Kubernetes를 이용한 효율적인 데이터 엔지니어링

(Airflow on Kubernetes VS Airflow Kubernetes Executor)

이웅규 LINE FINANCIAL+

CONTENTS

01 데이터 엔지니어링이란?

02 Apache Airflow는 무엇인가?

03 Kubernetes X Airflow 왜 필요했을까?

04 일반적인 Airflow on Kubernetes

05 KubernetesExecutor & KubernetesPodOperator

06 저희는 이렇게 사용합니다.

1. 데이터 엔지니어링이란?

사람마다 생각에 따라 정의 및 범위가 다르다고 생각한다.

1. 데이터 엔지니어링이란?

LINE

증권
스마트 투자
외환 거래
가계부
신용 정보
대출
보험
암호 화폐
뱅크
⋮

kubernetes × Apache Airflow

Kerberos

Apache NiFi | Apache Zeppelin | HUE | Apache Spark
Apache Kafka | Apache Spark | Hive | cloudera IMPALA
Apache Spark | YARN
Sqoop | hadoop

tableau

Service Layer

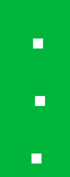
Kerberos

Hive
Apache Spark
hadoop

1. 데이터 엔지니어링이란?

LINE

증권
스마트 투자
외환 거래
가계부
신용 정보
대출
보험
암호 화폐
뱅크
⋮



kubernetes × Apache Airflow
Kerberos
nifi
Apache Zeppelin
HUE
Spark
kafka
Spark
HIVE
cloudera IMPALA
Spark
YARN
hadoop
tableau
Service Layer
Kerberos
HIVE
Spark
hadoop

1. 데이터 엔지니어링이란?

LINE

증권
스마트 투자
외환 거래
가계부
신용 정보
대출
보험
암호 화폐
뱅크
⋮

kubernetes × Apache Airflow

수집

Kerberos

nifi
kafka
Spark
sqoop

Apache Zeppelin
HUE
Spark
HIVE
cloudera IMPALA
Spark
YARN
hadoop

tableau
Service Layer
Kerberos
HIVE
Spark
hadoop

1. 데이터 엔지니어링이란?

LINE

증권

스마트 투자

외환 거래

가계부

신용 정보

대출

보험

암호 화폐

뱅크

kubernetes × Apache Airflow

Kerberos

nifi

Apache Zeppelin

HUE

kafka

Spark

HIVE

cloudera IMPALA

저장

YARN

hadoop

tableau

Service Layer

1. 데이터 엔지니어링이란?

LINE

증권

스마트 투자

외환 거래

가계부

신용 정보

대출

보험

암호 화폐

뱅크

kubernetes

Apache Airflow

Kerberos

가공

Apache Zeppelin

kafka

Spark

HIVE

cloudera IMPALA

YARN

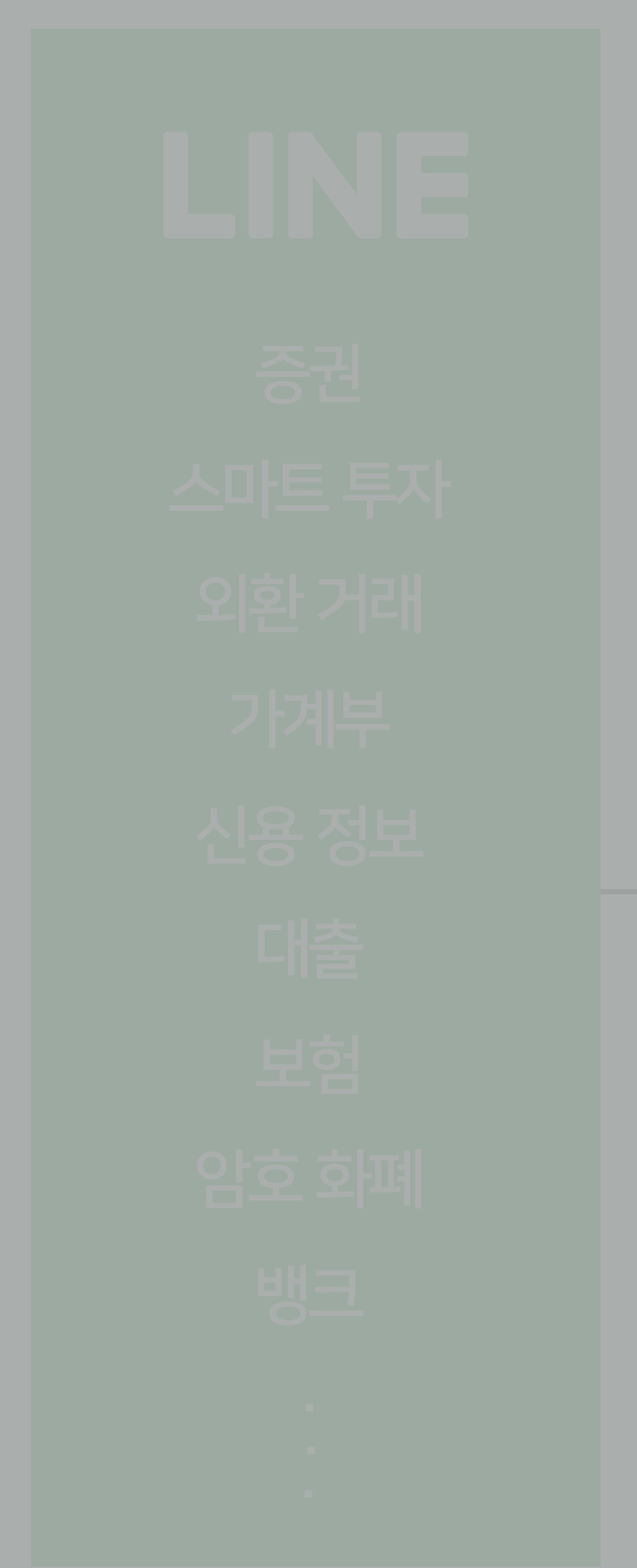
hadoop

tableau

Service Layer

Kerberos

1. 데이터 엔지니어링이란?



분석 및 가치창출

Kerberos

Apache Zeppelin

HUE

Spark

cloudera IMPALA

kubernetes

Apache Airflow

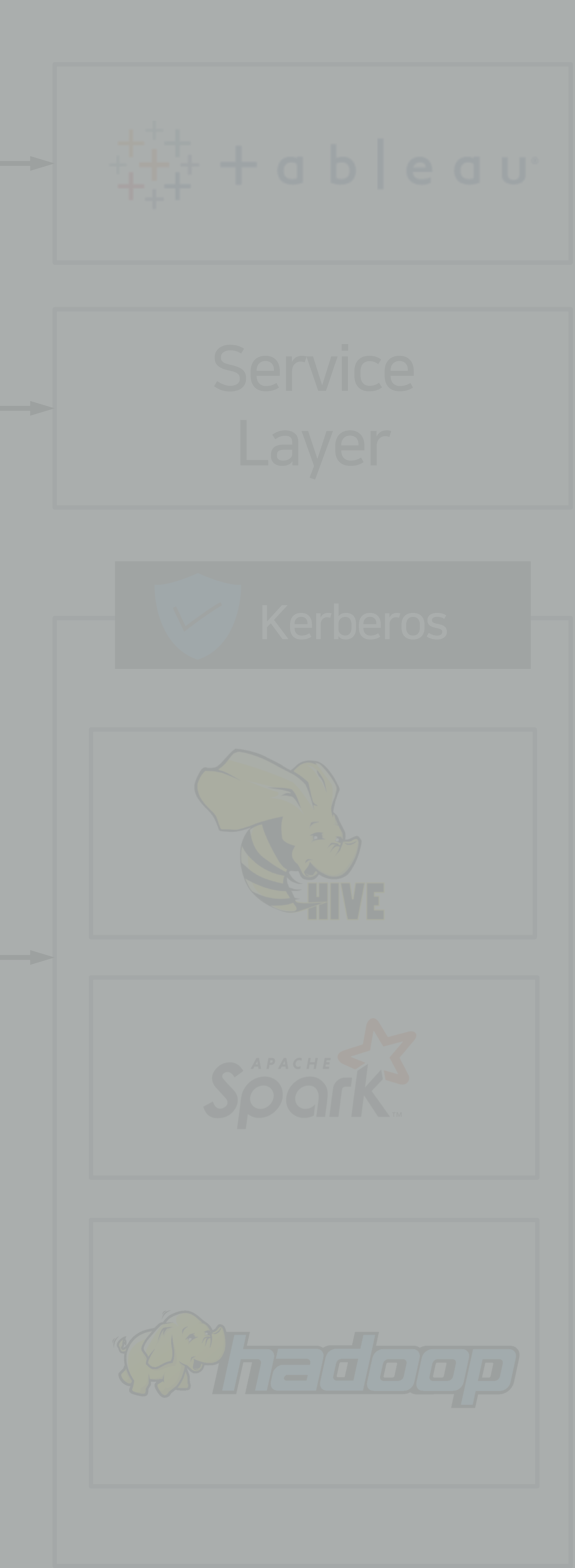
nifi

kafka

HIVE

YARN

hadoop



1. 데이터 엔지니어링이란?

LINE
증권
스마트 투자
외환 거래
가계부
신용 정보
대출
보험
암호 화폐
뱅크
⋮

kubernetes × Apache Airflow
Kerberos
nifi | Apache Zeppelin | HUE | Spark
kafka | Spark | HIVE | cloudera IMPALA
Spark | YARN
hadoop

서비스

tableau
Service Layer
Kerberos
HIVE
Spark
hadoop

1. 데이터 엔지니어링이란?

kubernetes × Apache Airflow

LINE
증권
스마트 투자
외환 거래
가계부
신용 정보
대출
보험
암호 화폐
뱅크

Kerberos
nifi
Apache Zeppelin
HUE
kafka
Spark
HIVE
cloudera IMPALA
YARN
hadoop
tableau
Service Layer

2. Apache Airflow는 무엇인가?

Airflow is a platform created by the community to programmatically author, schedule and monitor workflows.

2. Apache Airflow는 무엇인가?

Dag & Task

A_Service_MySQL_2_HDFS
B_Service_MSSQL_2_HDFS
C_Service_HDFS_2_HDFS
D_Service_HIVE_2_HDFS
E_Service_Log_2_HDFS
F_Service_FTP_2_HDFS
G_Service_S3_2_HDFS
Make_Data_Lake
Make_Data_Warehouse
Make_Data_Mart
Make_KPI_Report
Transform_Data
Deliver_Data
Monitoring_Data
Make_Feature
Prediction
Deliver_Output_Data
Update_Model
Deploy_Model
Transfer_Other_Platform

2. Apache Airflow는 무엇인가?

Dag (개발자가 코드로 작성한 Workflow)

A_Service_MySQL_2_HDFS
B_Service_MSSQL_2_HDFS
C_Service_HDFS_2_HDFS
D_Service_HIVE_2_HDFS
E_Service_Log_2_HDFS
F_Service_FTP_2_HDFS
G_Service_S3_2_HDFS
Make_Data_Lake
Make_Data_Warehouse
Make_Data_Mart
Make_KPI_Report
Transform_Data
Deliver_Data
Monitoring_Data
Make_Feature
Prediction
Deliver_Output_Data
Update_Model
Deploy_Model
Transfer_Other_Platform

2. Apache Airflow는 무엇인가?

Task (작업의 단위)

A_Service_MySQL_2_HDFS
B_Service_MSSQL_2_HDFS
C_Service_HDFS_2_HDFS
D_Service_HIVE_2_HDFS
E_Service_Log_2_HDFS
F_Service_FTP_2_HDFS
G_Service_S3_2_HDFS
Make_Data_Lake
Make_Data_Warehouse
Make_Data_Mart
Make_KPI_Report
Transform_Data
Deliver_Data
Monitoring_Data
Make_Feature
Prediction
Deliver_Output_Data
Update_Model
Deploy_Model
Transfer_Other_Platform

