Democratized data workflows at scale

Emil Todorov

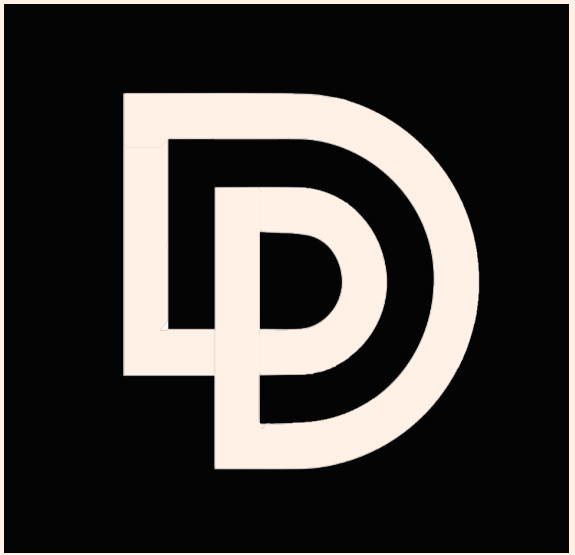
Mihail Petkov

FT
FINANCIAL TIMES

Our agenda for today

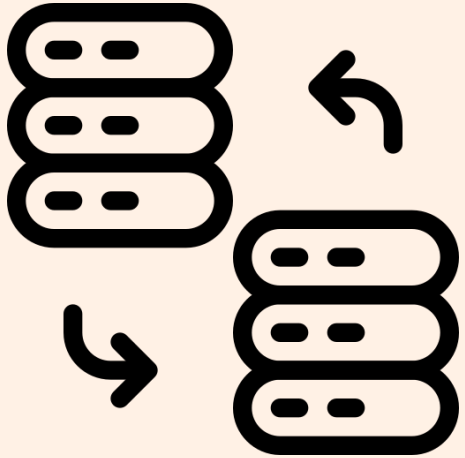
- Why Airflow?
- Architecture
- Security
- Execution environment in Kubernetes

