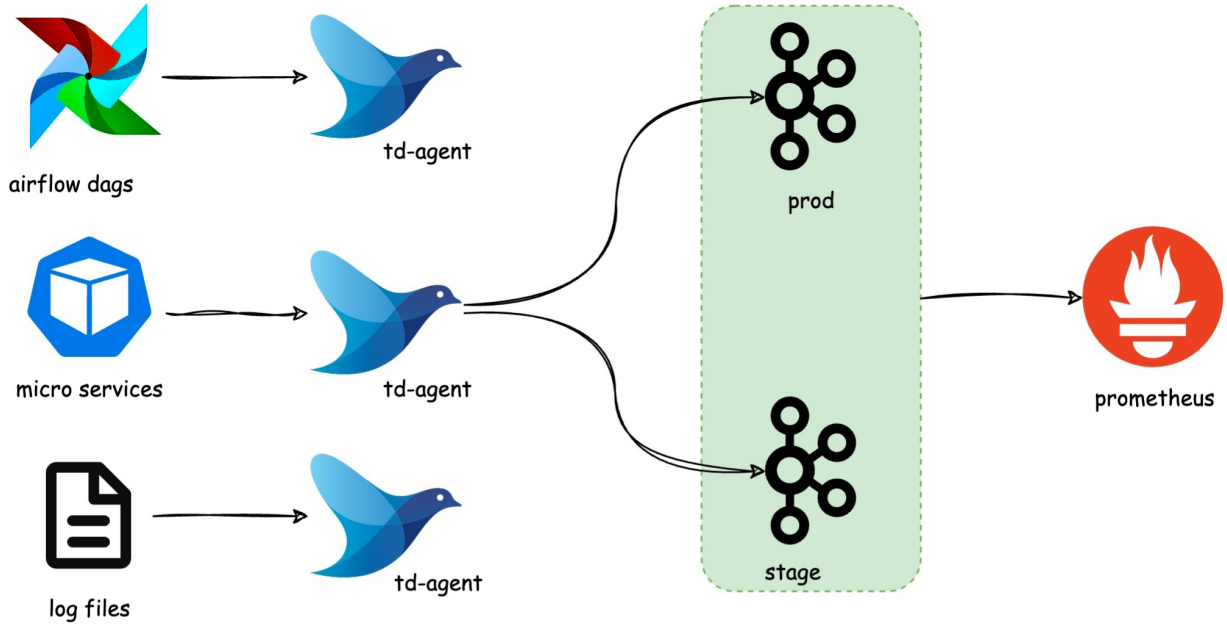
The first option is to add the routing rules into td-agent.



Then events will be delivered to

- Kafka
  Prometheus

The second option is to let td-agent deliver events directly into kafka, and the kafka cluster expose metrics to prometheus through a jmx exporter [prometheus-jmx-kafka]



Then events will be delivered to

- Kafka

The first option is more flexible and the second option is more clean and concise for td-agent.

I prefer the Second option and we must consider the remaining problem: should the non-critical tables be combined before they are sent into kafka?

Whether to defer the combination actions of non-critical tables depends on subsequent components(like mirrormaker or other tools). Using MirrorMaker we can merge different topics into one. This is achieved by implementing a custom message handler which can change every mirrored message on the fly.

The final pipeline can be designed like this

- td-agent outputs all events to the self-managed kafka cluster one topic per table.
  - outputs to prod cluster for events ends with tahoe-only, tahoe-prod suffix
  - outputs to prod cluster for tags in the whitelist
  - outputs to stage cluster for events ends with tahoe-stage
  - exports to prometheus
- mirrormaker replicate to the aws msk cluster
  - merge all non-critical topics

We are not going to bring actions other than simple outputs to td-agent, so the td-agent only needs to choose an appropriate topic to send events to instead of caring about other business logics.

From the analysis above, we think it is feasible to replace td-aggregator with kafka. In addition to proving that it is functionally feasible, we also need to explain some of the issues involved in deployment, such as network, security, etc.

Deployment feasibility

security

Let's first look at the security issues between td-agent and self-managed kafka cluster.

Security is an important issue for services deployed in the public open environment. These are a couple of methods to keep our services and data safe, including authentication, encryption of data and restriction of the connection to the public internet. In this section, we will illustrate the security issues in a kafka cluster.

