# Kubernetes Executor Proposal

## Background:

Kubernetes is a container-based cluster management system designed by google for easy application deployment. Companies such as Airbnb, Bloomberg, Palantir, and Google use kubernetes for a variety of large-scale solutions including data science, ETL, and app deployment. Integrating airflow into Kubernetes would increase viable use cases for airflow, promote airflow as a de facto workflow scheduler for Kubernetes, and create possibilities for improved security and robustness within airflow.

## Kubernetes Executor:

### Kubernetes Api:

We will communicate with Kubernetes using the Kubernetes python client. This client will allow us to create, monitor, and kill jobs. Users will be required to either run their airflow instances within the kubernetes cluster, or provide an address to link the API to the cluster.

### Launching Jobs:

Unlike the current MesosExecutor, which uses pickle to serialize DAGs and send them to pre-built slaves, the KubernetesExecutor will launch a new temporary job for each task. Each job will have contain a full airflow deployment and will run an **airflow run <dag_id> <task_id>** command.This design has two major benefits over the previous system. The first benefit is that dynamically creating airflow workers simplifies the cluster set-up. Users will not need to pre-build airflow workers or consider how the nodes will communicate with eachother. The second benefit is that dynamically creating pods allows for a highly elastic system that can easily scale to large workloads while not wasting resources during periods of low usage.

### Monitoring jobs:

When we create the Kubernetes jobs, we will maintain a mapping of job_id -> job key. Using these job ids we can use the **read_namespaced_jobs** endpoint to consistently query kubernetes for the status of running jobs. Upon recieving a failure or success status from the API, the executor can forward the given state to the scheduler to show in the UI. By using the airflow batch job API (as opposed to launching pods), we get an assurance that any failed kubernetes job can retry a pre-set number of times before the executor kills the task.

```
for job_id in current_jobs:
    status = api.read_namespaced_job(job_id, namespace).status
    process_status(job_id, key, status)
```

## Sharing Dags:

To encourage a wide array of potential storage options for airflow users, we will take advantage of kubernetes persistent volume claims. With these claims, users will be allowed to use a wide variety of distributed storage options such as github, EBS, cinder, NFS, and glusterFS. We will offer a few initial options (such as github and cinder), but will also create a "kubernetes_volume" plugin for users that wish to use other distributed file systems.

### Security:

Kubernetes offers multiple inherent security benefits that would allow airflow users to safely run their jobs with minimal risk. By running airflow instances in non-default namespaces, administrators can populate those namespaces with only the secrets required to access data that is allowed for a user or role-account. We could also further restrict access using airflows' multi-tenancy abilities and kerberos integration.

## Kubernetes Operator

The Kubernetes operator will have a very straightforward implementation. In the same way that DAG folders are all placed within the $AIRFLOW_HOME/dags folder, kubernetes yaml files should be placed in a $AIRFLOW_HOME/kub-yaml folder. This means that a user only needs to identify the name of the yaml file to launch the kubernetes job.

```
PodOperator
def __init__(
        self,
        trigger_dag_id,
        yaml_name,
        *args, **kwargs):
    super(KubernetesOperator, self).__init__(*args, **kwargs)
    self.yaml_url = yaml_url
    self.trigger_dag_id = trigger_dag_id
    self.container_id = self.task_id
```

Another option that will give more flexibility for users that do not want to use yamls will be to offer first class kubernetes classes that will create and launch pods/jobs at will.

```
class Pod():
    def __init__(
            self,
            image,
            envs = {},
            cmds = [],
            secrets = [],
         labels = {},
            node_selectors = {},
         kube_req_factory = None,
            name = None,
         namespace = 'default',
            result = None):

 def launch_pod(self):
 def _execution_finished(self):

class PodOperator(BaseOperator):
```

# Questions posed by the airflow team:

**What tools would we provide to allow users to launch their own docker images/clusters?:**

- We intend to build two new operators for dealing with internal docker images and kubernetes pods. A KubernetesOperator and a HelmOperator. Using the KubernetesOperator the user will either point to a yaml file in the $AIRFLOW_HOME/yaml folder. While we believe this capability should be placed into a separate parallel PR (since it is a different feature), merging the KubernetesExecutor will be a powerful first step towards making kubernetes a first class citizen in airflow.

**How will the scheduler parse DAGs that have external dependencies (i.e. DAGS that require third party imports)? Currently our plan is to restrict dependencies.**

- We agree that complexity should be kept away from the actual DAG creation/parsing. For further complexity, users will be able to use the KubernetesOperator to customize within their own sandboxes. Using unknown add-ons could also have an adverse affect on airflow stability and security.

**If we want to use Kubernetes properly, then there won't be special resources on the hosts that are shared (e.g. can run airflow alongside other types of kubernetes pods). The problem with this is the whole DAG folder needs to be fetched on every worker which could cause a lot of load and increase task latency time.**

- With this set up, each worker only retrieves/parses the parent DAG its assigned task. This would minimize the network overhead to maintain the system.

**Docker image deployment/rollbacks (e.g. if upgrading your airflow docker image, how to handle long-running tasks, wait for them to finish/time them out and then restart them using the new docker image? Airflow would need to support retries that don't count as**

**failures in this case)**

- Users could handle new roll-outs by implementing a separate airflow pod, setting all not-currently-running jobs to only run on the replacement pod, and destroying the old deployment when all jobs are finished running.

**Task logging, right now logs are stored locally/in s3 but can't fetch local logs from kubernetes (our intern is working on making this better)**

- AirBnb currently has an airflow team-member working on ELK integration for airflow-kubernetes.

**If an airflow worker fails it might be useful to keep the kubernetes worker reserved and preserved in it's same state for debugging purposes**

- Kubernetes is primarily meant to run stateless applications. To our current knowledge there is no way to preserve state for a kubernetes job.

**Other interesting points:**
**The Airflow Kubernetes executor should try to respect the resources that are set in tasks for scheduling when hitting the kubernetes API**

# Future work

## Spark-On-K8s integration:

Teams at Google, Palantir, and many others are currently nearing release for a beta for spark that would run natively on kubernetes. This application would allow users to submit spark-submit commands to a resource manager that can dynamically spawn spark clusters for data processing. A seperate spark-on-k8s hook can be developed to sit within the SparkSubmitOperator depending on user configurations.