FT



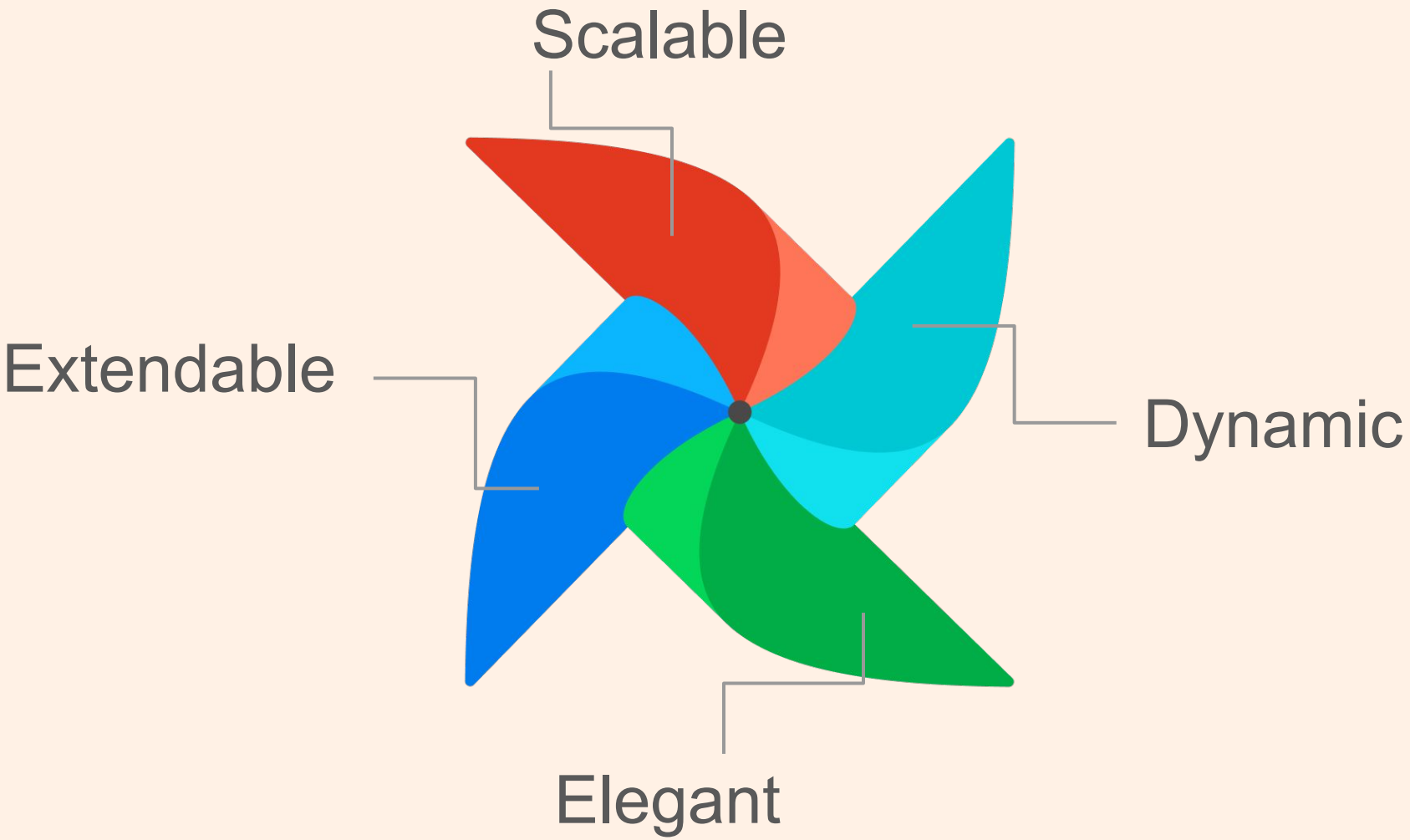
FT is a data driven organization

Time for a change



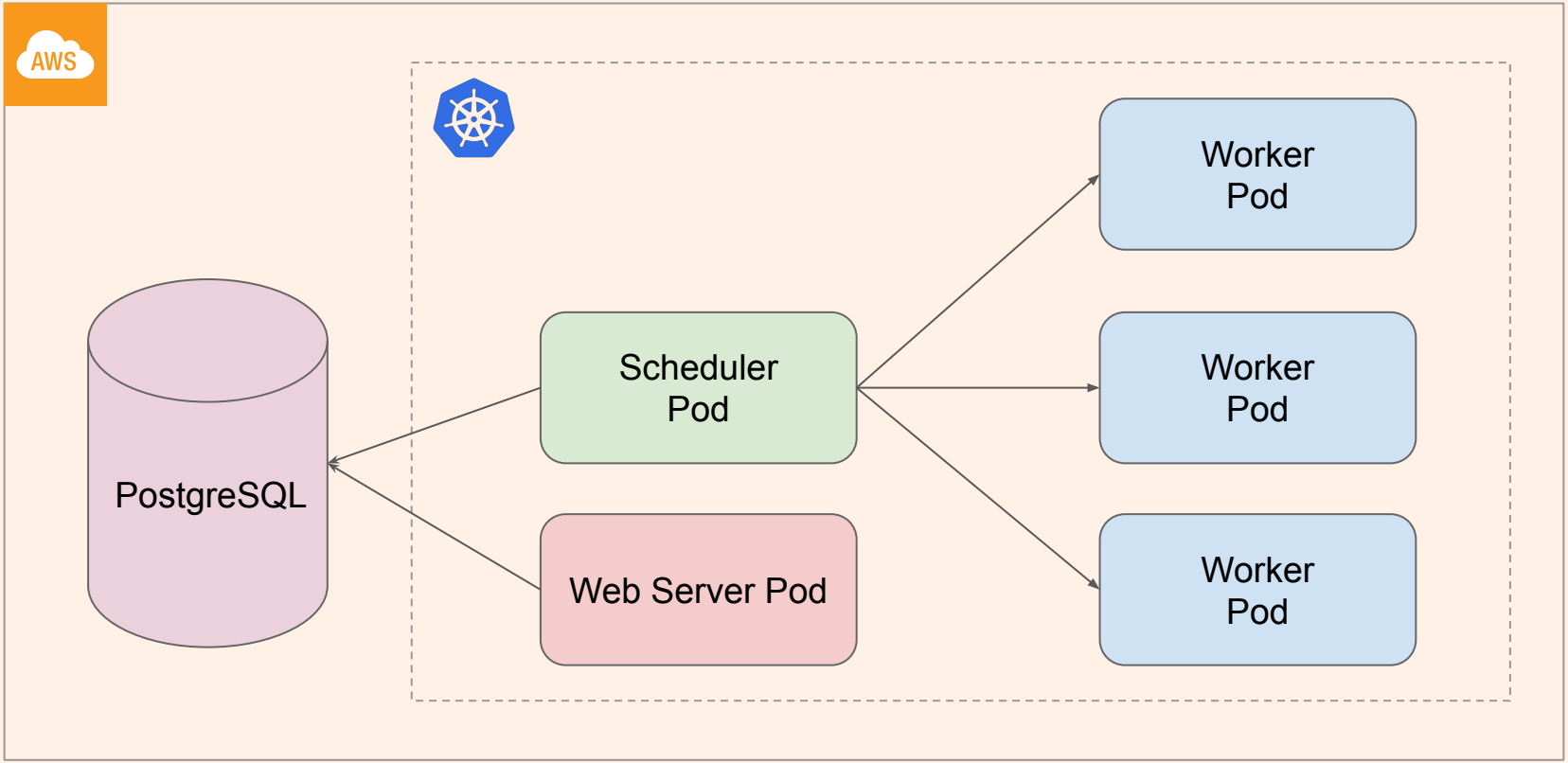
Apache
Airflow

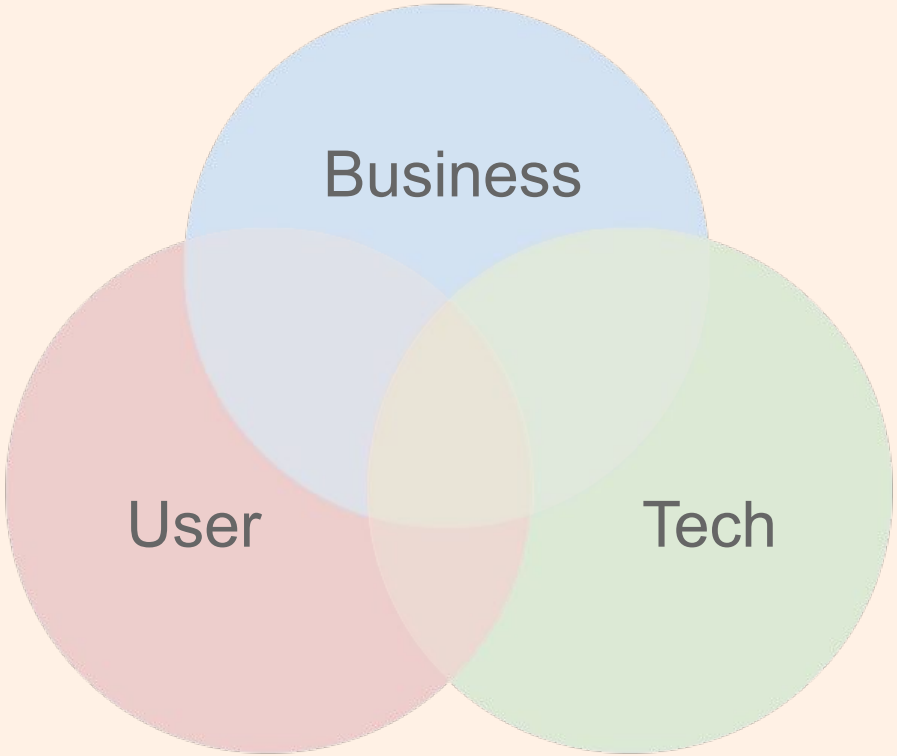
Why Airflow?