As we know, the kafka community adds a number of features to increase security in a kafka cluster. There are two main aspects to kafka's security

- Authentication: Authentication is a mechanism to verify the identity of a connection to a kafka broker. It typically includes two kinds of connection

        - client-broker connection: Not only the identity of the broker is verified, but also the client's.
        - inter-broker connection: just like client-broker authentication.
    - Encryption: Encryption means data transferred between brokers and clients, between brokers, or between brokers and tools are encrypted using SSL

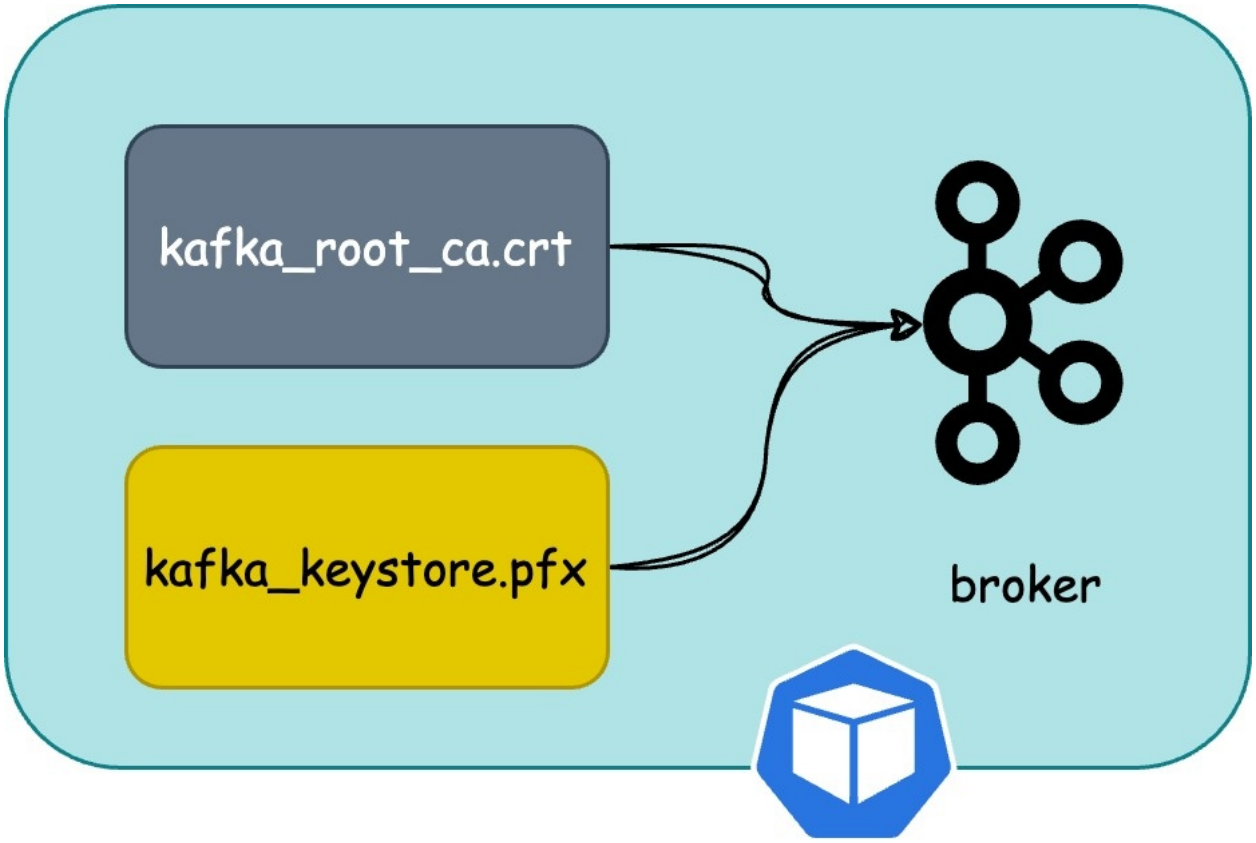
In most cases, encryption using SSL is necessary and authentication can have many choices based on the authentication mechanisms