2. Apache Airflow는 무엇인가?

2.1 Apache Airflow의 동작원리

Ticket renewal

Kerberos

Dag Script

Queue

Enqueue Task

Dequeue Task

Find Executable Task

Read DAG

Create Task

Set Task Status

Set Task Status

Execute

Run

Scheduler

Meta DB

Executor

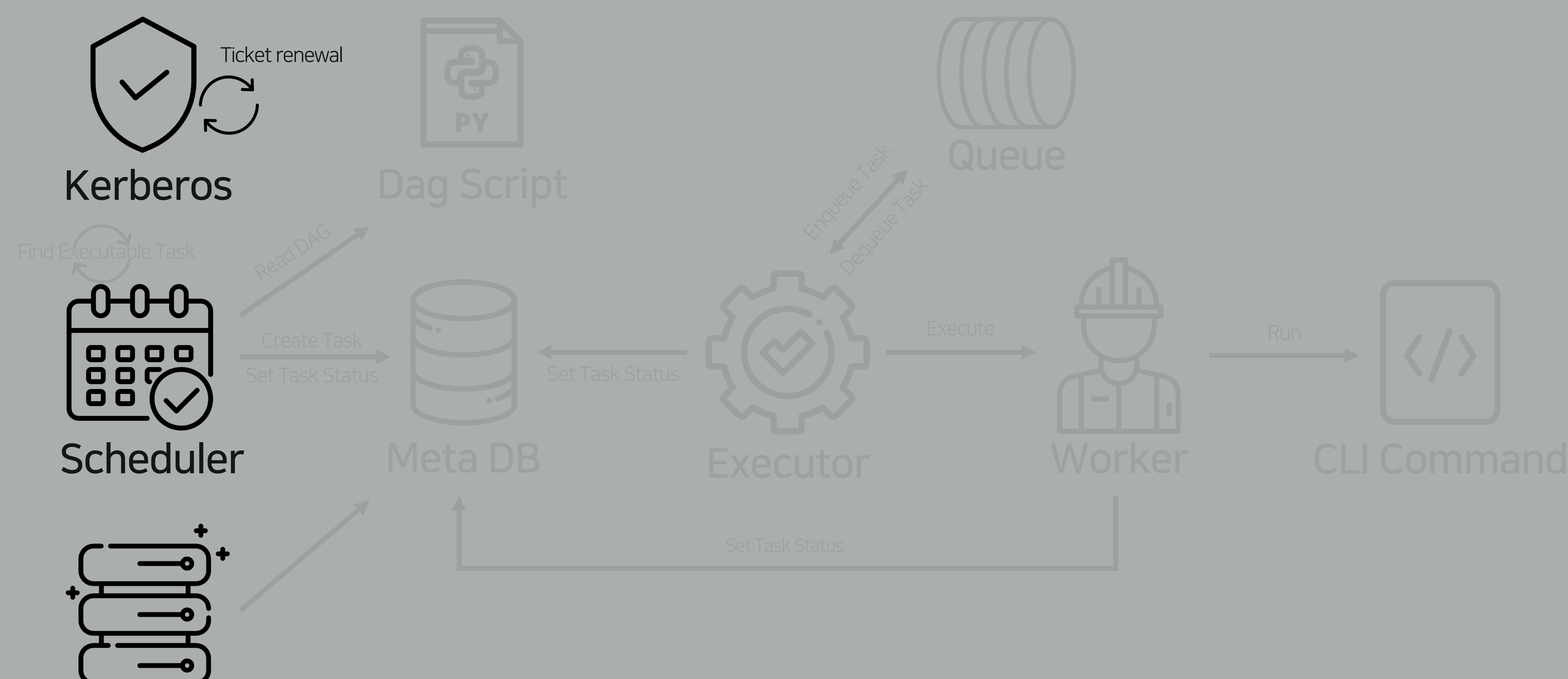
Worker

CLI Command

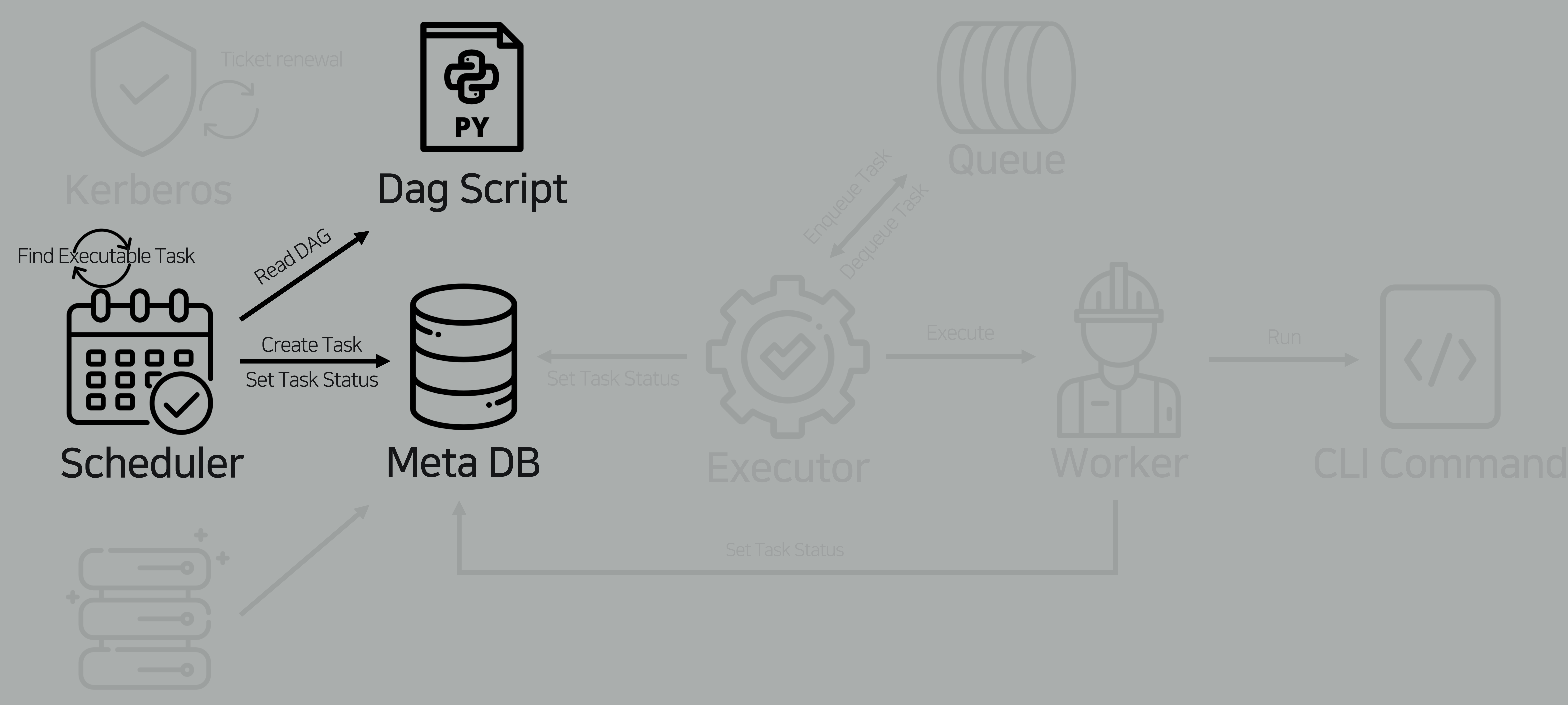
Set Task Status

Web Sever

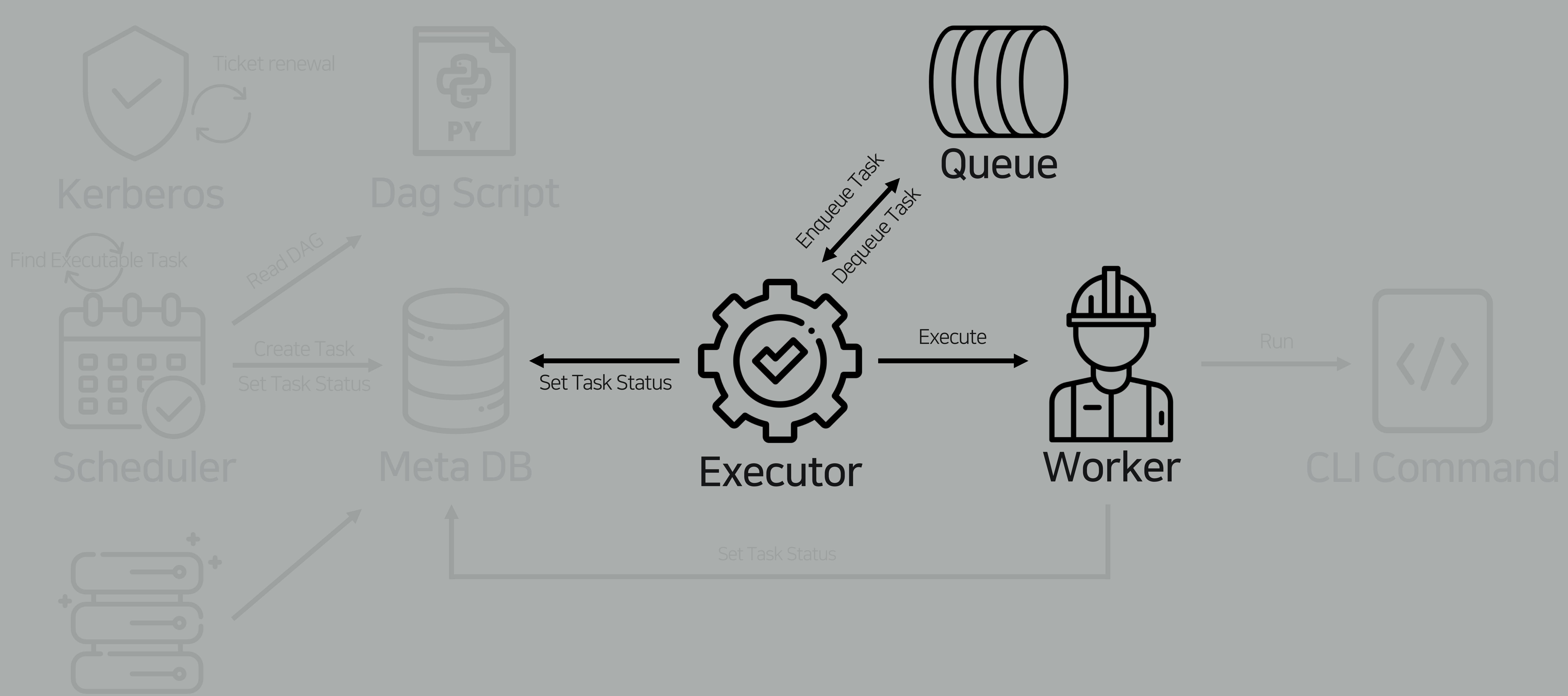
2.1 Apache Airflow의 동작원리



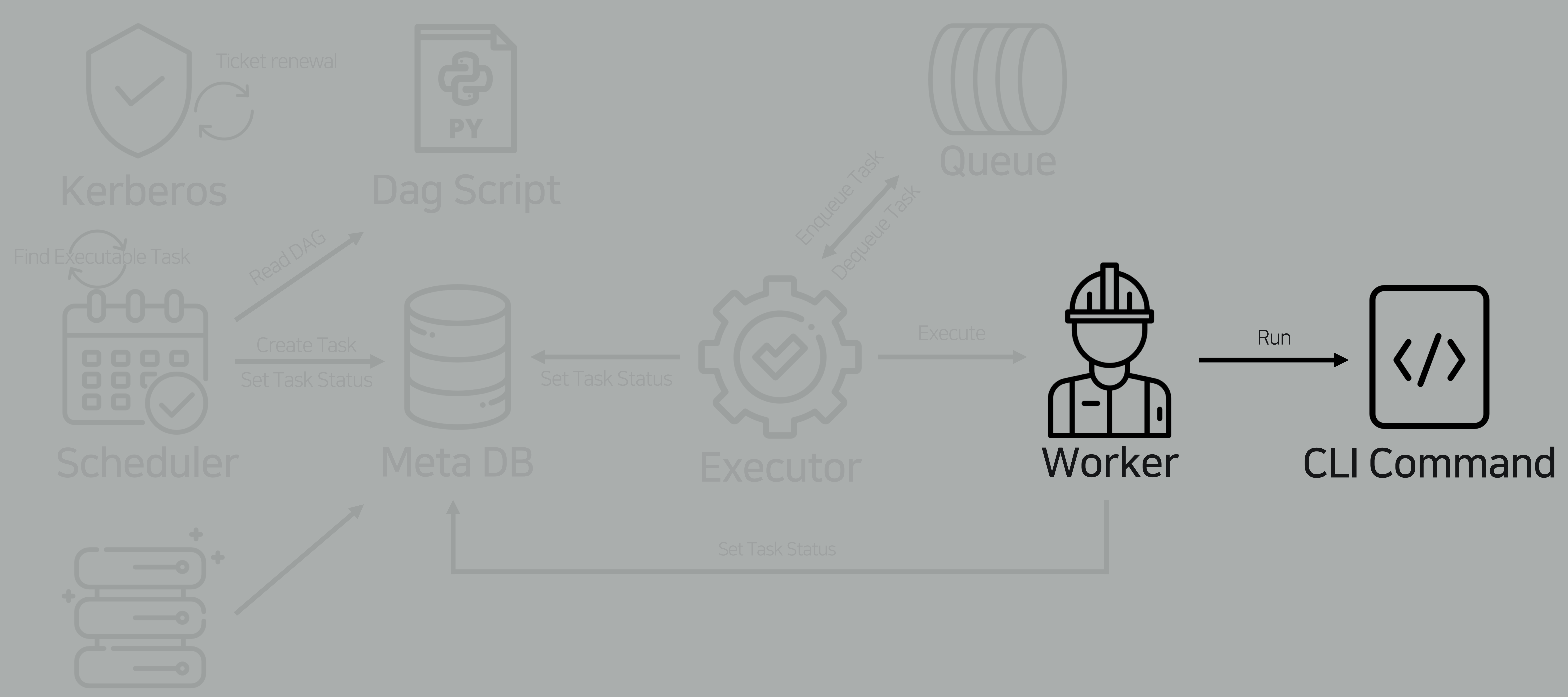
2.1 Apache Airflow의 동작원리



2.1 Apache Airflow의 동작원리



2.1 Apache Airflow의 동작원리



2.1 Apache Airflow의 동작원리

```
SELECT
   column 1
  , transform(column1)
FROM database.table
```

Generate Hive Cli

```
$ beeline
-u …
-n … -p …
-f transform.hql
-hiveconf …
```

subprocess

Scheduler process
(Parent of Executor Process)

Execute

Worker process
(Parent of Hive Client Process
Child of Scheduler Process)

Execute

Hive Client process
(Child of worker Process)

Hive Client
(Java Process)

2.1 Apache Airflow의 동작원리

HiveOperator → Generate Hive Cli → subprocess → Hive Client

```
SELECT
    column 1
   , transform(column1)
FROM database.table
```

```
$ beeline
-u …
-n … -p …
-f transform.hql
-hiveconf …
```

Scheduler process
(Parent of Executor Process)
↓ Execute
Worker process
(Parent of Hive Client Process
Child of Scheduler Process)
↓ Execute
Hive Client process
(Child of worker Process)

Hive Client
(Java Process)

2.1 Apache Airflow의 동작원리

HiveOperator → **Generate Hive Cli** → **subprocess** →

Hive Client
(Java Process)

```
SELECT
    column 1
   , transform(column1)
FROM database.table
```

```
$ beeline
-u …
-n … -p …
-f transform.hql
-hiveconf …
```

Scheduler process
(Parent of Worker Process)

Execute

Worker process
(Parent of Hive Client Process
Child of Scheduler Process)

Execute

Hive Client process
(Child of worker Process)

2.1 Apache Airflow의 동작원리

Airflow에서 HiveOperator 사용 시 Hive 설치 및 환경이 구성돼야 함

HiveOperator → Generate Hive Cli → subprocess → Hive Client

HiveOperator

```
SELECT
    column 1
   , transform(column1)
FROM database.table
```

Generate Hive Cli

```
$ beeline
-u …
-n … -p …
-f transform.hql
-hiveconf …
```

subprocess

Scheduler process
(Parent of Worker Process)

↓ Execute

Worker process
(Parent of Hive Client Process
Child of Scheduler Process)

↓ Execute

Hive Client process
(Child of worker Process)