Scalable
Extendable
Dynamic
Elegant

Architecture

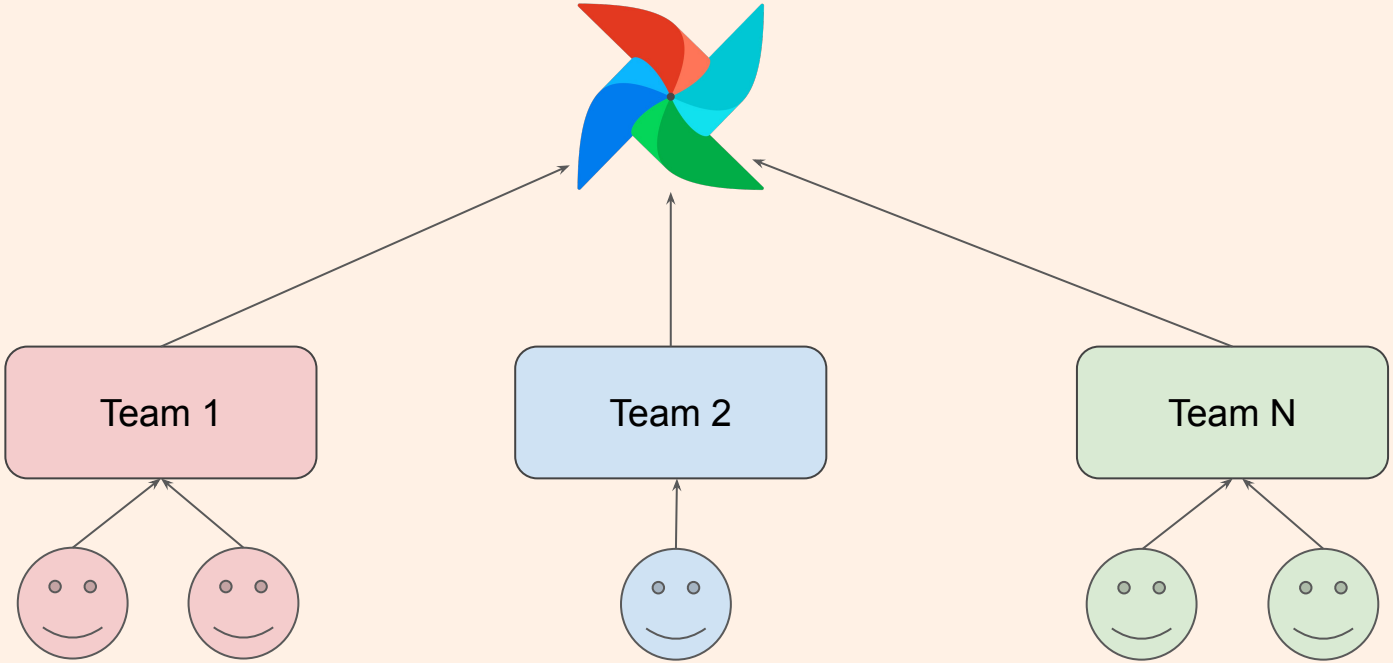
Architecture

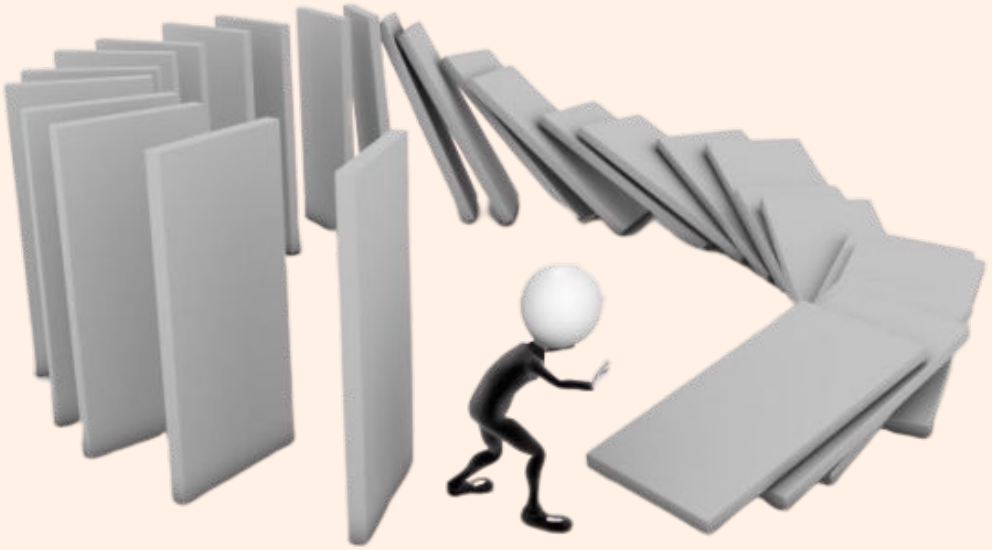



Business
User
Tech

Airflow will be used by multiple teams

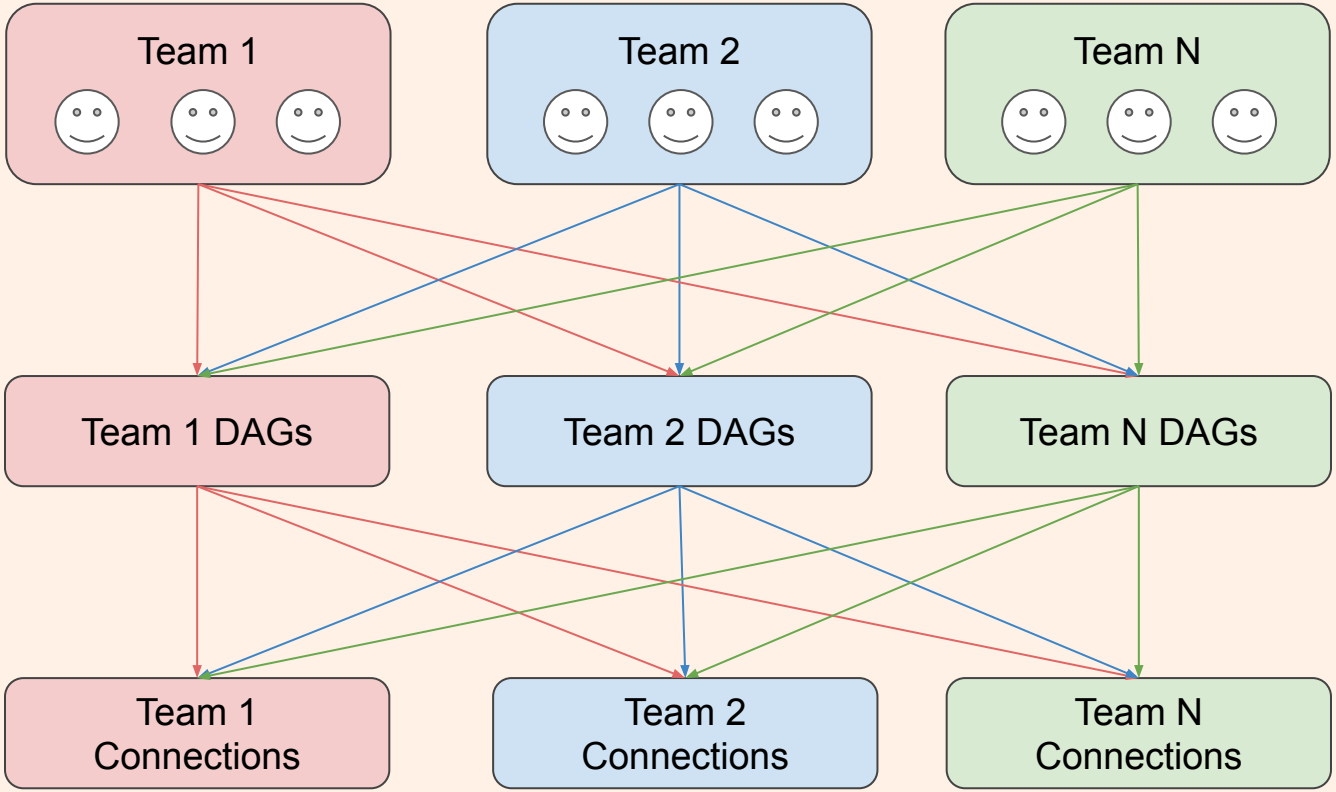
Airflow requirements





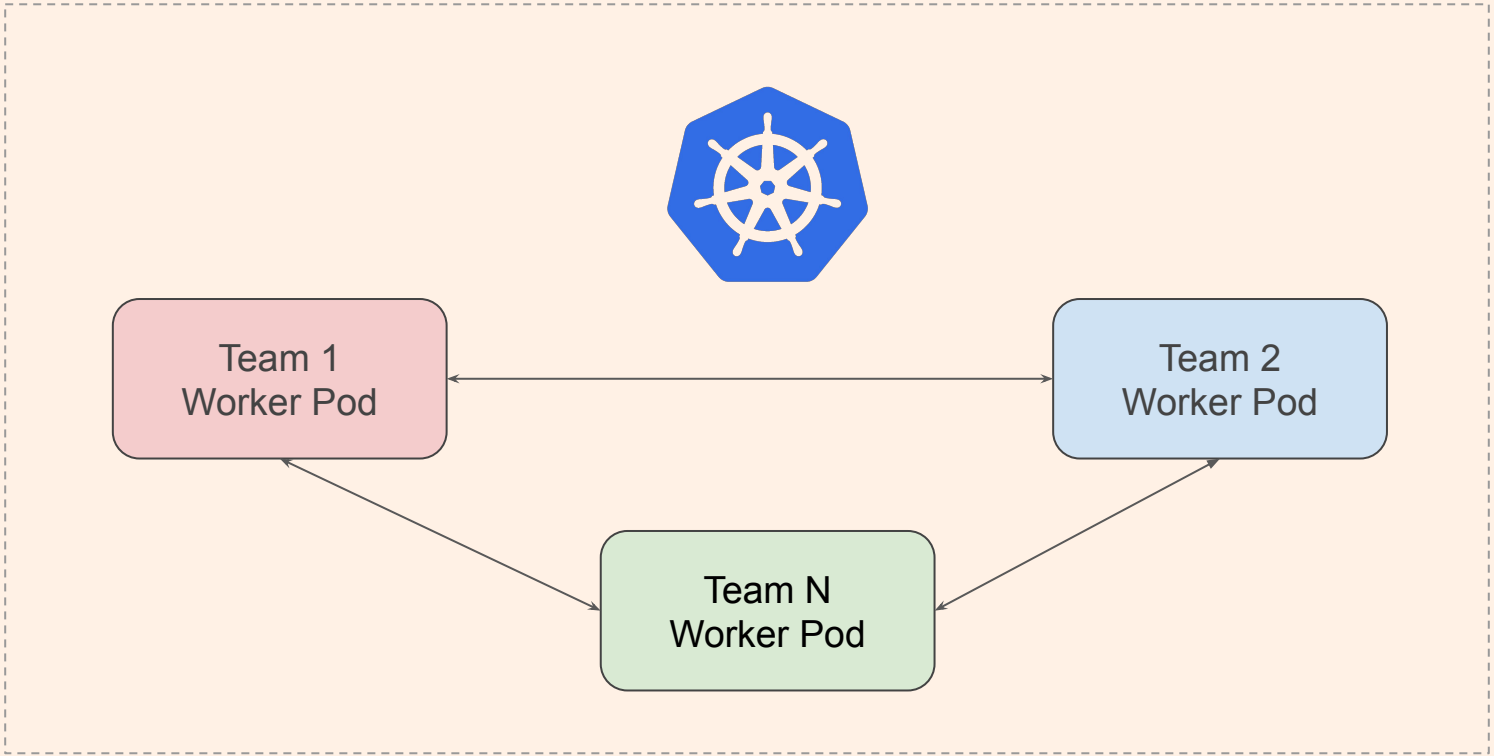
Teams will share Airflow resources

Airflow shared components



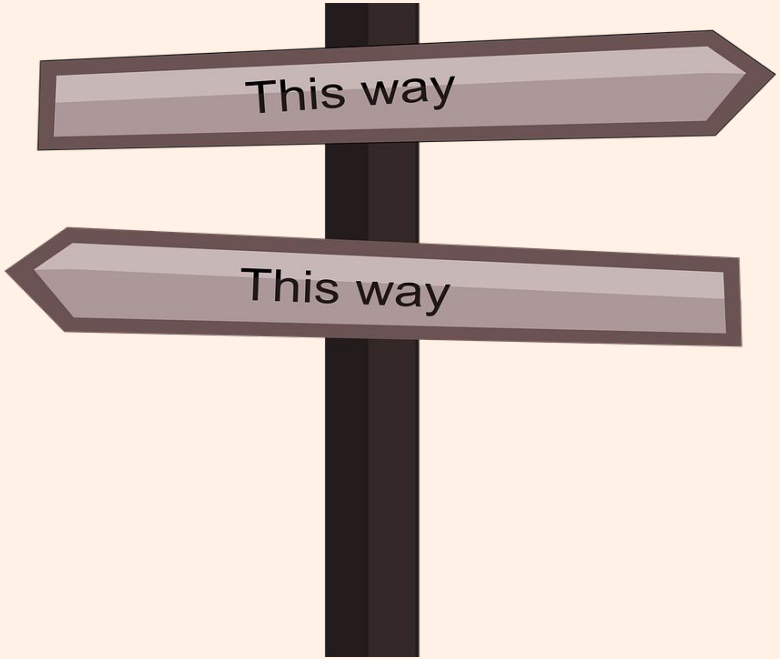
Teams will share Kubernetes resources

Kubernetes shared components



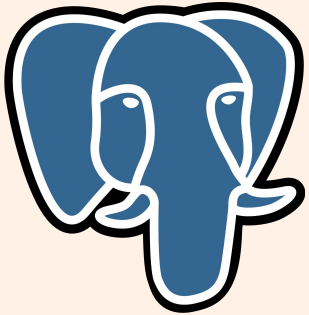
ISSUE
SECURITY • SECURITY

How to evolve this architecture?