- You only use SSL for both authentication and encryption
- You can use SASL protocol for authentication(with SSL for encryption). Kafka supports the following SASL mechanisms
    - SASL/GSSAPI
    - SASL/PLAIN
    - SASL/SCRAM-SHA-256, SCRAM-SHA-512
    - SASL/OAUTHBEARER

Currently SASL_SSL with SCRAM-SHA-512 mechanism is used in tahoe aws msk cluster and SSL is used in wish k8s kafka cluster. If an extra kafka cluster is added between td-agent and msk, td-agent, as a kafka client, should be able to use SSL for security. In this case, the client, including consumer and producer, must be configured like the following(just an example, not totally completed):

```
Unset
security.protocol=SSL ssl.truststore.type={{truststore type}}
--JKS(default),PKCS12,PEM
ssl.truststore.location={{truststore file}}
ssl.truststore.password={{truststore file password}}
ssl.keystore.type={{keystore type}} --JKS(default),PKCS12,PEM
ssl.keystore.location={{keystore file}} ssl.keystore.password={{keystore file
password}} --not support for PEM format
```

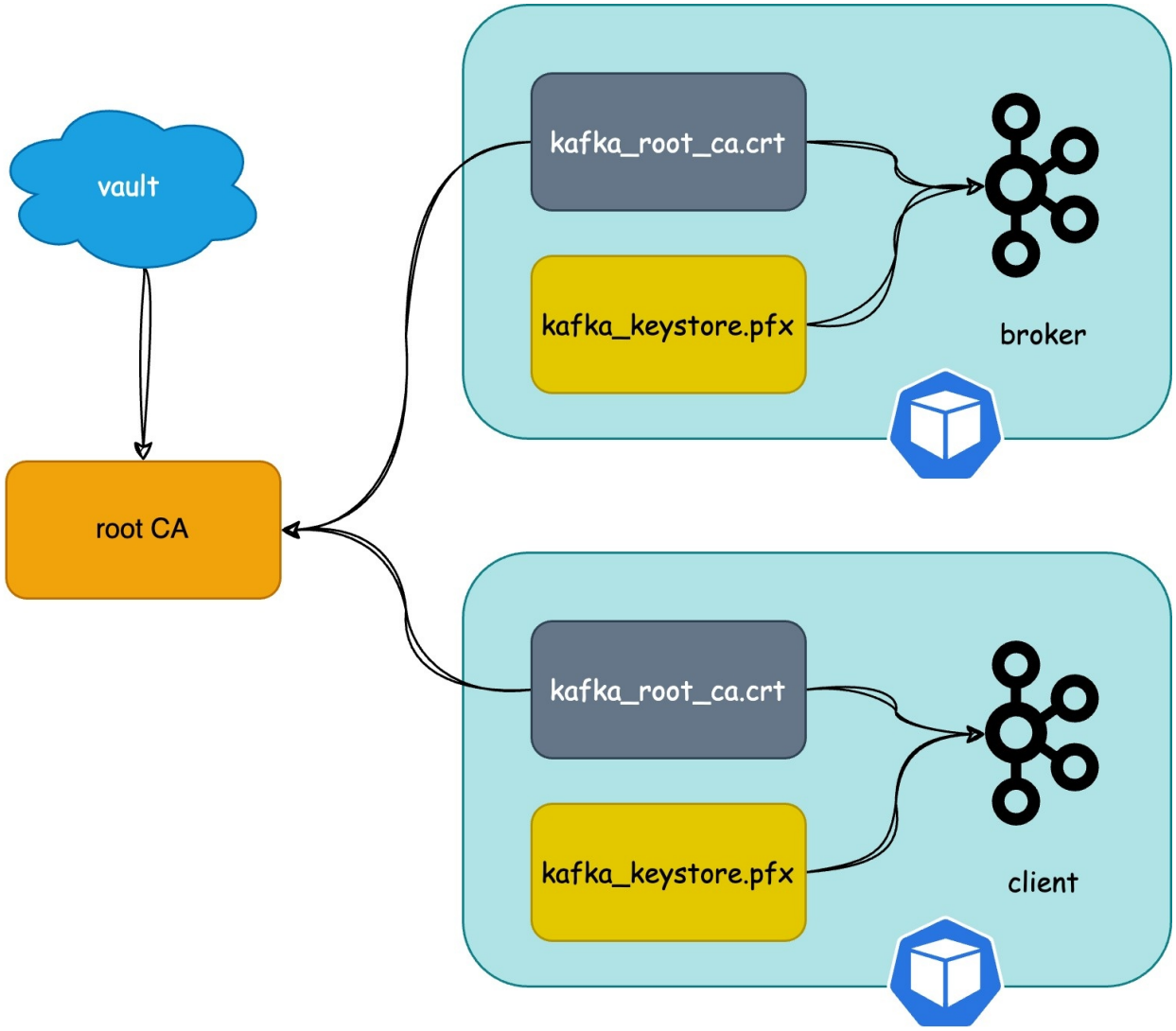
We see a truststore and a keystore is necessary. It should be noticed that creating secret key files and certificate files is non-trivial in the k8s deployment environment. In current deployment, each kafka cluster in k8s is deployed with a vault sidecar. The vault sidecar will prepare secret keys, certificates and ca’s certificates for kafka brokers in the same pod.