HIVE

Hive Client
(Java Process)

2.1 Apache Airflow의 동작원리

```
SELECT
    column 1
   , transform(column1)
FROM database.table
```

Generate Hive Cli

```
$ beeline
-u …
-n … -p …
-f transform.hql
-hiveconf …
```

subprocess

Scheduler process
(Parent of Worker Process)

Execute

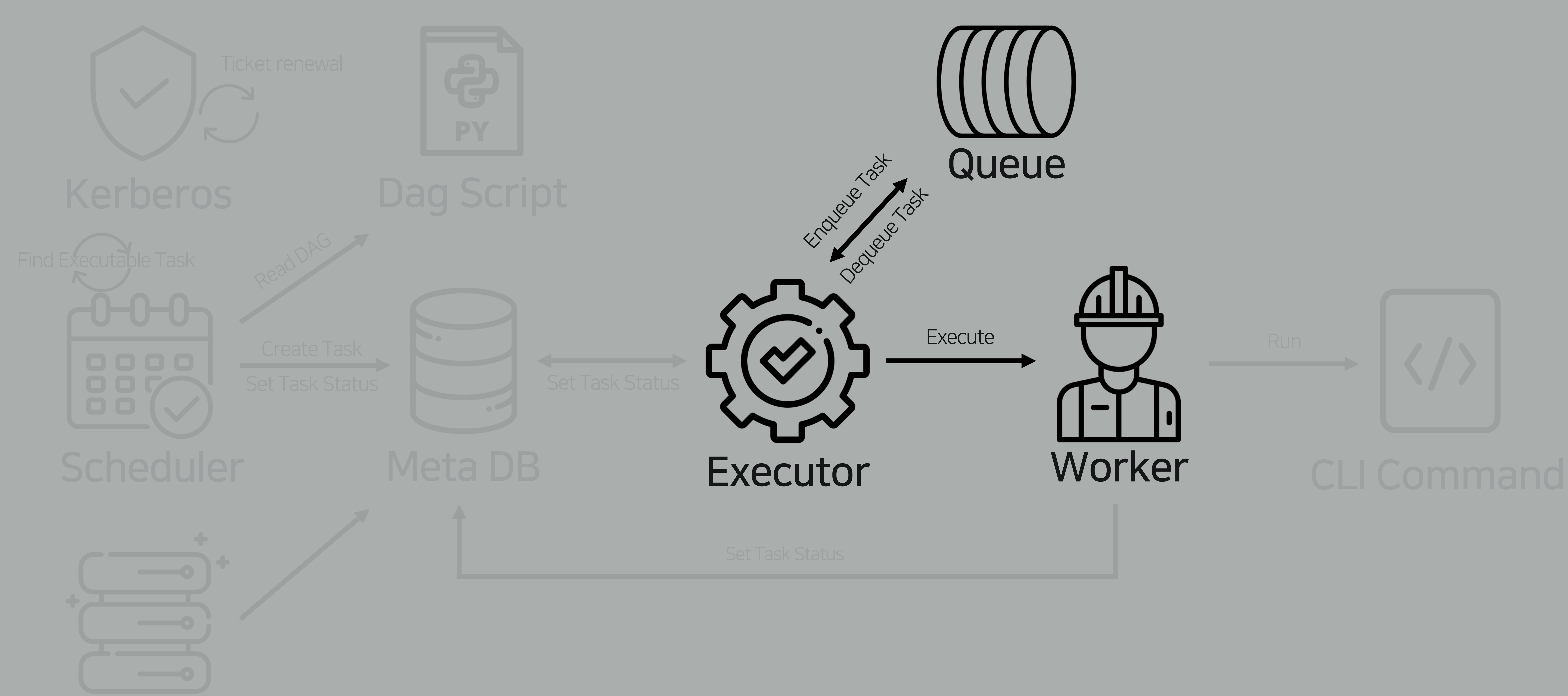
Worker process
(Parent of Hive Client Process
Child of Scheduler Process)

Execute

Hive Client process
(Child of worker Process)

Hive Client
(Java Process)

2.1 Apache Airflow의 동작원리



2.2 Apache Airflow Executor별 특징

Base
Executor

Local
Executor

Celery
Executor

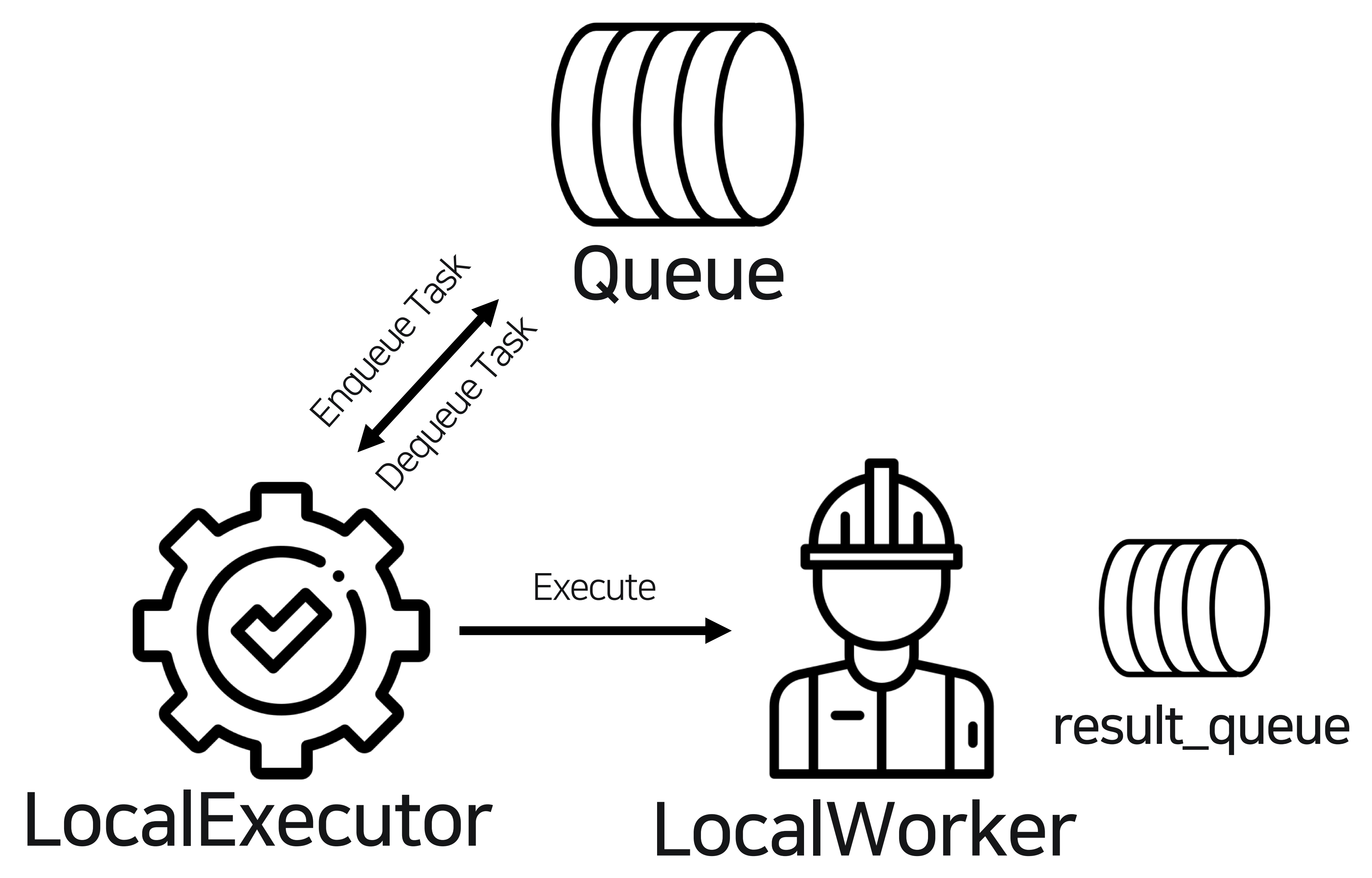
Dask
Executor

Debug
Executor

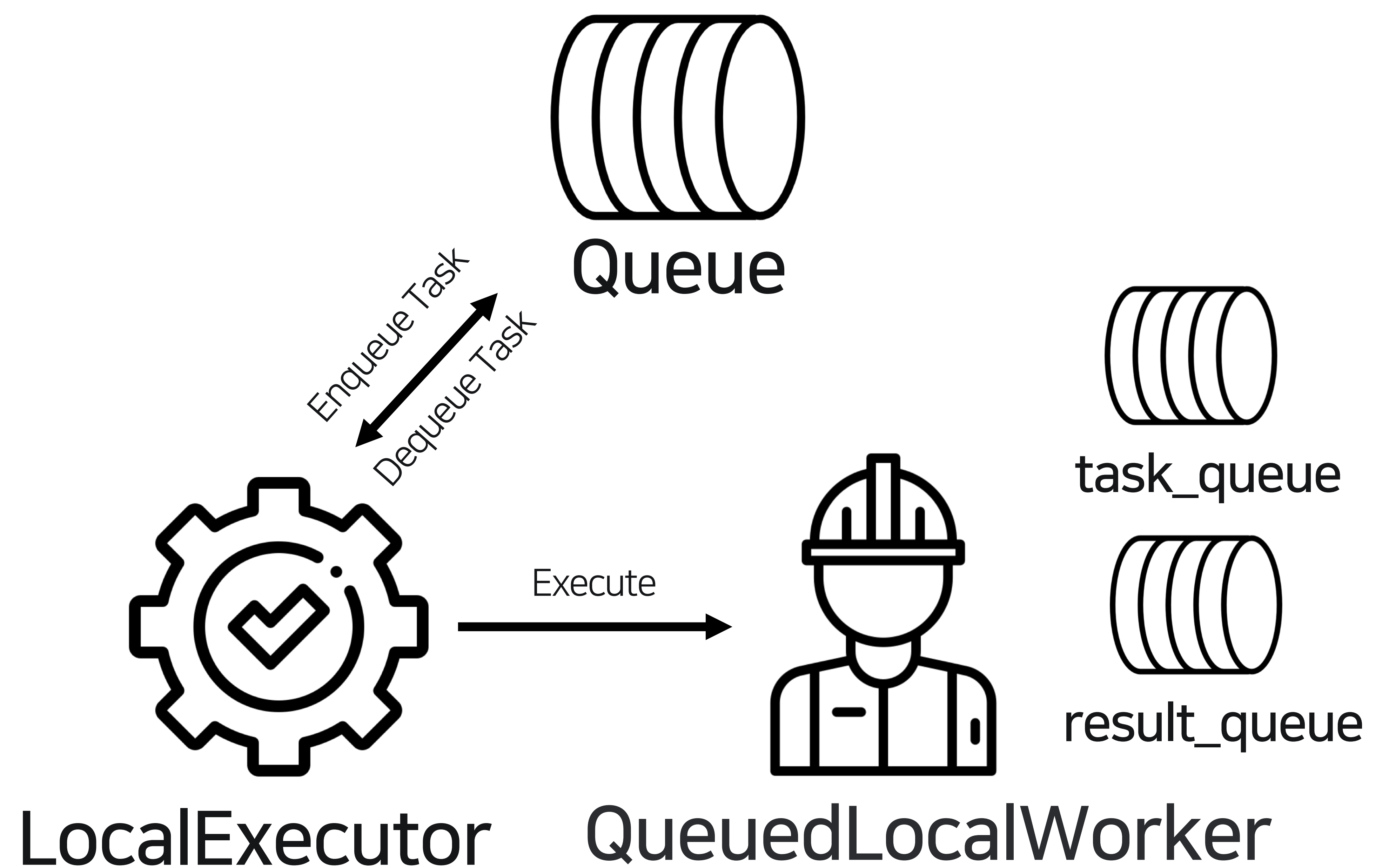
Sequential
Executor

Kubernetes
Executor

2.2 LocalExecutor



_UnlimitedParallelism



_LimitedParallelism

2.2 LocalExecutor

local_executor.py

```
self.impl = (LocalExecutor._UnlimitedParallelism(self) if self.parallelism == 0
             else LocalExecutor._LimitedParallelism(self))

self.impl.start()
```

```
local_worker = LocalWorker(self.executor.result_queue)
local_worker.key = key
local_worker.command = command
self.executor.workers_used += 1
self.executor.workers_active += 1
local_worker.start()
```

_UnlimitedParallelism

```
self.queue = self.executor.manager.Queue()
self.executor.workers = [
    QueuedLocalWorker(self.queue, self.executor.result_queue)
    for _ in range(self.executor.parallelism)
]

self.executor.workers_used = len(self.executor.workers)

for w in self.executor.workers:
    w.start()
```

_LimitedParallelism

2.2 CeleryExecutor

CeleryExecutor

Airflow Machine

RabbitMQ

OR

redis

...