This way
This way

Airflow instance per team

One instance components



Instance per team problems

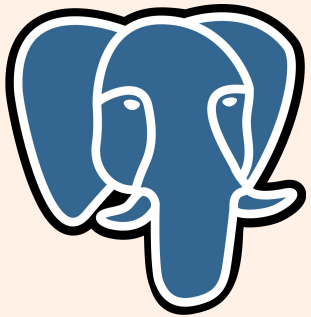
- Adding new team is **hard**
- Maintaining environment per team is **difficult**
- Releasing new features is **slow**
- Resources are **not fully utilised**
- Total **cost increase**

Another way?

Multitenancy

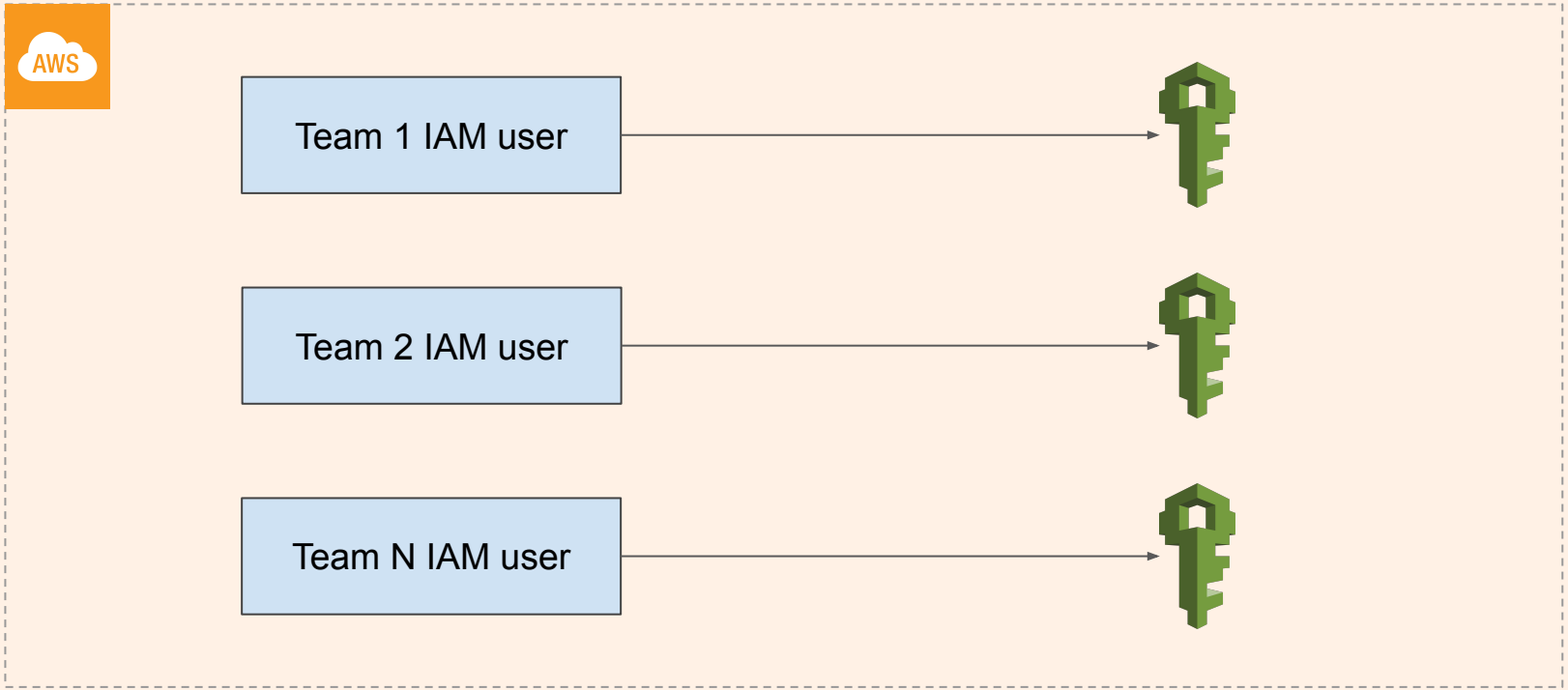
Multiple independent instances in a shared environment

Multi-tenant components

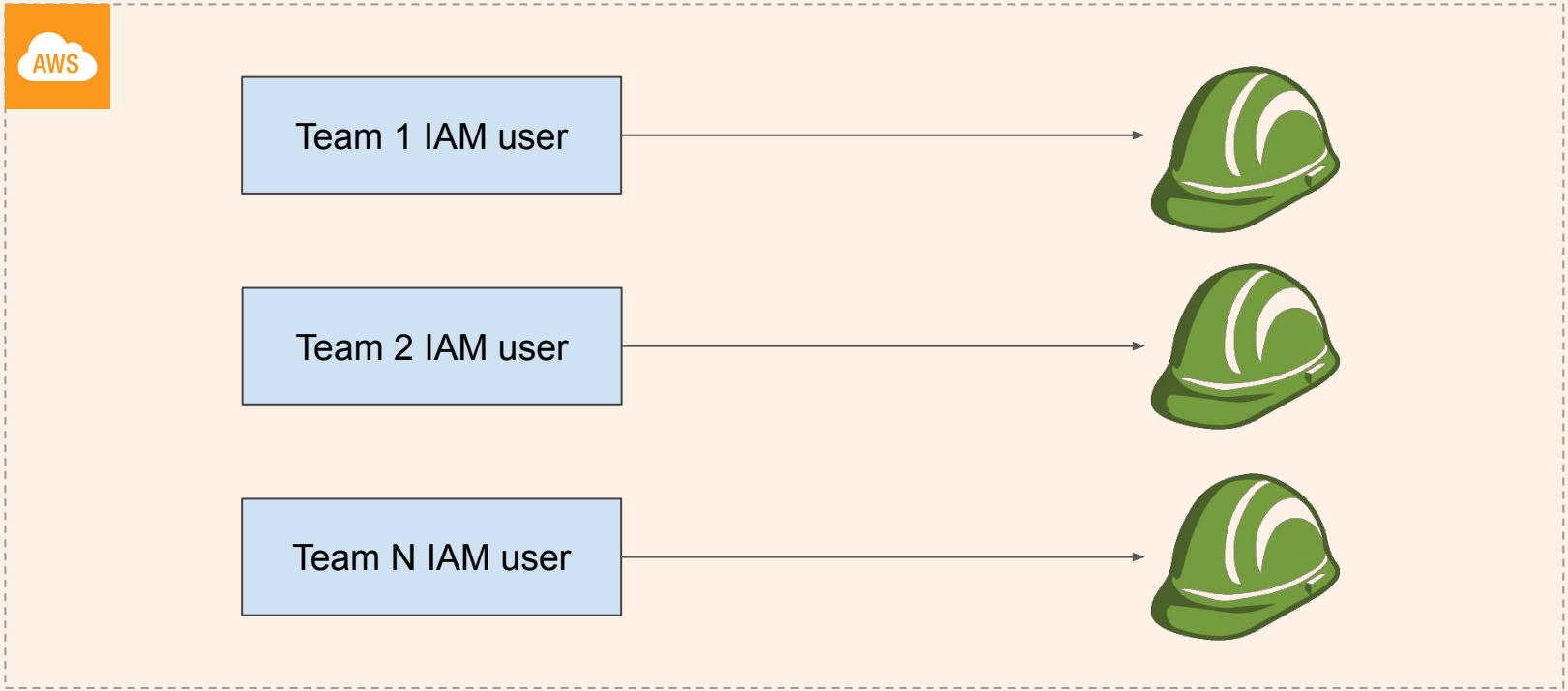


How to make AWS multi-tenant?