In the above picture, **kafka_root_ca.crt** is the certificate of the root CA. With using the root CA's certificate as the truststone, all certificates issued by the root CA or its intermediate CA will be trusted by this broker. **kafka_keystore.pfx** contains secret keys and certificates of this broker.

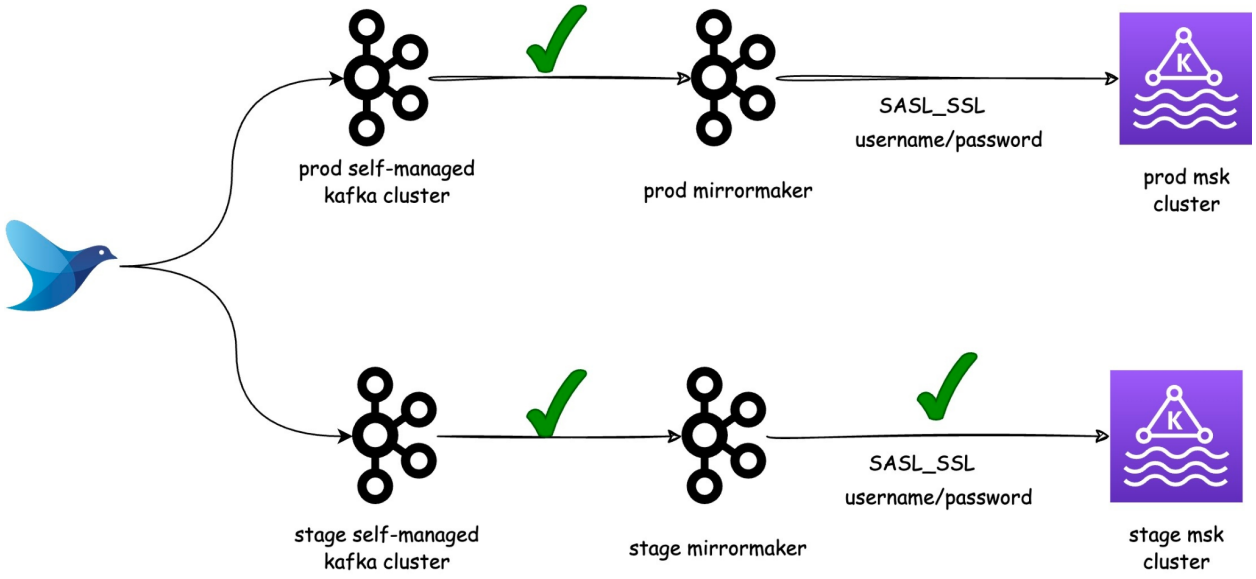
In the client side, a truststore that holds onto certificates is necessary. If client authentication is required, then a keystore is needed as well. All keys and certificates are generated by vault like in the broker side.

In order to use a SSL connection between kafka clients and brokers, the corresponding root CA must be added to the truststore of the peer. In current vault deployment, each entity adds the root CA of their certificate into their trustone, so if each client and broker add the same root CA in their truststore, they can trust each other.



Because all certificates are signed under a common mount path pki/, so they are sharing the same root CA and they will trust each other.