Broker Cluster

Celery Worker

Worker Machine 1

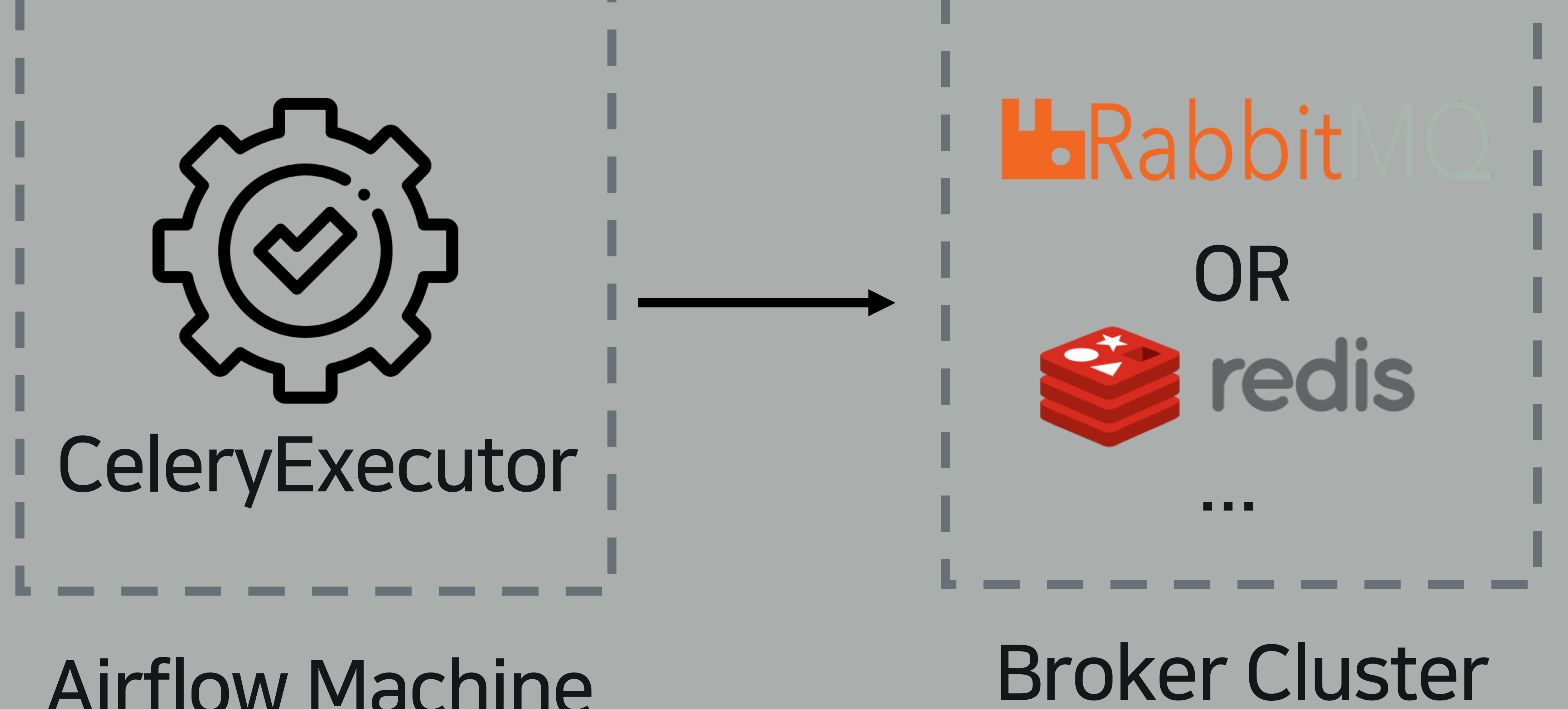
CLI Command

Celery Worker

Worker Machine 2

CLI Command

2.2 CeleryExecutor




```python
def trigger_tasks(self, open_slots):
    """
    Overwrite trigger_tasks function from BaseExecutor

    :param open_slots: Number of open slots
    :return:
    """
    sorted_queue = sorted(
        [(k, v) for k, v in self.queued_tasks.items()],
        key=lambda x: x[1][1],
        reverse=True)

    task_tuples_to_send = []

    for i in range(min((open_slots, len(self.queued_tasks)))):
        key, (command, _, queue, simple_ti) = sorted_queue.pop(0)
        task_tuples_to_send.append((key, simple_ti, command, queue,
                                    execute_command))

    cached_celery_backend = None
    if task_tuples_to_send:
        tasks = [t[4] for t in task_tuples_to_send]

        # Celery state queries will stuck if we do not use one same backend
        # for all tasks.
        cached_celery_backend = tasks[0].backend

    if task_tuples_to_send:
        # Use chunking instead of a work queue to reduce context switching
        # since tasks are roughly uniform in size
        chunksize = self._num_tasks_per_send_process(len(task_tuples_to_send))
        num_processes = min(len(task_tuples_to_send), self._sync_parallelism)

        send_pool = Pool(processes=num_processes)
        key_and_async_results = send_pool.map(
            send_task_to_executor,
            task_tuples_to_send,
            chunksize=chunksize)
```

수행해야 할 Task

비동기로 Task를 전송

celery_executor.py

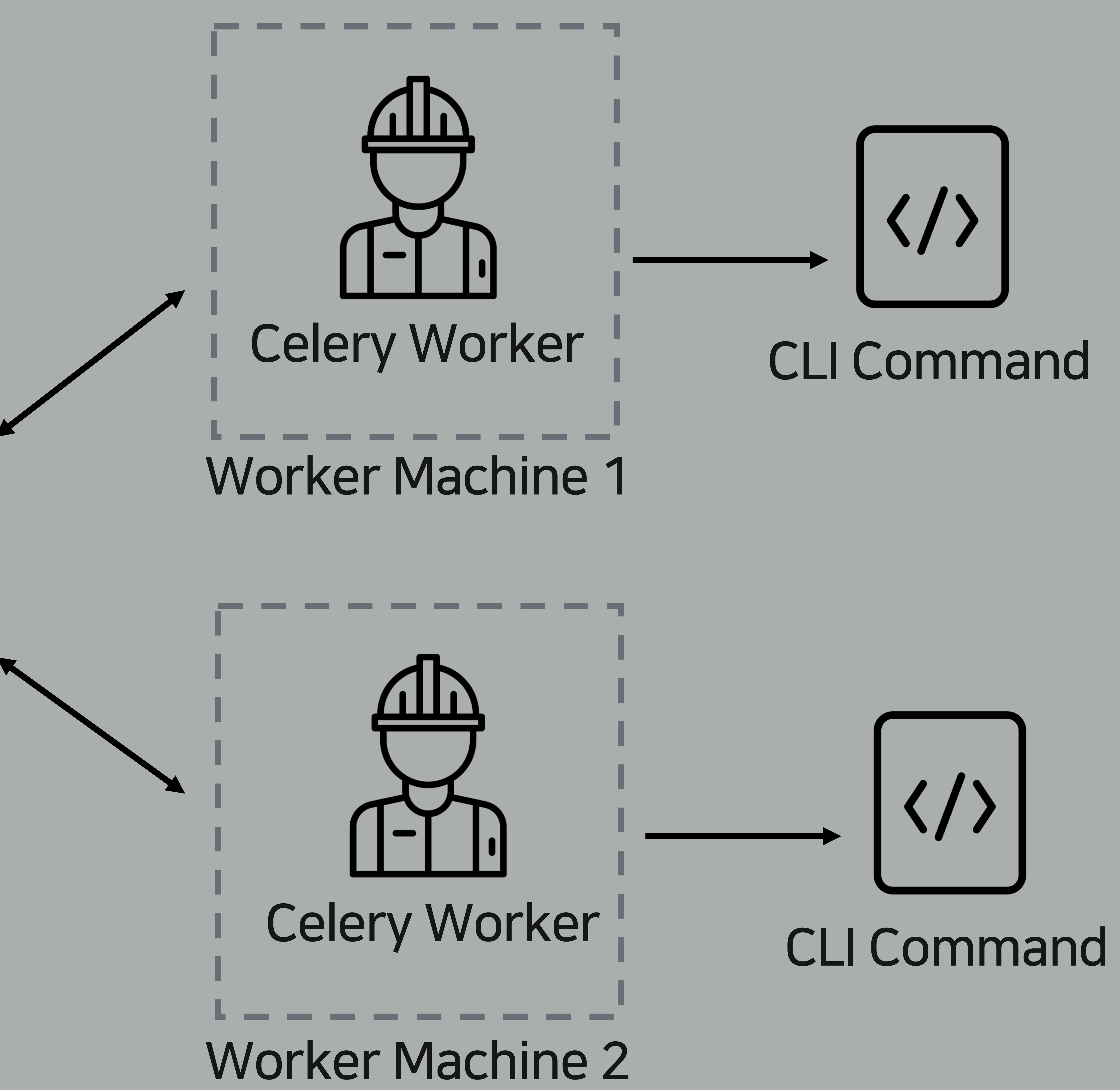
2.2 CeleryExecutor

cli.py

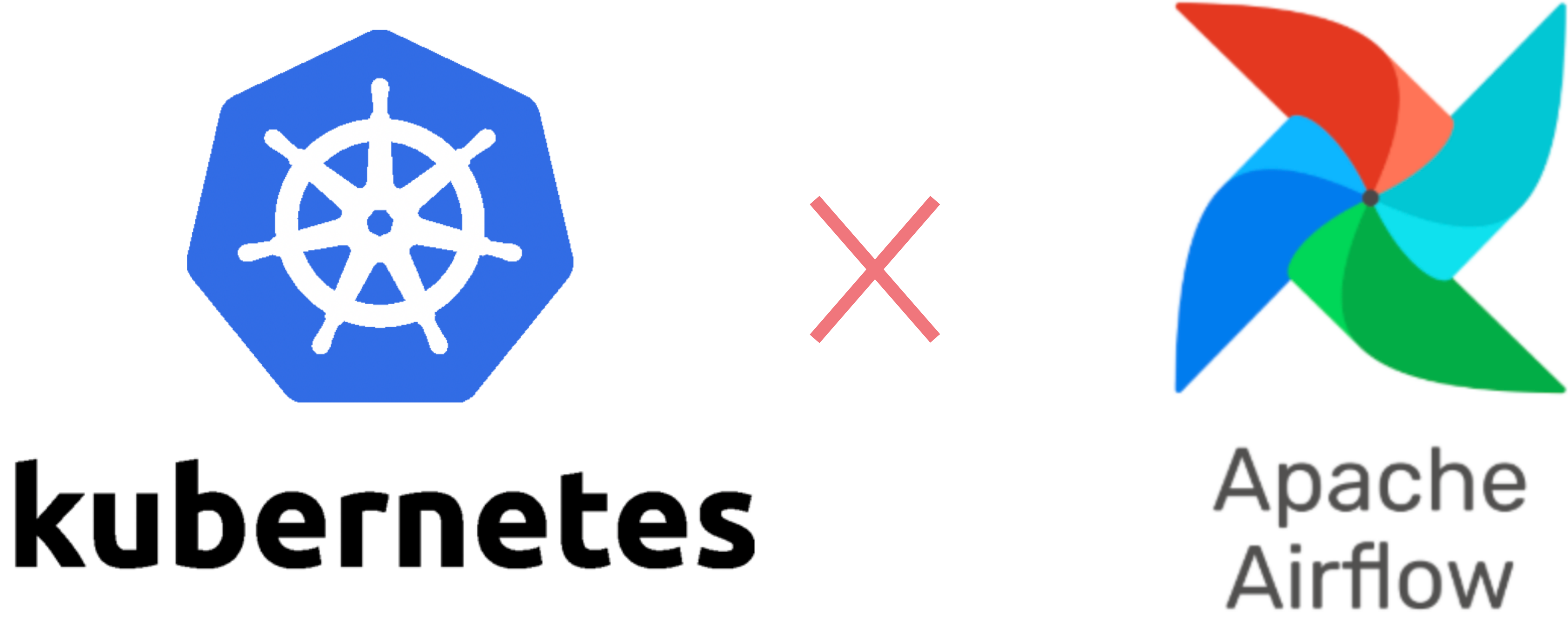
```python
@cli_utils.action_logging
def worker(args):
    env = os.environ.copy()
    env['AIRFLOW_HOME'] = settings.AIRFLOW_HOME

    if not settings.validate_session():...

    # Celery worker
    from airflow.executors.celery_executor import app as celery_app
    from celery import maybe_patch_concurrency
    from celery.bin import worker

    worker = worker.worker(app=celery_app)
    options = {...}

    if conf.has_option("celery", "pool"):...

    if args.daemon:
        pid, stdout, stderr, log_file = setup_locations("worker",
                                                        args.pid,
                                                        args.stdout,
                                                        args.stderr,
                                                        args.log_file)
        handle = setup_logging(log_file)
        stdout = open(stdout, 'w+')
        stderr = open(stderr, 'w+')

        ctx = daemon.DaemonContext(
            pidfile=TimeoutPIDLockFile(pid, -1),
            files_preserve=[handle],
            stdout=stdout,
            stderr=stderr,
        )
        with ctx:
            sp = _serve_logs(env, skip_serve_logs)
            worker.run(**options)

        stdout.close()
        stderr.close()
```

비동기 작업 Queue
Celery Worker

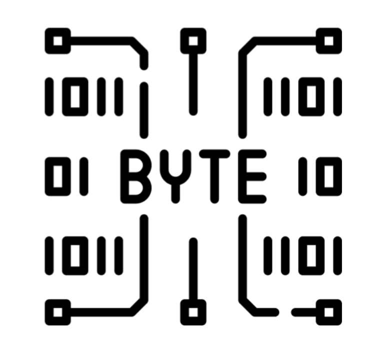
Daemon Process

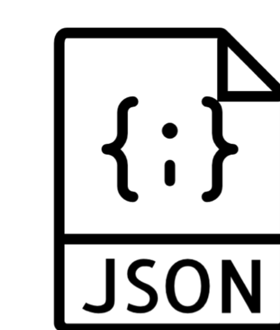


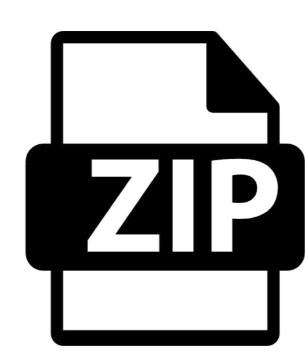
3. 왜 필요했을까?

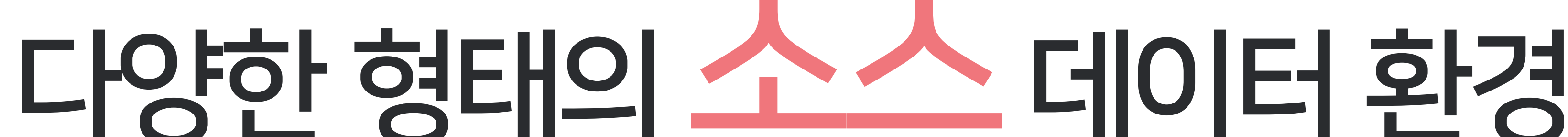


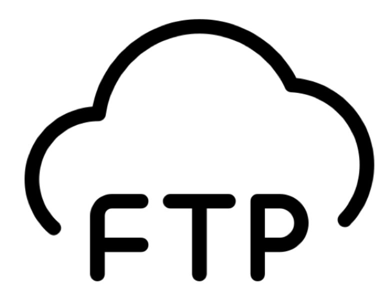
3. 왜 필요했을까?