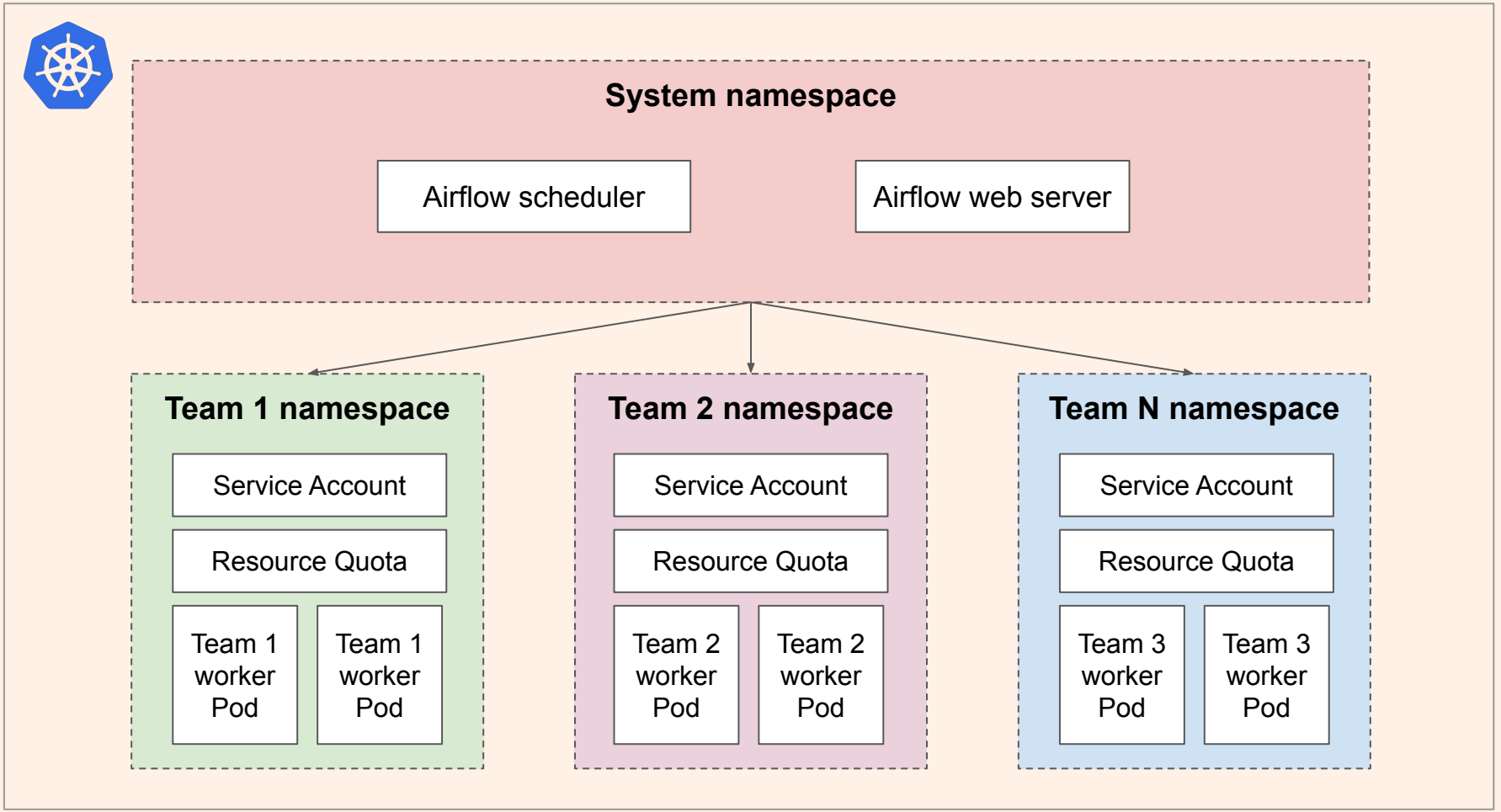
IAM Security



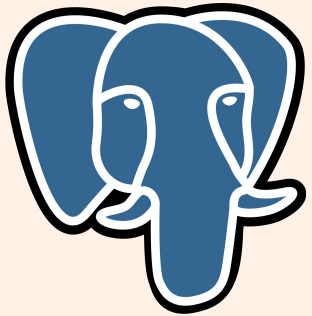
IAM Security



How to enhance Kubernetes?


System namespace
Airflow scheduler
Airflow web server
Team 1 namespace
Service Account
Resource Quota
Team 1 worker Pod
Team 1 worker Pod
Team 2 namespace
Service Account
Resource Quota
Team 2 worker Pod
Team 2 worker Pod
Team N namespace
Service Account
Resource Quota
Team 3 worker Pod
Team 3 worker Pod

How to improve PostgreSQL?



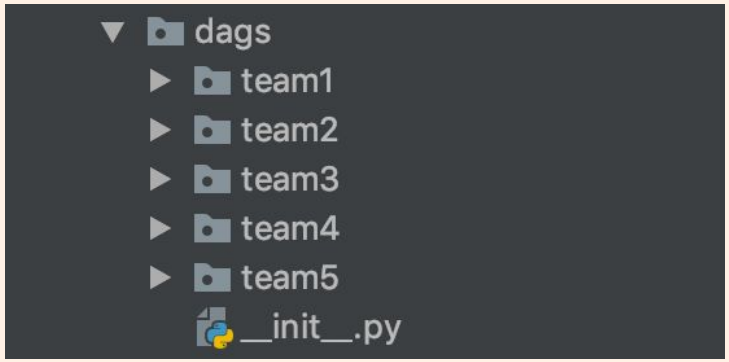
CHANGES

How to extend Airflow?

Redesign Airflow source code

Redesign Airflow source code

- Module per team



Redesign Airflow source code

- Module per team
- Connections per team

```
class ExtendedConnection(Connection):

    @staticmethod
    def get(conn_id: str) -> str:
        team_id = DAGMetaService.get_team_id_from_dag()
        return team_id + '_' + conn_id
```

Redesign Airflow source code

- Module per team
- Connections per team
- Extend hooks, operators and sensors

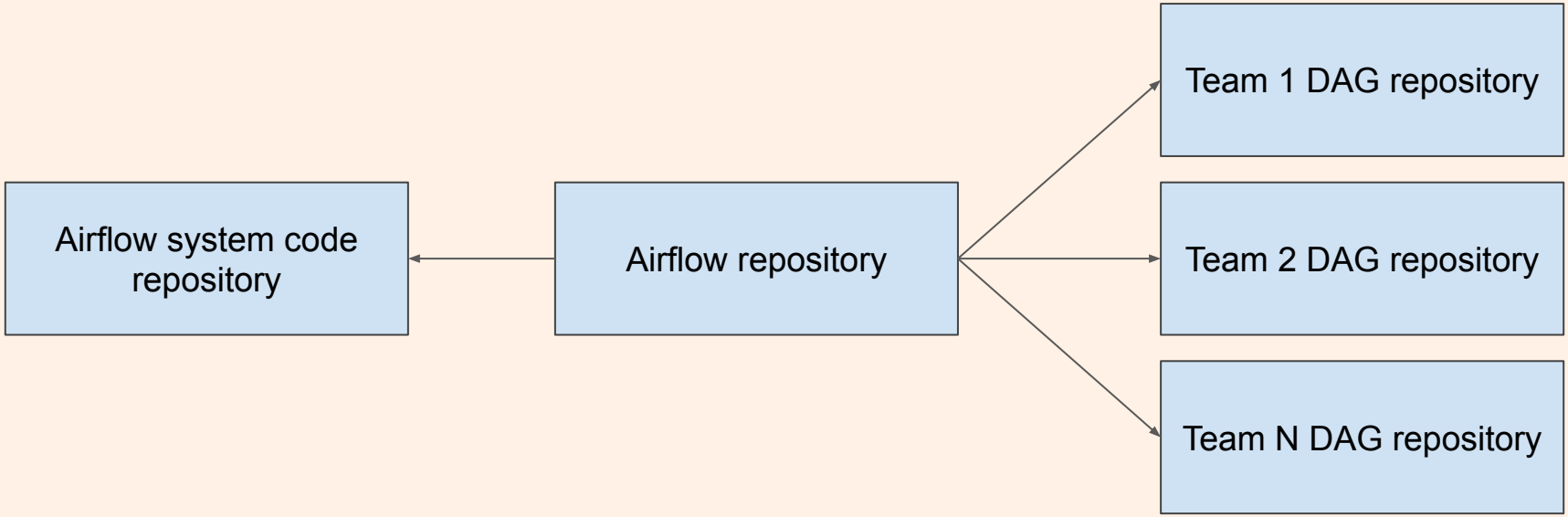
```
class ExtendedS3Hook(S3Hook):

    def __init__(self, *args, **kwargs) -> None:
        super().__init__(self, *args, **kwargs)
        self.aws_conn_id = ExtendedConnection.get(self.aws_conn_id)
```