Then let's look at the security issues between mirrormaker and the aws msk cluster. The consumer of mirrormaker should be configured to use SSL as we describe above. The producer of mirrormaker can use SASL_SSL protocol to aws msk like td-aggregator did before.

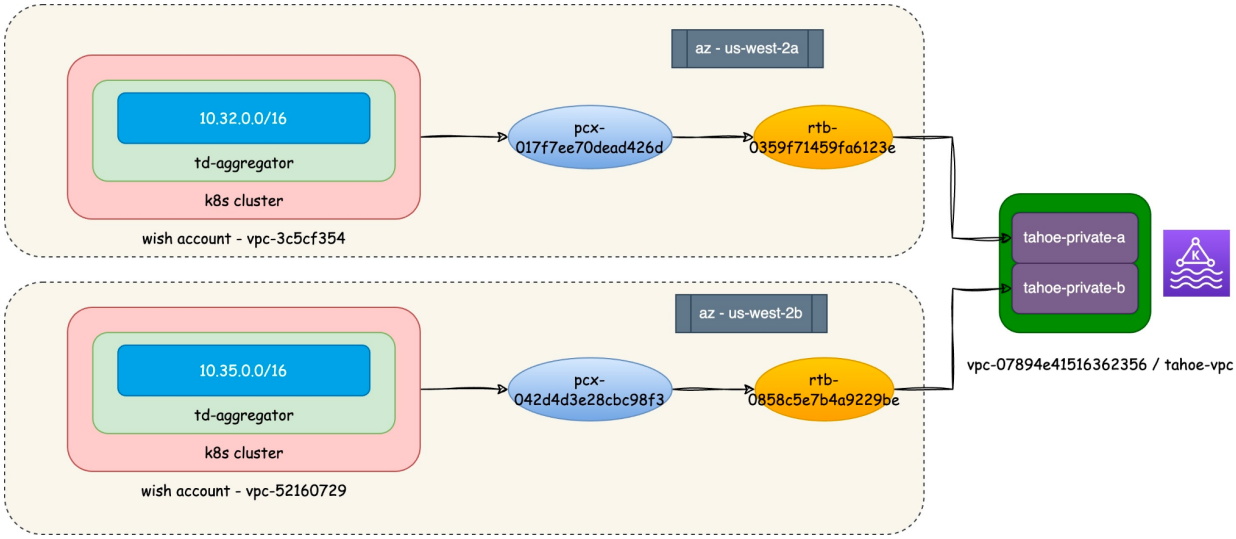


So for a mirrormaker, it should use two different security ways to ensure safety.

Through the above analysis, we have proved that it is security feasible to replace td-aggregator with extra kafka-cluster and mirrormaker.

network

In current architecture with td-aggregators, this network topology is like

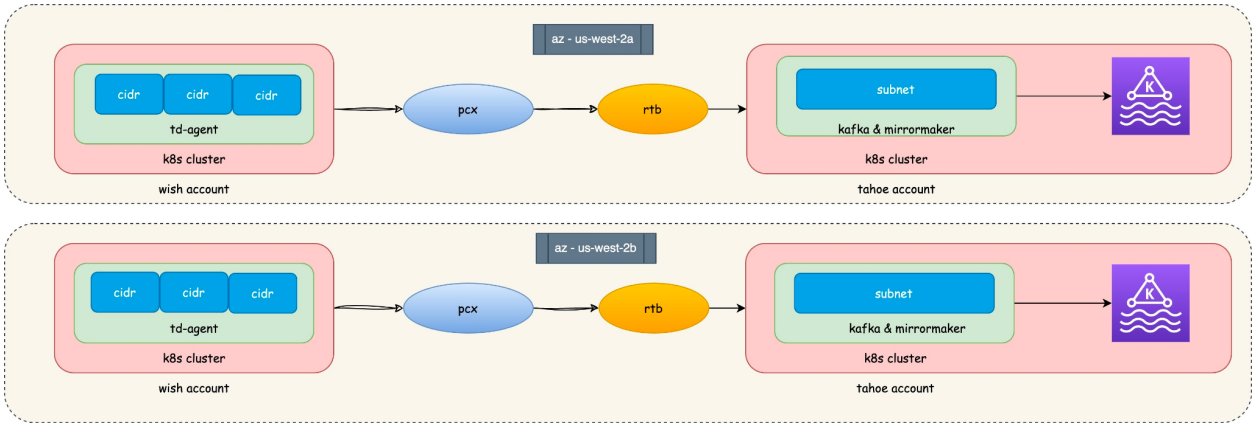


Network across k8s clusters is omitted here. We can see that there are three vpcs of two accounts involved

vpc	account	subnet
vpc-3c5cf354	951896542015	10.32.0.0/16
vpc-52160729	951896542015	10.35.0.0/16
vpc-07894e41516362356 | tahoe-vpc	343388184567	tahoe-private-a, tahoe-private-b