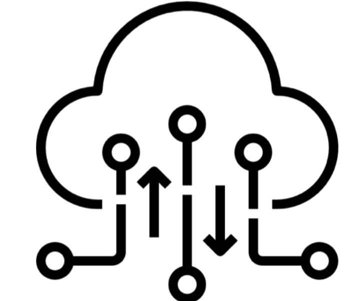












Airflow On Kubernetes VS Kubernetes Executor

4. 일반적인 Airflow on Kubernetes

DEVIEW 2020

Kubernetes Cluster

Airflow Webserver
Airflow Kerberos

Airflow Scheduler
Airflow Kerberos

LocalExecutor Env.

4. 일반적인 Airflow on Kubernetes

Kubernetes Cluster

N

pod

pod

pod

pod

pod

Airflow Webserver
Airflow Scheduler
Airflow Kerberos

Redis or RabbitMQ
Cluster

Airflow Worker
Airflow Kerberos

CeleryExecutor Env.

4. 일반적인 Airflow on Kubernetes

장점	단점
구성이 간단 Template화 하기 쉬움	유지보수 비용 Lib. 의존성에 따라 무거워 짐 Kubernetes의 장점을 살리지 못함

4. 일반적인 Airflow on Kubernetes 장점

Kubernetes Cluster

Central Platform

Deploy

Airflow Webserver
Airflow Scheduler
Airflow Kerberos

pod

Airflow Webserver
Airflow Scheduler
Airflow Kerberos

pod

Airflow Webserver
Airflow Scheduler
Airflow Kerberos

pod

Data Science Dept.

B.I Dept.

Data Engineering Dept.

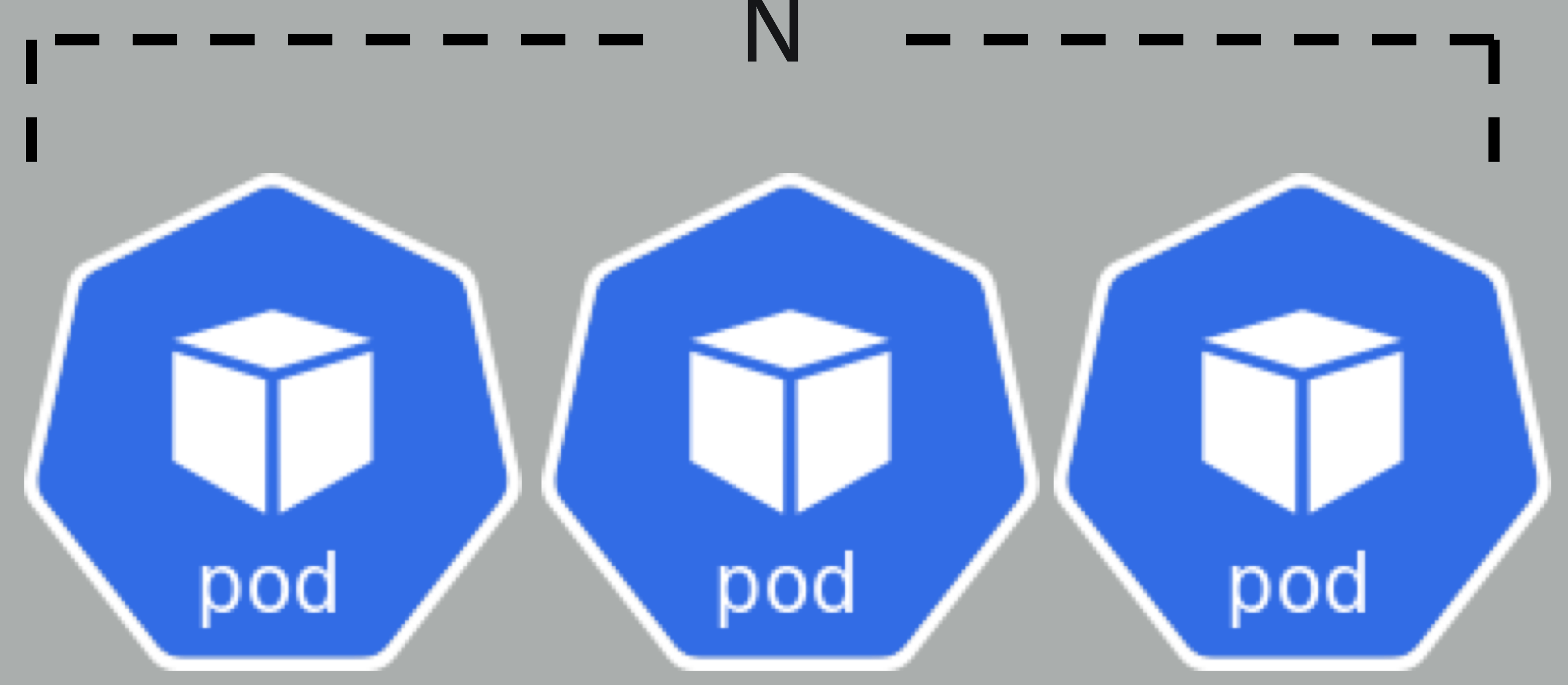
4. 일반적인 Airflow on Kubernetes 단점

Kubernetes Cluster

해당 구성은 Kubernetes 환경에 상주 하고 있으며, 모니터링 필요

Airflow Webserver
Airflow Scheduler
Airflow Kerberos

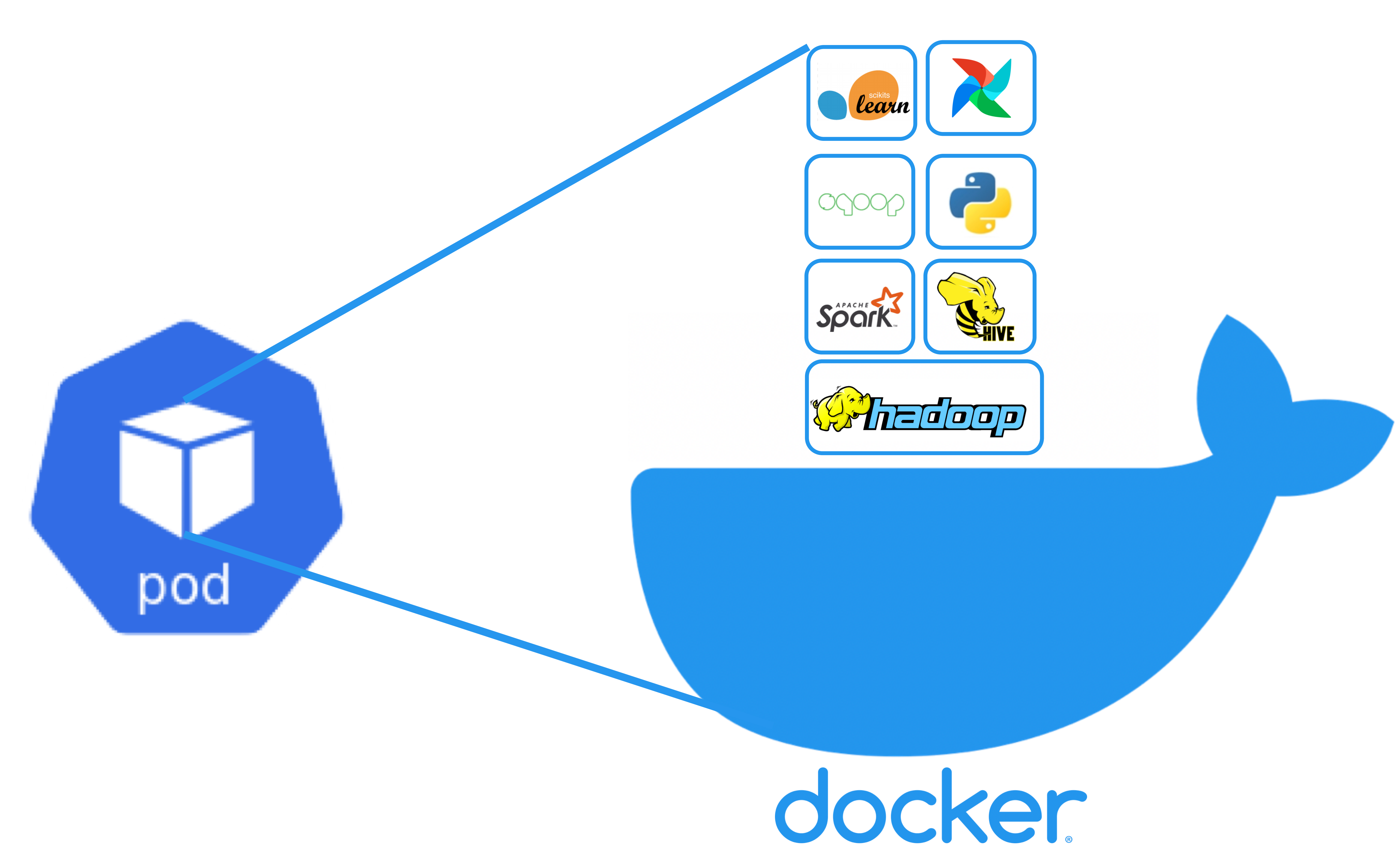
Redis or RabbitMQ
Cluster



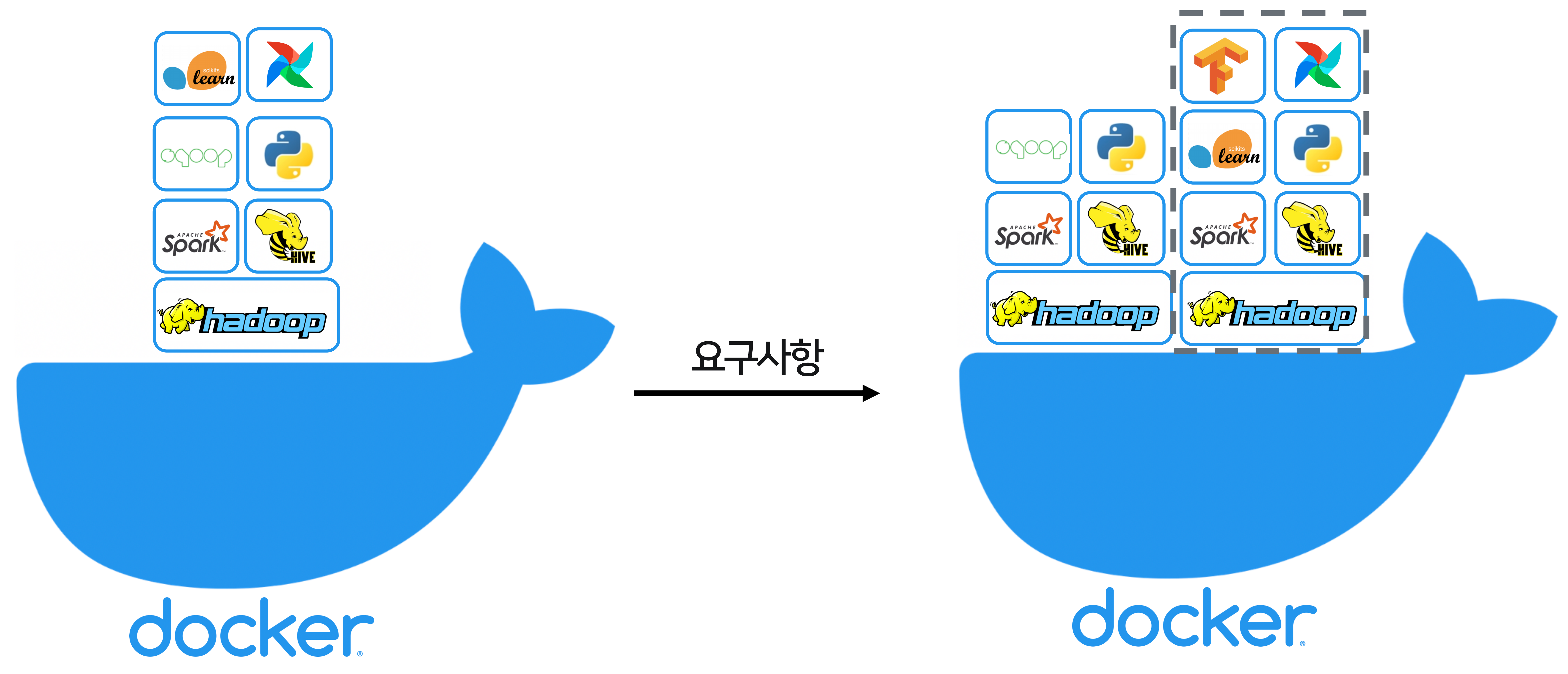
Airflow Worker
Airflow Kerberos

CeleryExecutor Env

4. 일반적인 Airflow on Kubernetes 단점



4. 일반적인 Airflow on Kubernetes 단점



4. 일반적인 Airflow on Kubernetes 단점

작업 List...