Redesign Airflow source code

- Module per team
- Connections per team
- Extend hooks, operators and sensors
- Use **airflow_local_settings.py**

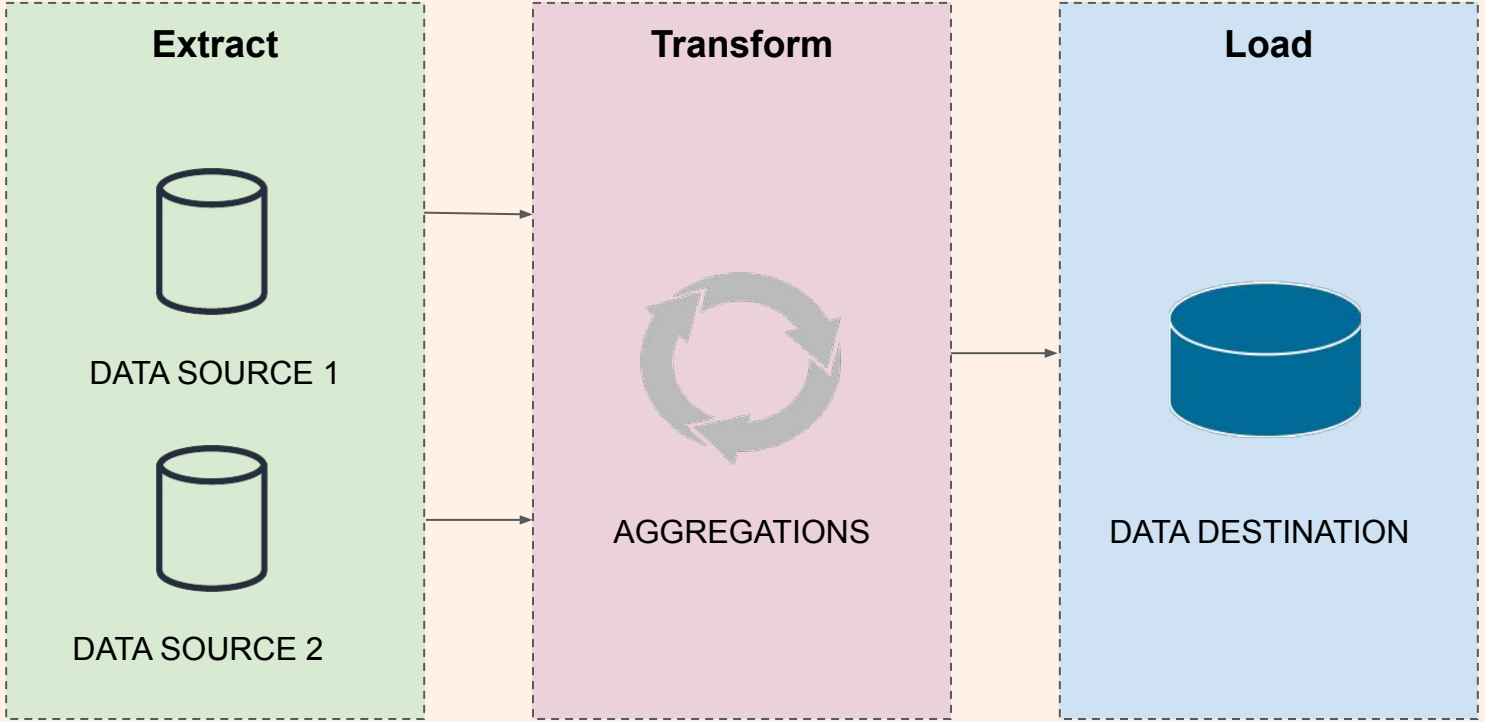
```python
def policy(task_instance: TaskInstance):
    team_id = get_team_id_from_dag_filepath(task_instance.dag.filepath)
    task_instance.executor_config['KubernetesExecutor']['labels']['team_id'] = team_id


def pod_mutation_hook(pod: Pod):
    team_id = pod.labels.get('team_id')
    pod.namespace = get_team_namespace(team_id)
```