If we deprecate td-aggregator and add an extra self-managed kafka cluster and a kafka mirrormaker service, we should

- create new subnet(s) for the new kafka cluster
- connect subnets where td-agent is located to subnet of the new kafka cluster
    - create a VPC peering connection between two VPCs(current this exists now)
    - update the route tables for the VPC peering connection
        - update all cidr blocks of td-agent in the route table
        - update all cidr blocks(subnets) of kafka in the route table
        - update the security groups
- connect subnets where the kafka mirrormaker is located to the subnet of the tahoe aws msk.
    - it's trivial since they are deployed in the same VPC of tahoe account



Migration Plan

We have proved that it is feasible to deprecate td-aggregator. In this section, we give a detailed migration plan.

1. First of all, we must set up all necessary clusters, including kafka clusters and kafka mirrormakers.
2. refactor td-agent service
3. Migrate traffic to the new cluster
    1. replicate traffic to the new cluster.
    2. verify data integrity.
    3. deprecate the old cluster or services.

Cluster Setup

Kafka

We totally need two self-managed kafka-clusters: a prod cluster and a stage cluster. Although there has been some practice of how to build a kafka cluster on k8s, we still list some key configuration of the deployment.

Hardware inventory:

	cluster	instances	resource
prod kafka cluster	app-21-tahoe.k8s.local, app-22-tahoe.k8s.local	3 per zone, total 2*2*3=12	m5.2xlarge
stage kafka cluster	app-03-stage.k8s.local, app-04-stage.k8s.local	2 per zone, total 2*2*2=8	m5.xlarge

Mirrormaker

We deploy mirrormaker to any pods with kafka installed since Kafka distribution comes with connect-mirror-maker.sh script that is bundled with the Kafka library that implements a distributed Mirror Maker cluster. (However, the best practice is to locate mirrormaker processes as close as possible to the target clusters.)

Each mirrormaker process is started up with two configuration files, a consumer properties file and a producer properties file. Moreover a whitelist argument is necessary to filter source topics.

A sample consumer configuration file is

```
Unset
bootstrap.servers=localhost:19092
group.id=test-consumer-group
```

If the source cluster is using a SSL protocol, these configurations are needed as well

```
Unset
security.protocol=SSL
ssl.truststore.location=/var/private/ssl/client.truststore.jks
ssl.truststore.password=test1234
ssl.keystore.location=/var/private/ssl/client.keystore.jks
ssl.keystore.password=test1234
```

And a sample producer configuration file is

```
Unset
bootstrap.servers=localhost:29092
```

If the target cluster is using a SASL_SSL protocol, these configurations are needed as well

```
Unset
security.protocol=sasl_ssl,
sasl.mechanism=SCRAM-SHA-512,
ssl.ca.location=/etc/ssl/certs/,
sasl.username={{ $stage_kafka_username }},
sasl.password={{ $stage_kafka_password }}
```

then run the mirrormaker script

```
Unset
bin/kafka-mirror-maker.sh --consumer.config config/consumer.properties
--producer.config config/producer.properties --whitelist '.*' --message.handler
com.wish.log.tools.TestTopicHandler
```

Here we aim to replicate all topics in the source cluster to the remote cluster. We also provide a customized message handler class called TestTopicHandler, which is used to combine all non-critical topics by rewriting the target topic field in a producer record.