신규 Hadoop Client 설치

환경변수 확인

사용 Component기능 확인

Library 의존성 여부

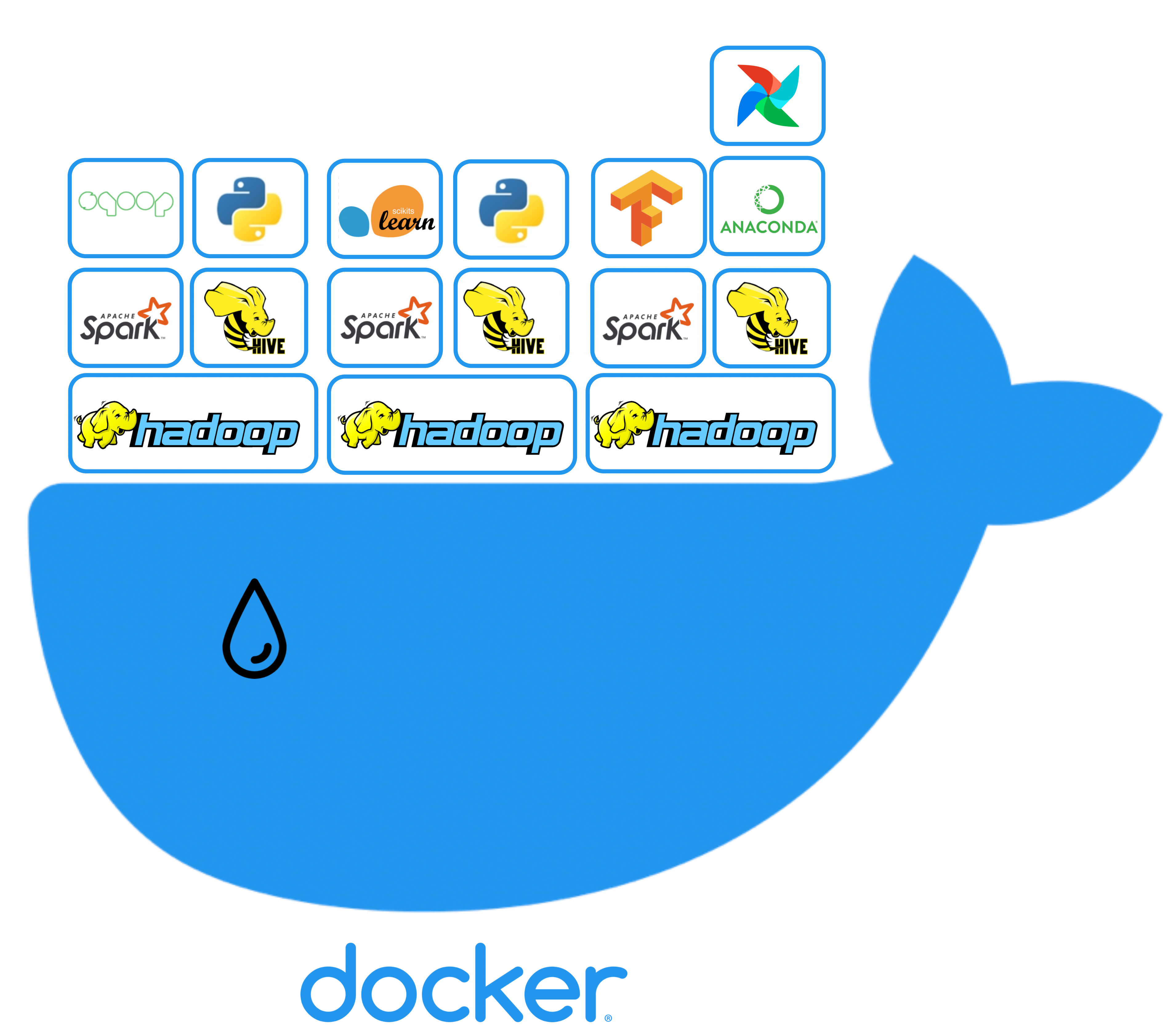
Python Version 확인

기존 코드 정상 작동 여부 확인

...

추가로 또 다른 Hadoop Client도 설치 돼야 한다면…

4. 일반적인 Airflow on Kubernetes 단점

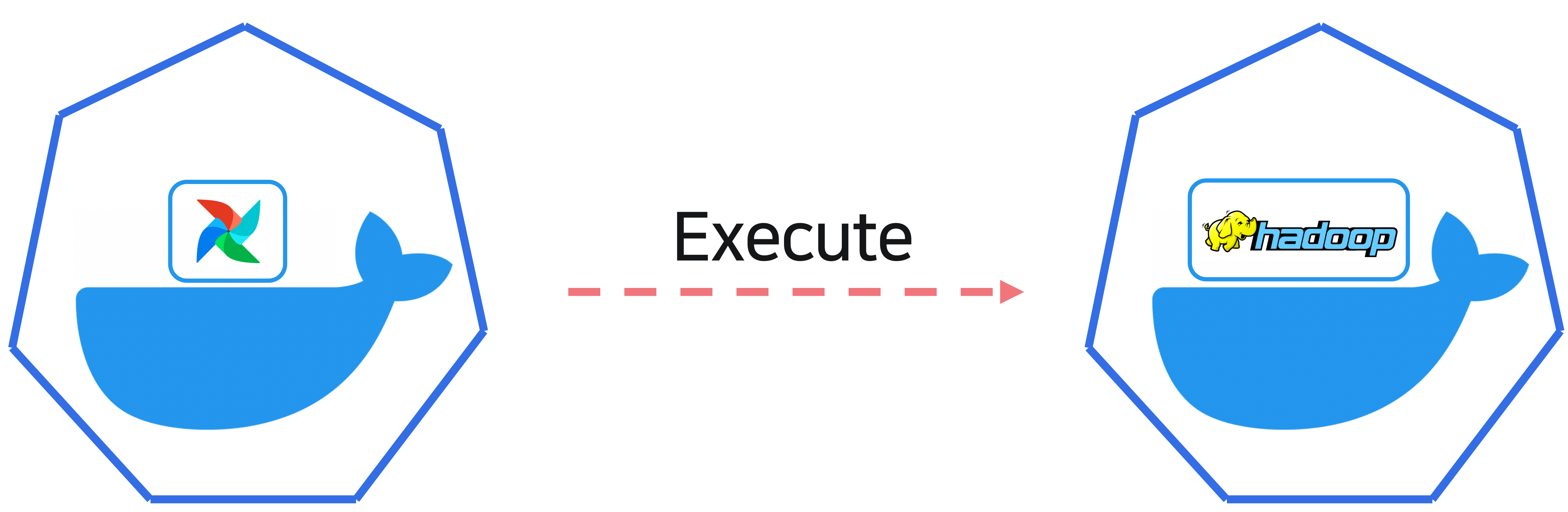


5. K8sExecutor & K8sPodOperator

필요할 때만 Kubernetes 자원을 사용하고

필요한 Docker Container만을 골라 Pod로 실행시킨다

Like Micro Service Architecture...



5. K8sExecutor & K8sPodOperator

Hive Cluster A Pod
Hadoop 2.x Hive 1.x

Hadoop Cluster A Pod
Hadoop 2.x Hive 1.x

ETL_Data_Hadoop_Cluster_A → Make_Data_Warehouse_Clustr_A → Make_Feature_Clustr_A → Transfer_To_Clustr_B → Transform_Data_From_Cluser_B

Spark Cluster A Pod
Hadoop 2.x Spark 2.x

Python Pod
Python 3.x Pandas 1.x

Spark Cluster B Pod
Hadoop 3.x Spark 2.x

5. K8sExecutor & K8sPodOperator

Hive Cluster A Pod
Hadoop 2.x Hive 1.x

Hadoop Cluster A Pod
Hadoop 2.x Hive 1.x

ETL_Data_Hadoop_Cluster_A → Make_Data_Warehouse_Clustr_A → Make_Feature_Clustr_A → Transfer_To_Clustr_B → Transform_Data_From_Cluser_B

Spark Cluster A Pod
Hadoop 2.x Spark 2.x

Python Pod
Python 3.x Pandas 1.x

Spark Cluster B Pod
Hadoop 3.x Spark 2.x

5. K8sExecutor & K8sPodOperator

일반 Operator & KubernetesPodOperator

KubernetesExecutor Env.

5. K8sExecutor & K8sPodOperator

Kubernetes Cluster

pod
Airflow Webserver

pod
Airflow Scheduler

K8sExecutor

pod
Airflow Worker

BashOperator

pod
Airflow Worker

PythonOperator

5. K8sExecutor & K8sPodOperator

Kubernetes Cluster

pod

Airflow Webserver

pod

Airflow Scheduler

K8sExecutor

pod

Airflow Worker

BashOperator

pod

Airflow Worker

PythonOperator

일반 Operator

5. K8sExecutor & K8sPodOperator

Kubernetes Cluster

kubernetes_executor.py

```
for _ in range(self.kube_config.worker_pods_creation_batch_size):
    try:
        task = self.task_queue.get_nowait()
        try:
            self.kube_scheduler.run_next(task)
        except ApiException as e:
            self.log.warning('ApiException when attempting to run task, re-queueing. '
                             'Message: %s', json.loads(e.body)['message'])
            self.task_queue.put(task)
        except HTTPError as e:
            self.log.warning('HTTPError when attempting to run task, re-queueing. '
                             'Exception: %s', str(e))
            self.task_queue.put(task)
        finally:
            self.task_queue.task_done()
    except Empty:
        break
```

Worker Pod 생성

```
if command[0:2] != ["airflow", "run"]:
    raise ValueError('The command must start with ["airflow", "run"].')

pod = PodGenerator.construct_pod(
    namespace=self.namespace,
    worker_uuid=self.worker_uuid,
    pod_id=self._create_pod_id(dag_id, task_id),
    dag_id=pod_generator.make_safe_label_value(dag_id),
    task_id=pod_generator.make_safe_label_value(task_id),
    try_number=try_number,
    date=self._datetime_to_label_safe_datestring(execution_date),
    command=command,
    kube_executor_config=kube_executor_config,
    worker_config=self.worker_configuration_pod
)
```

airflow.cfg에 정의한 config & image

pod

Executor

pod

Airflow Scheduler

일반 Operator

5. K8sExecutor & K8sPodOperator

Kubernetes Cluster

pod
Airflow Webserver

pod
Airflow Scheduler

K8sExecutor

pod
Airflow Worker

pod
Airflow Worker

Spark
hadoop
Hadoop 3.x Spark 2.x

Spark
hadoop
Hadoop 2.x Spark 2.x

5. K8sExecutor & K8sPodOperator

Kubernetes Cluster

pod

Airflow Webserver

pod

Airflow Scheduler

K8sExecutor

pod

Airflow Worker

pod

Airflow Worker

Spark
hadoop

Hadoop 3.x Spark 2.x

Spark
hadoop

Hadoop 2.x Spark 2.x

KubernetesPodOperator

5. K8sExecutor & K8sPodOperator

kubernetes_pod_operator.py

```python
def execute(self, context):
    try:
        if self.in_cluster is not None:...
        else:...

        # Add combination of labels to uniquely identify a running pod
        labels = self.create_labels_for_pod(context)

        label_selector = self._get_pod_identifying_label_string(labels)

        pod_list = client.list_namespaced_pod(self.namespace, label_selector=label_selector)

        if len(pod_list.items) > 1 and self.reattach_on_restart:...

        launcher = pod_launcher.PodLauncher(kube_client=client, extract_xcom=self.do_xcom_push)

        if len(pod_list.items) == 1:...
        else:
            final_state, _, result = self.create_new_pod_for_operator(labels, launcher)
        if final_state != State.SUCCESS:...
        return result
    except AirflowException as ex:
        raise AirflowException('Pod Launching failed: {error}'.format(error=ex))
```

새로운 User Define Pod생성

Dag script에 정의한 Config & Image

```python
pod = pod_generator.PodGenerator(
    image=self.image,
    namespace=self.namespace,
    cmds=self.cmds,
    args=self.arguments,
    labels=self.labels,
    name=self.name,
    envs=self.env_vars,
    extract_xcom=self.do_xcom_push,
    image_pull_policy=self.image_pull_policy,
    node_selectors=self.node_selectors,
    annotations=self.annotations,
    affinity=self.affinity,
    image_pull_secrets=self.image_pull_secrets,
    service_account_name=self.service_account_name,
    hostnetwork=self.hostnetwork,
    tolerations=self.tolerations,
    configmaps=self.configmaps,
    security_context=self.security_context,
    dnspolicy=self.dnspolicy,
    init_containers=self.init_containers,
    restart_policy='Never',
    schedulername=self.schedulername,
    pod_template_file=self.pod_template_file,
    priority_class_name=self.priority_class_name,
    pod=self.full_pod_spec,
).gen_pod()
```

KubernetesPodOperator

5. K8sExecutor & K8sPodOperator

가볍다

Airflow는 Lib.의존성이 없는 기본 이미지

Kerberos 의존성 제거 (Kerberos 인증은 Custom Pod가 가져가도록 구성)

5. K8sExecutor & K8sPodOperator

유지보수 비용 절감

Docker를 통한 Task간 독립성 보장

운영중인 여러 Airflow 통합 가능

5. K8sExecutor & K8sPodOperator

효율적 자원관리

기존 방법의 경우 webserver, scheduler, broker cluster, worker 상주

동적으로 task생성 후 사용한 자원 반납

5. K8sExecutor & K8sPodOperator

개발 효율성

Dag Script의 Template화 가능

(비 개발자는 자신들의 비즈니스에만 집중)

5. K8sExecutor & K8sPodOperator

Reference가 적다

좋은 자료 - 박근희님 blog (https://humbledude.github.io/blog/2019/07/12/airflow-on-k8s/)

좋은 자료 – Document (https://airflow.apache.org/docs/stable/)