Redesign repository structure

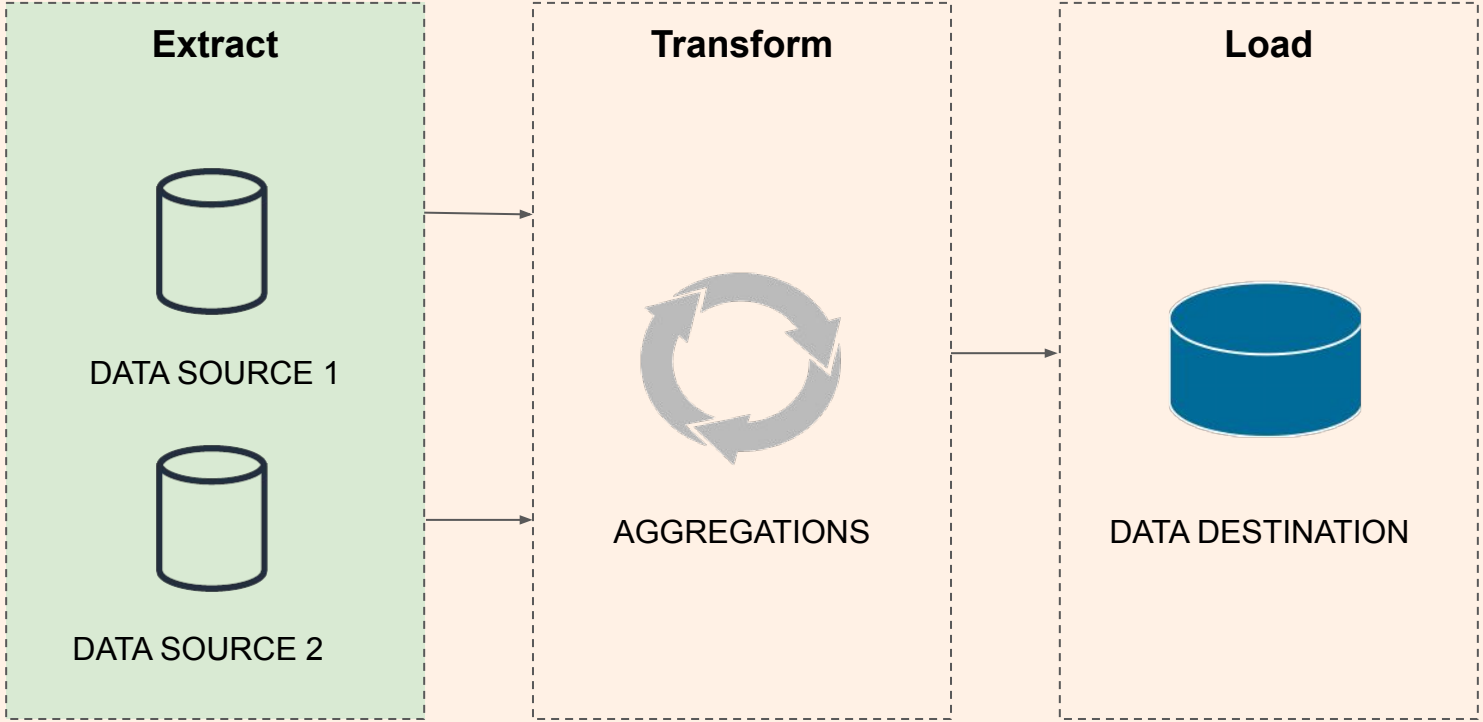


Execution environment in Kubernetes

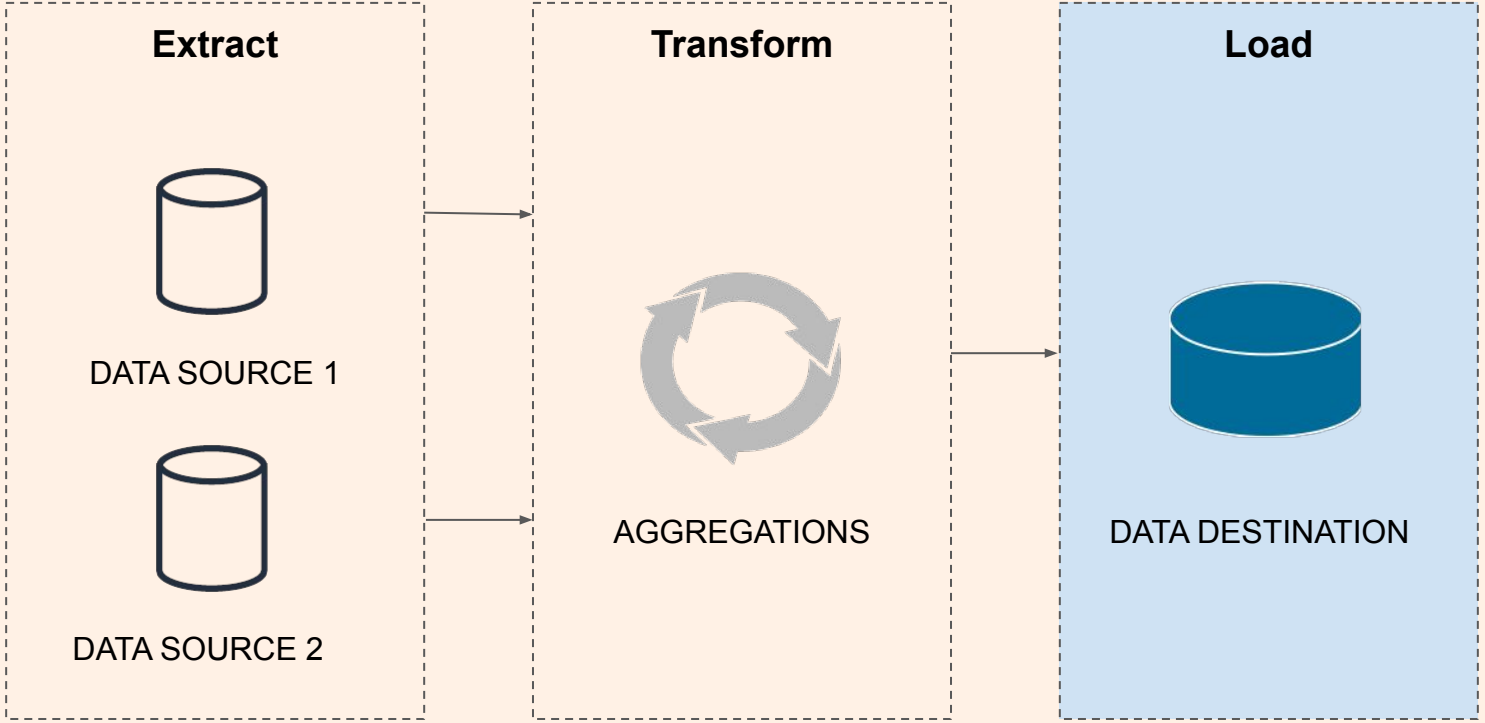
ETL


Extract
DATA SOURCE 1
DATA SOURCE 2
Transform
AGGREGATIONS
Load
DATA DESTINATION

Extract

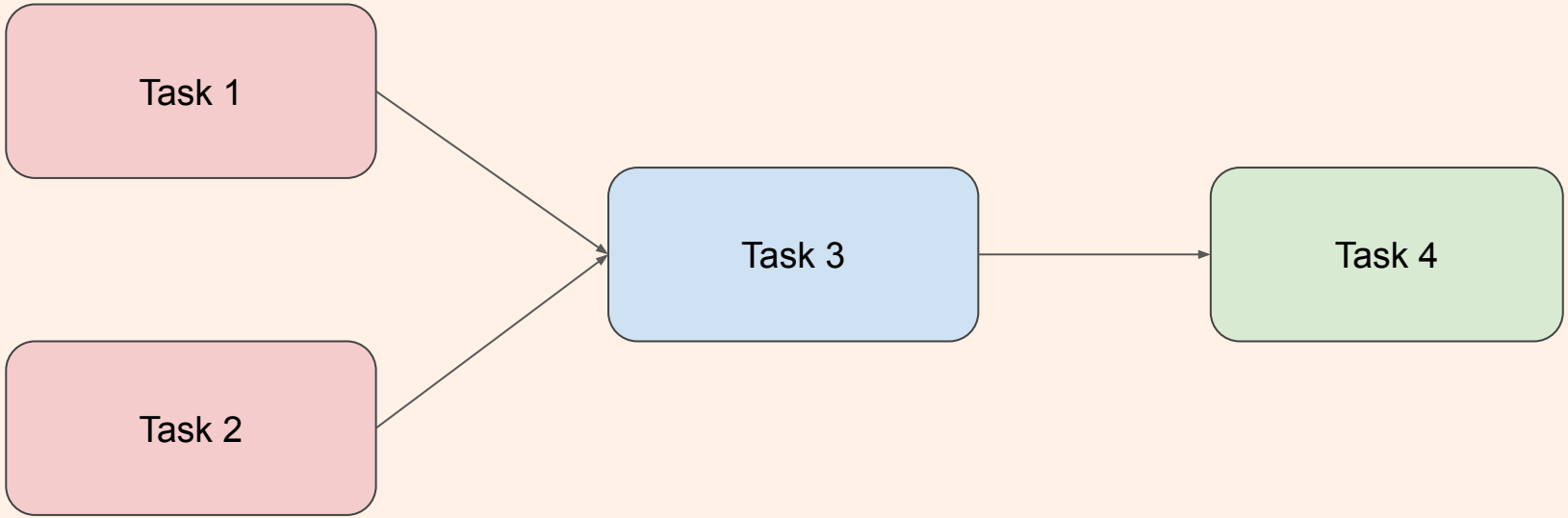

Extract
DATA SOURCE 1
DATA SOURCE 2
Transform
AGGREGATIONS
Load
DATA DESTINATION

Load


Extract
DATA SOURCE 1
DATA SOURCE 2
Transform
AGGREGATIONS
Load
DATA DESTINATION

Transform?