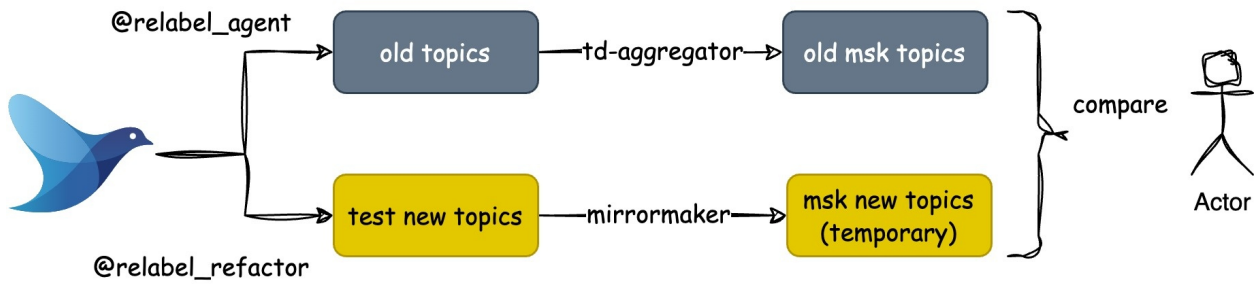
td-agent refactor

We can't refactor td-agent service all at once otherwise it may impact on the online running log collecting. We split the refactor into two steps

- support copying logs to the cluster as a new data flow

```
Unset
<match td.**>
    @type copy
    @id copy_td
    copy_mode shallow

    <store>
        @type relabel
        @id relabel_agent
        @label @agent
    </store>

    <store>
        @type relabel
        @id relabel_refactor
        @label @refactor
    </store>
</match>

<label @refactor>
    {{refactored configuration}}
</label>

<label @agent>
    {{original td-agent's configuration}}
</label>
```

The @rrelabel_agent is the origin routing configuration of td-agent. The @relabel_refactor label routes events to the new kafka cluster. It can refer to the configuration of td-aggregator.



- After the running is stable, remove legacy routing rules from the configuration.

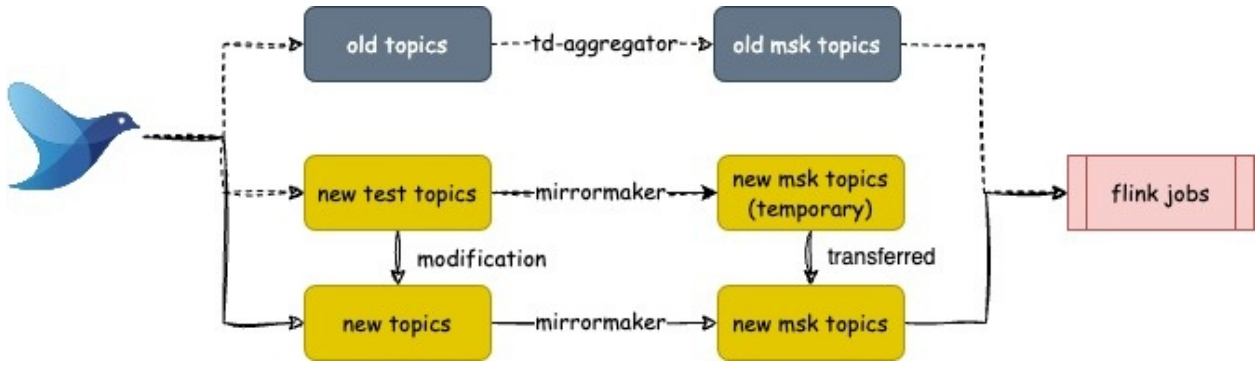
Traffic Migration

From the above steps, we can create some topics at the remote topic cluster(i.e. aws msk) with the same data as that of current topics by mirrormaker. The last step is to migrate our ingestion flink jobs from using current topics to use the new ones. Since we don't change the log distribution, we need not to change the main logic of these jobs. We only need to

- change the consumer configuration to use the new topic patterns
- restart the jobs

It is crucial to avoid log loss and duplication. What i recommend to do is to

- first verify our refactor works(i.e., the new topics and old topics have the same input size)
- create some new remote topics in the msk(current is empty) and add those topics into ingestion flink jobs.
- remove legacy configurations from some td-agents and and modify topic in @relabel_refactor to use new created empty topics and then restart td-agent



As we remove legacy configurations from td-agents and restart, the traffic will be transferred into new msk topics from old msk topics. The total traffic into kafka is the same as the origin, so we keep the logs exactly the same as before.