구성이 상대적으로 까다로움

6. 저희는 이렇게 사용합니다

svc

pod
Airflow Webserver

secret
Git secret

github

pod
Airflow Scheduler

vol
DAGs Volume

pod
Airflow Worker

pod
Custom Pod

Harbor

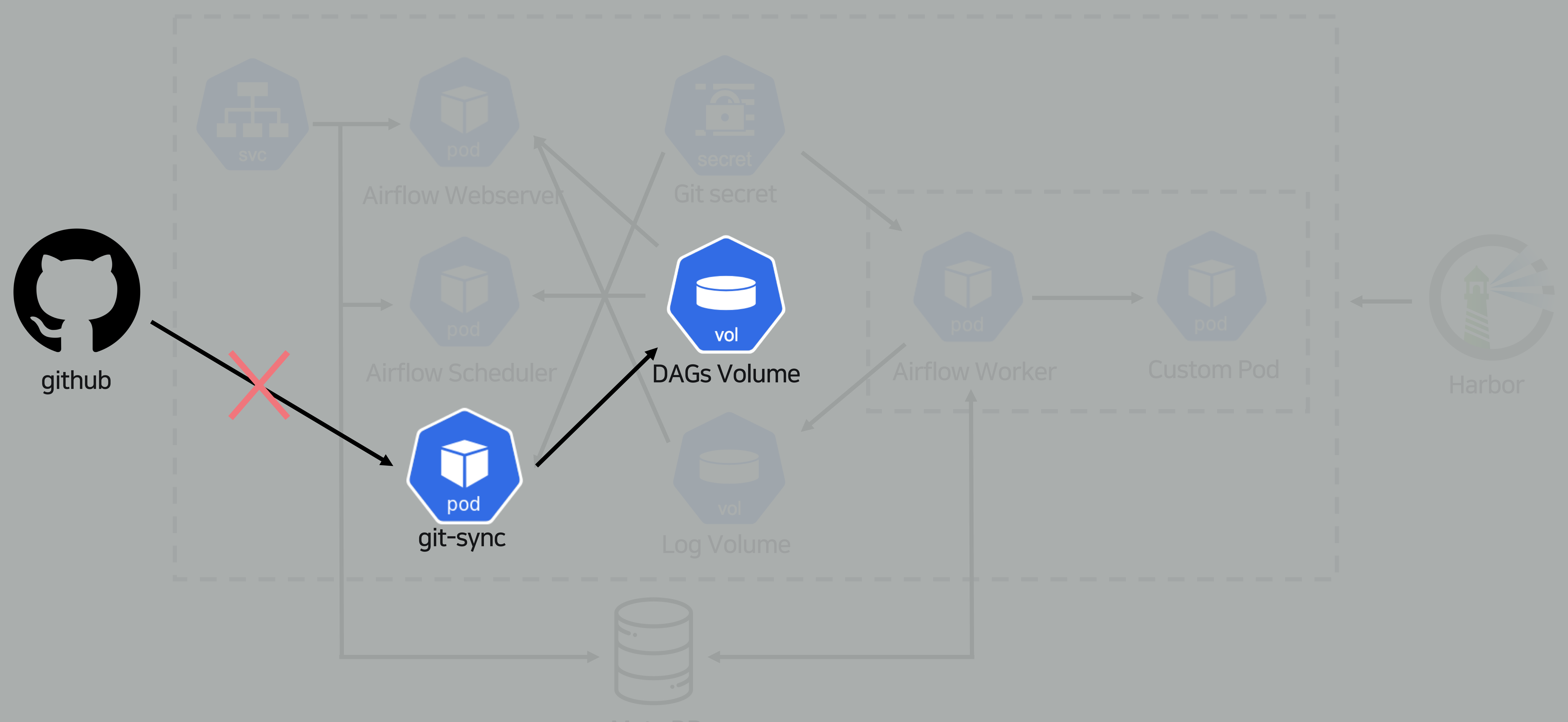
pod
git-sync

vol
Log Volume

Meta DB

6. 저희는 이렇게 사용합니다

DEVIEW 2020



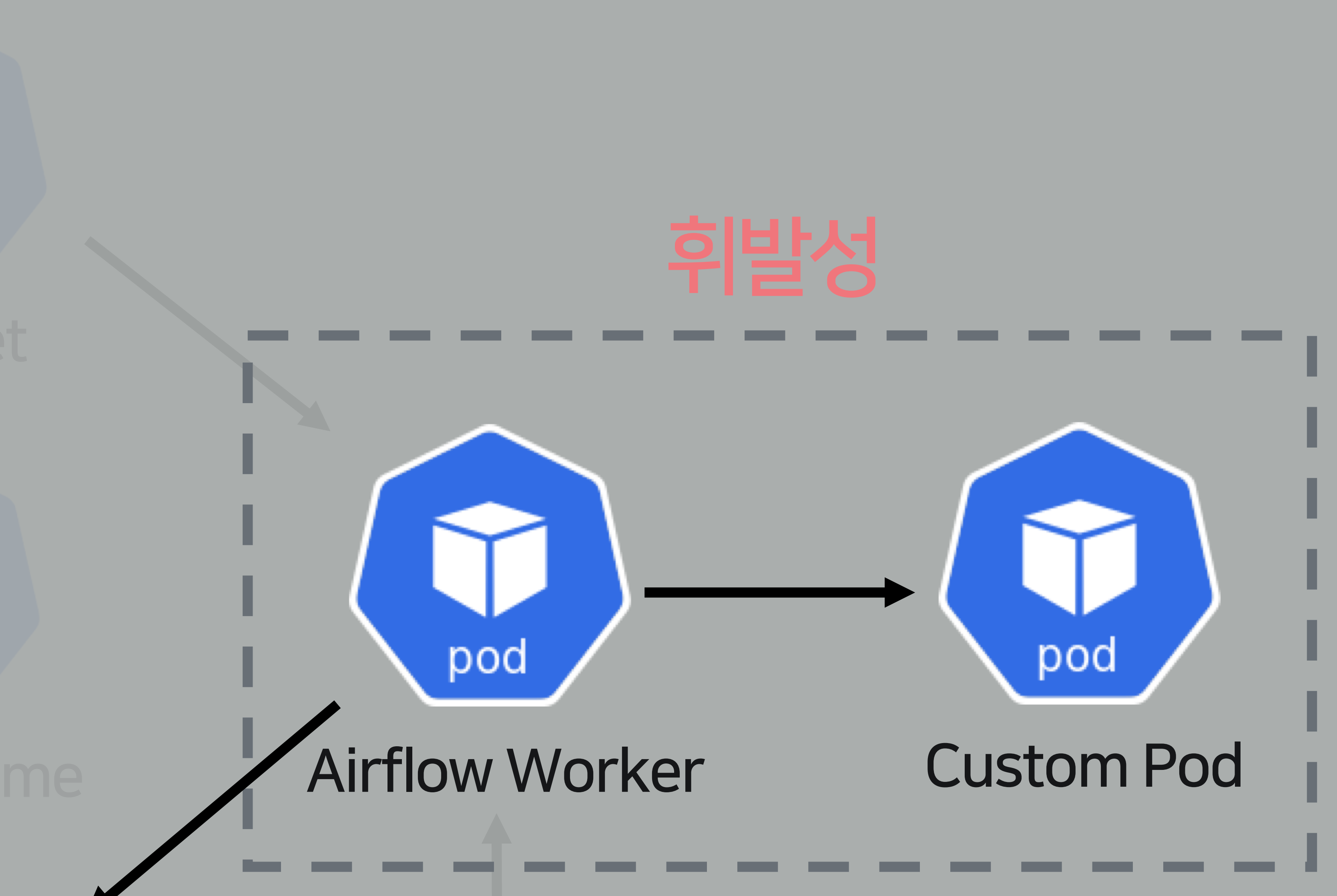
6. 저희는 이렇게 사용합니다

kubernetes_executor.py

```
self.dags_volume_claim = conf.get(self.kubernetes_section, 'dags_volume_claim')

self.dags_volume_mount_point = conf.get(self.kubernetes_section, 'dags_volume_mount_point')

# This prop may optionally be set for PV Claims and is used to write logs
self.logs_volume_claim = conf.get(self.kubernetes_section, 'logs_volume_claim')

# This prop may optionally be set for PV Claims and is used to locate DAGs
# on a SubPath
self.dags_volume_subpath = conf.get(
    self.kubernetes_section, 'dags_volume_subpath')

# This prop may optionally be set for PV Claims and is used to locate logs
# on a SubPath
self.logs_volume_subpath = conf.get(
    self.kubernetes_section, 'logs_volume_subpath')

# Optionally, hostPath volume containing DAGs
self.dags_volume_host = conf.get(self.kubernetes_section, 'dags_volume_host')

# Optionally, write logs to a hostPath Volume
self.logs_volume_host = conf.get(self.kubernetes_section, 'logs_volume_host')
```

kubernetes_pod_operator.py

```
try:
    launcher.start_pod(
        pod,
        startup_timeout=self.startup_timeout_seconds)
    final_state, result = launcher.monitor_pod(pod=pod, get_logs=self.get_logs)
except AirflowException as ex:
    if self.log_events_on_failure:
        for event in launcher.read_pod_events(pod).items:
            self.log.error("Pod Event: %s - %s", event.reason, event.message)
    raise AirflowException('Pod Launching failed: {error}'.format(error=ex))
finally:
    if self.is_delete_operator_pod:
        launcher.delete_pod(pod)
return final_state, pod, result
```



6. 저희는 이렇게 사용합니다

Airflow OpenSource 수정 사항

UTC Time zone To KST Time zone

Kubernetes worker ssh secret configuration

PodGenerator multiple command (v.1.10.11에 수정)

Kubernetes pod name underscore 허용

Web UI 수정 & Dag Like 검색 (최신 버전에 수정)

NONE_SKIPPED Trigger Rule추가 (최신 버전에 수정)

...

6. 저희는 이렇게 사용합니다

Airflow 성능 개선 방안

Kubernetes Executor & KubernetesPodOperator 사용 시
수 백 개의 DAG 환경에서 수 천개의 Task가 생성되면
Task Pending현상이 발생할 수 있습니다.
이 현상은 Kubernetes Scheduler의 Pod Assign알고리즘에 의해
특정 Node에만 Airflow Worker Pod들이 할당될 경우 발생할 수 있습니다.
그리고 Pod가 특정 Node에 쏠려서 할당되는 이유는 Airflow Worker 생성 시
별도로 Airflow Worker 리소스(cpu,memory)를 설정해주지 않기 때문에 발생합니다.
저희는 신규로 airflow.cfg에 worker resource config를 생성하고
이를 Airflow worker pod 생성 시 사용하도록 수정하여 성능 저하 현상을 해결하였습니다.

6. 저희는 이렇게 사용합니다

Airflow 성능 개선 방안

worker_configuration.py

```python
def as_pod(self):
    """Creates POD."""
    if self.kube_config.pod_template_file:
        return PodGenerator(pod_template_file=self.kube_config.pod_template_file).gen_pod()
    pod = PodGenerator(
        image=self.kube_config.kube_image,
        image_pull_policy=self.kube_config.kube_image_pull_policy or 'IfNotPresent',
        image_pull_secrets=self.kube_config.image_pull_secrets,
        volumes=self._get_volumes(),
        volume_mounts=self._get_volume_mounts(),
        init_containers=self._get_init_containers(),
        labels=self.kube_config.kube_labels,
        annotations=self.kube_config.kube_annotations,
        affinity=self.kube_config.kube_affinity,
        tolerations=self.kube_config.kube_tolerations,
        envs=self._get_environment(),
        node_selectors=self.kube_config.kube_node_selectors,
        service_account_name=self.kube_config.worker_service_account_name or 'default',
        restart_policy='Never'
    ).gen_pod()

    pod.spec.containers[0].env_from = pod.spec.containers[0].env_from or []
    pod.spec.containers[0].env_from.extend(self._get_env_from())
    pod.spec.security_context = self._get_security_context()
```

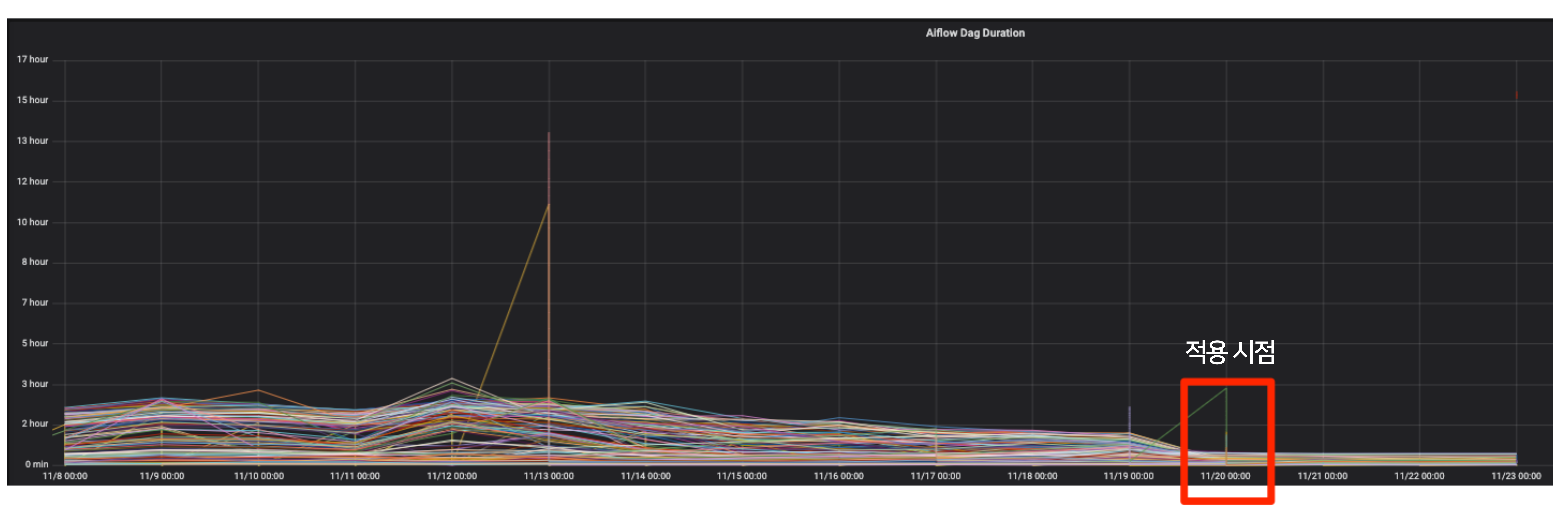
→

```python
def as_pod(self):
    """Creates POD."""
    if self.kube_config.pod_template_file:
        return PodGenerator(pod_template_file=self.kube_config.pod_template_file).gen_pod()

    resources = Resources(
            request_memory=self.kube_config.worker_request_memory,
            request_cpu=self.kube_config.worker_request_cpu,
            limit_memory=self.kube_config.worker_limit_memory,
            limit_cpu=self.kube_config.worker_limit_cpu
    )

    pod = PodGenerator(
        resources=resources,
        image=self.kube_config.kube_image,
        image_pull_policy=self.kube_config.kube_image_pull_policy or 'IfNotPresent',
        image_pull_secrets=self.kube_config.image_pull_secrets,
        volumes=self._get_volumes(),
        volume_mounts=self._get_volume_mounts(),
        init_containers=self._get_init_containers(),
        labels=self.kube_config.kube_labels,
        annotations=self.kube_config.kube_annotations,
        affinity=self.kube_config.kube_affinity,
        tolerations=self.kube_config.kube_tolerations,
        envs=self._get_environment(),
        node_selectors=self.kube_config.kube_node_selectors,
        service_account_name=self.kube_config.worker_service_account_name or 'default',
        restart_policy='Never'
    ).gen_pod()

    pod.spec.containers[0].env_from = pod.spec.containers[0].env_from or []
```

airflow.cfg 신규
config 생성

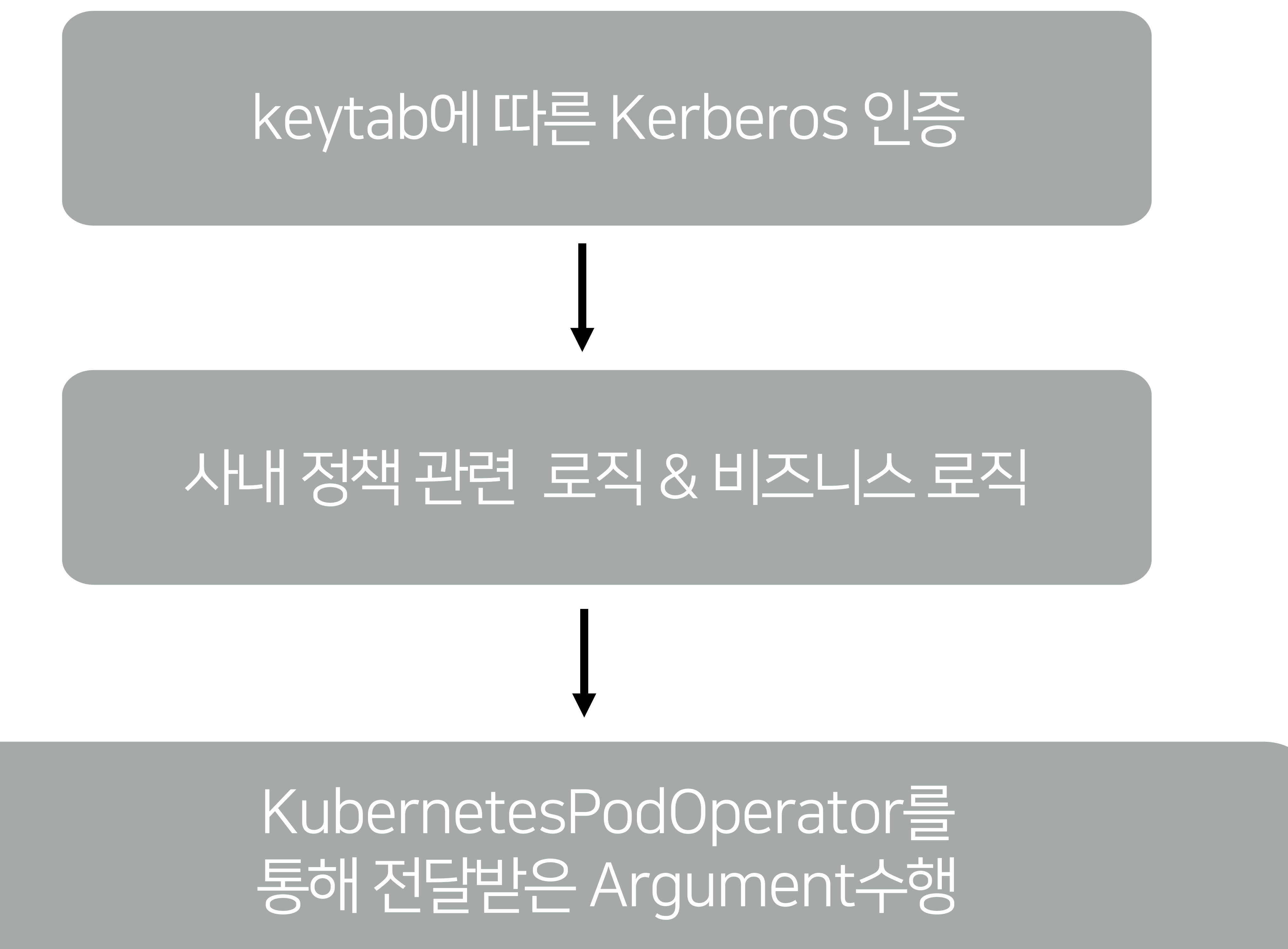
Airflow Worker 생성 시
user가 정의한
resource로 Pod생성

Airflow 성능 개선 효과



6. 저희는 이렇게 사용합니다

Custom Pod의 Entrypoint 공통 로직



6. 저희는 이렇게 사용합니다

DEVIEW 2020

Java or Scala Spark Application

- Spark Submit Command Tool을 별도로 개발하여 사용
- Docker Container 기동 시 해당 Command가 실행
- Spark Driver에 대한 자원 제어 가능
- Client Mode에서 Docker Host Network를 활성화하여 사용

```
spark_submit_sample = KubernetesPodOperator(
    task_id='spark_submit_sample',
    name='spark_submit_sample',
    namespace='airflow',
    image='spark_cluster1:1.0',
    arguments=["spark", SparkSubmitCommandTool.getCommand(SparkSubmitCommandTool(),
                                                          class_name='com.linecorp.spark.application.SampleApplication'
                                                          master='yarn',
                                                          keytab='user.keytab',
                                                          principal='user@FINANCIAL.HADOOP.DATA.COM',
                                                          deploy_mode='client',
                                                          driver_cores='1',
                                                          driver_memory='2g',
                                                          executor_cores='1',
                                                          executor_memory='4',
                                                          jars='hdfs://line_financial/spark_app/spark_sample.jar',
                                                          args=[execution_date],
                                                          conf=["spark.dynamicAllocation.enabled=true",
                                                                "spark.shuffle.service.enabled=true",
                                                                "spark.dynamicAllocation.minExecutors=5",
                                                                "spark.dynamicAllocation.maxExecutors=10",
                                                                "spark.rpc.message.maxSize=256"])],
    resources={'request_cpu': '1000m',
               'request_memory': '2048Mi',
               'limit_cpu': '4096Mi',
               'limit_memory': '2000m'},
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=True,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    on_retry_callback=SlackTool.make_handler(config.notification.slack_channel, color='warning', message="Retry Task"),
    dag=dag
)
```

6. 저희는 이렇게 사용합니다

PySpark Application

- Spark Project와 Airflow Project를 통합 가능
- Airflow Project내의 실행하고 싶은 PySpark Application Script를 Dag에서 지정하여 실행

```
pyspark_submit_sample = KubernetesPodOperator(
    task_id='pyspark_submit_sample',
    name='pyspark_submit_sample',
    namespace='airflow',
    image='spark_cluster1:1.0',
    arguments=["pyspark", "pyspark_sample.py"],
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=True,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    on_retry_callback=SlackTool.make_handler(config.notification.slack_channel, color='warning', message="Retry Task"),
    dag=dag
)
```

6. 저희는 이렇게 사용합니다

DEVIEW 2020

Hadoop Control

- distcp, mkdir, rm, test, ls ...
- Command Tool을 별도로 개발하여 사용
- Docker Container 기동 시 해당 Command가 실행

```
hadoop_distcp_sample = KubernetesPodOperator(
    task_id='hadoop_distcp_sample',
    name='hadoop_distcp_sample',
    namespace='airflow',
    image='hadoop_cluster1:1.0',
    arguments=["hadoop_command", HadoopDistcpCommandTool.getCommand(HadoopDistcpCommandTool(),
                                                                    direction='cluster1_to_cluster2',
                                                                    src='src_path',
                                                                    dst='dst_path',
                                                                    options={'-Dmapred.job.queue.name=root.default',
                                                                             '-overwrite',
                                                                             '-strategy dynamic'},
                                                                    args={
                                                                        'YESTERDAY': execution_date,
                                                                        'TODAY': execution_date_plus_1day
                                                                    })],
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=False,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    on_retry_callback=SlackTool.make_handler(config.notification.slack_channel, color='warning', message="Retry Task"),
    dag=dag
)
```

6. 저희는 이렇게 사용합니다

Python Application

- 원하는 Python Version & Lib.가 설치된 이미지 사용
- Airflow Project내의 실행하고 싶은 Python Application Script를 Dag에서 지정하여 실행

```
python2_app_sample = KubernetesPodOperator(
    task_id='python2_app_sample',
    name='python2_app_sample',
    namespace='airflow',
    image='python2:1.0',
    arguments=["python2", "python2_app_sample.py"],
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=True,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    dag=dag
)
```

Python2 Application

```
python3_app_sample = KubernetesPodOperator(
    task_id='python3_app_sample',
    name='python3_app_sample',
    namespace='airflow',
    image='python3:1.0',
    arguments=["python3", "python3_app_sample.py"],
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=True,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    dag=dag
)
```

Python3 Application

6. 저희는 이렇게 사용합니다

DEVIEW 2020

HIVE & Beeline

- Beeline OR Hive Command Tool을 별도로 개발하여 사용
- Airflow Project내의 실행하고 싶은 SQL file을 Dag에서 지정하여 실행
- Docker Container기동 시 해당 Command가 실행

```
beeline_sample = KubernetesPodOperator(
    task_id='beeline_sample',
    name='beeline_sample',
    namespace='airflow',
    image='hive_cluster1:1.0',
    arguments=["beeline", BeelineCommandTool.getCommand(BeelineCommandTool(),
                                                        service_name='fintech_service',
                                                        hql_file_name='hv_daily_sample.hql',
                                                        conn_id='jdbc_conn_id',
                                                        args={
                                                            'YESTERDAY': execution_date,
                                                            'DB_NAME': 'sample_db',
                                                            'TABLE_NAME': 'sample_table'
                                                        })],
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=True,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    on_retry_callback=SlackTool.make_handler(config.notification.slack_channel, color='warning', message="Retry Task"),
    dag=dag
)
```

6. 저희는 이렇게 사용합니다

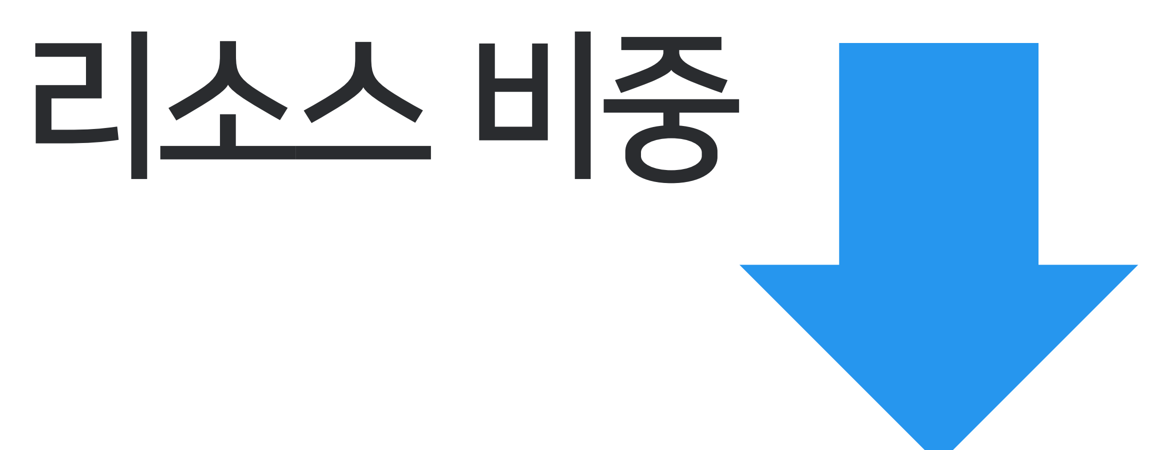
DEVIEW 2020

Sqoop

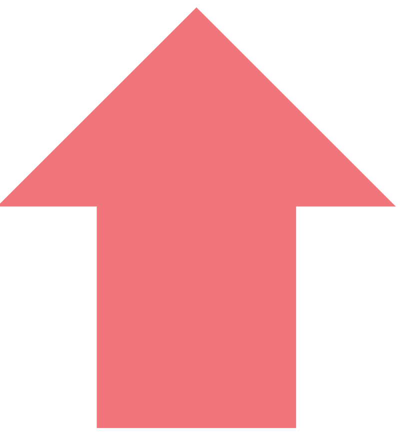
- Sqoop Command Tool을 별도로 개발하여 사용
- Airflow Project내의 실행하고 싶은 Sqoop Meta Information file을 Dag에서 지정하여 실행
- Docker Container기동 시 해당 Command가 실행

```
sqoop_sample = KubernetesPodOperator(
    task_id='sqoop_sample',
    name='sqoop_sample',
    namespace='airflow',
    image='sqoop_cluster1:1.0',
    arguments=["sqoop", ScoopCommandTool.getCommand(ScoopCommandTool(),
                                                eval_type='all',
                                                service_name='fintech_service',
                                                file_name='sqoop_sample_definition.yml'
                                                args={
                                                    'YESTERDAY': execution_date,
                                                    'FROM': execution_date,
                                                    'TO': execution_date_plus_1day
                                                })],
    in_cluster=True,
    is_delete_operator_pod=True,
    startup_timeout_seconds=180,
    get_logs=True,
    image_pull_policy='IfNotPresent',
    service_account_name='airflow',
    hostnetwork=False,
    execution_timeout=timedelta(minutes=120),
    retries=2,
    retry_delay=timedelta(minutes=2),
    on_retry_callback=SlackTool.make_handler(config.notification.slack_channel, color='warning', message="Retry Task"),
    dag=dag
)
```

6. 저희는 이렇게 사용합니다



타 프로젝트 리소스 비중



6. 저희는 이렇게 사용합니다

동료를 찾습니다.

Global Fintech
데이터 플랫폼 개발

[담당업무]
- 여러 서비스에 걸쳐있는 사용자의 데이터와 금융 데이터를 수집하고 적재하는 시스템 구축
- 적재된 데이터를 분석/처리할 수 있는 시스템 구축
- 분석된 데이터를 LINE의 Scoring 및 다양한 Fintech 서비스에 활용

[자격요건]
- 개발 경력 3년이상
- Hadoop ecosystem 에 대한 이해(HDFS,YARN,MapReduce,HBase, Spark,Hive 등)와 경험이 있는 분
- 대용량 트래픽/데이터 처리 경험이 있는 분
- Java 언어, Java Framework 기반 개발에 능숙한 분
- 멤버간 수평 구조를 중시하고, 유연한 사고를 가진 분

[우대사항]
- 모바일 금융 서비스에 익숙하고 관련 최신 트렌드 관심
- 금융 데이터 분석과 관련된 경험
- 빅데이터 / 머신러닝 / 딥러닝에 대한 관심
- Data Warehouse 및 Data Mart 경험
- 외국어로 업무 진행이 가능한 분 (영어, 일본어)

Thank You

Q & A