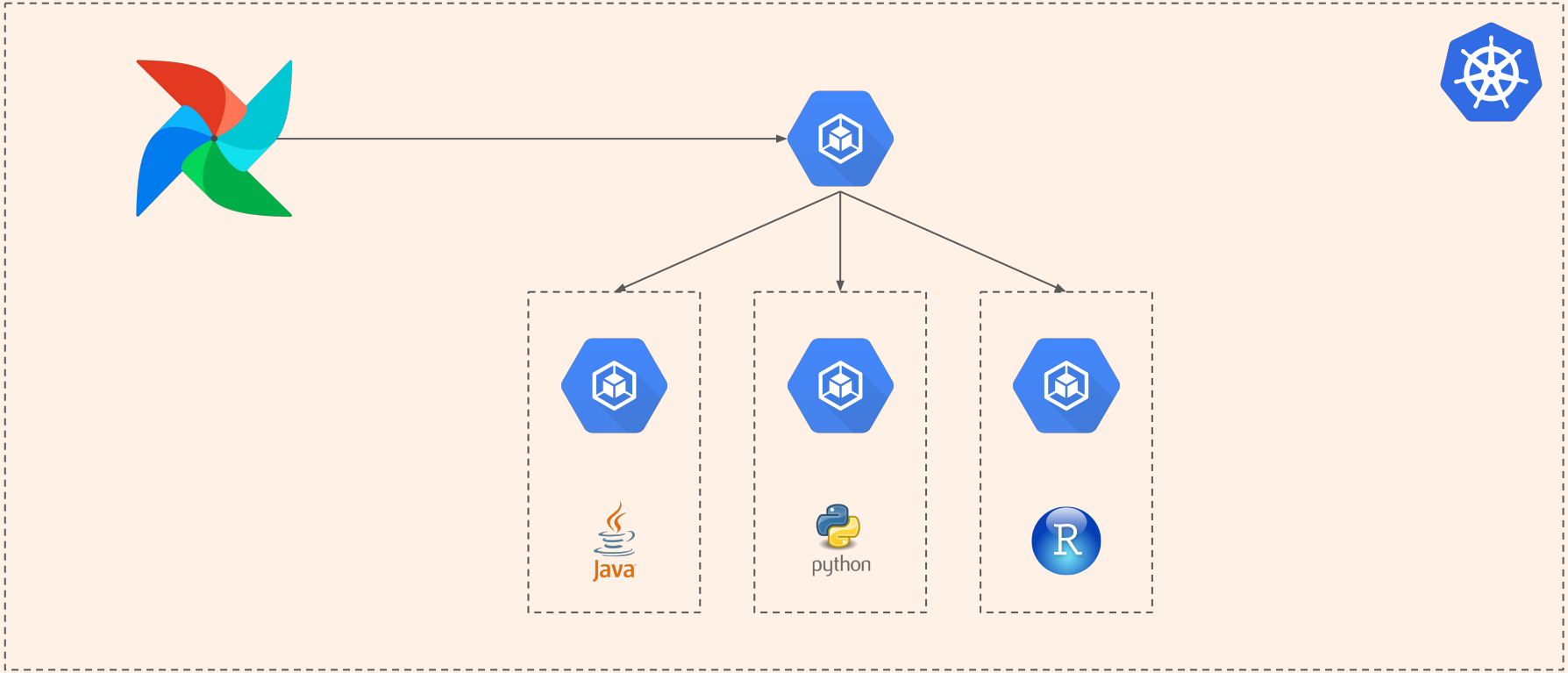
Example workflow



Our goals

- Language agnostic jobs
- Cross task data access

KubernetesPodOperator



Our goals

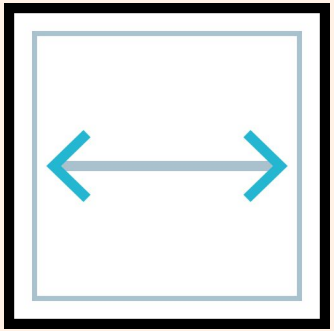
- Language agnostic jobs
- Cross task data access

Unique storage pattern

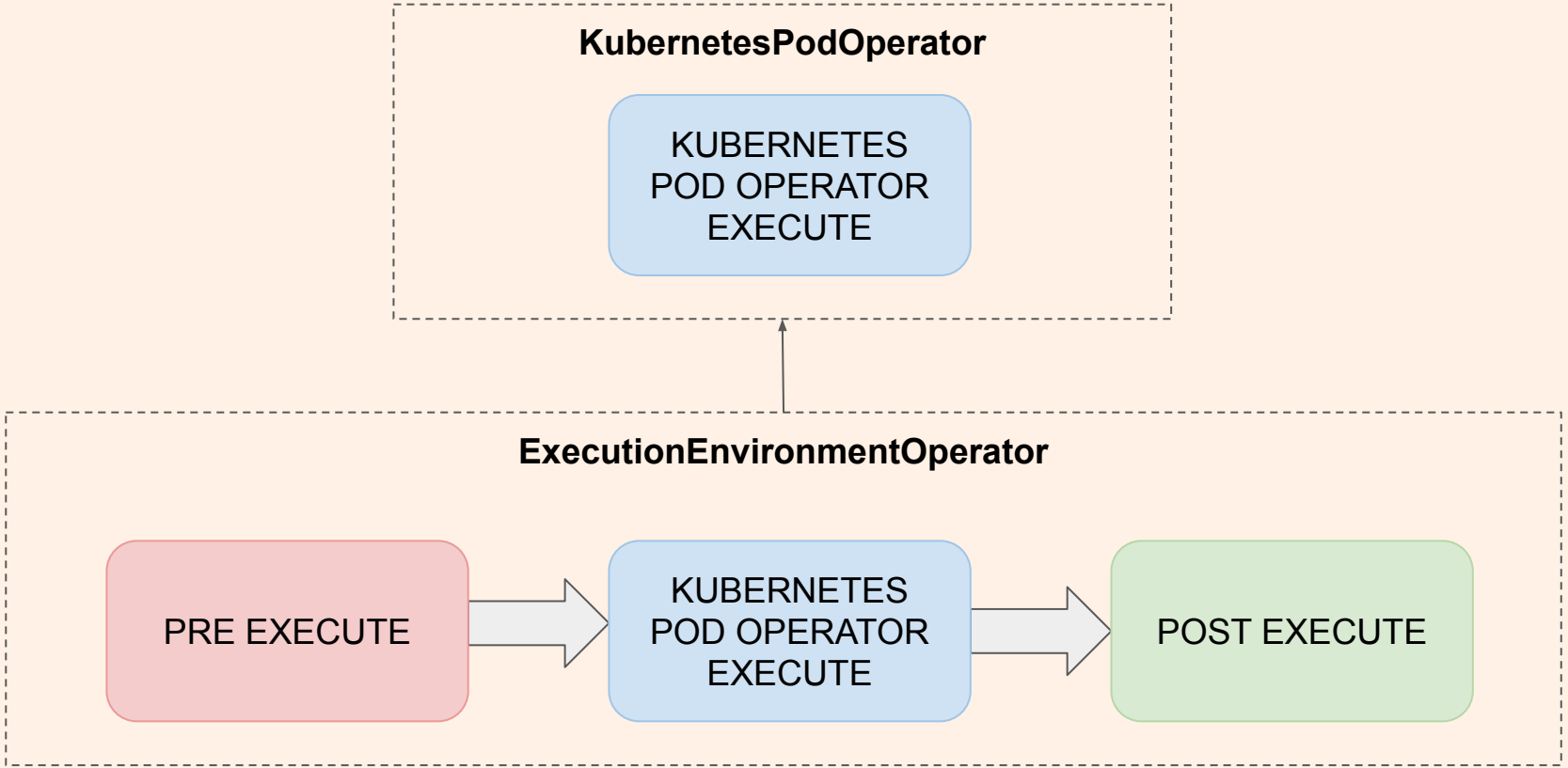
- Unique team name from the multitenancy
- Unique DAG id
- Unique task id per DAG
- Unique execution date per DAG run

/{team}/{dag_id}/{task_id}/{execution_date}

The power of extensibility



ExecutionEnvironmentOperator



Configurable cross task data dependencies

Example input configuration

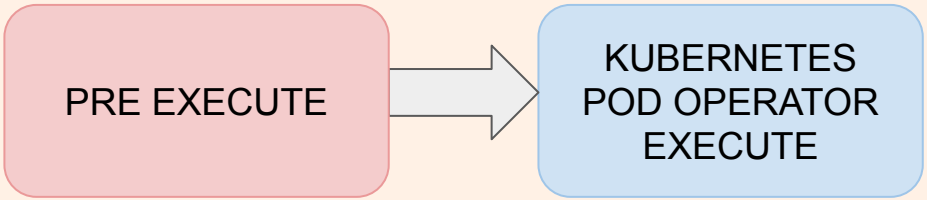
```
example_execution_environment_config = {
    "input": {
        "operators": [
            {
                "task_id": "task_1"
            },
            {
                "task_id": "task_2"
            }
        ]
    }
}

example_execution_environment_operator = ExecutionEnvironmentOperator(
    task_id='task_3',
    job_config=example_execution_environment_config,
    image='example_docker_image',
    tag='latest'
)
```

Example output configuration

```
example_execution_environment_config = {
    "output": {
        "default": {
            "destinations": [
                {
                    "type": "s3",
                    "data": {
                        "bucket": "<s3_bucket>",
                        "path": "<s3_path>",
                        "aws_conn_id": "<s3_connection_for_upload>"
                    }
                }
            ]
        }
    }
}

example_execution_environment_operator = ExecutionEnvironmentOperator(
    task_id='task_3',
    job_config=example_execution_environment_config,
    image='example_docker_image',
    tag='latest'
)
```

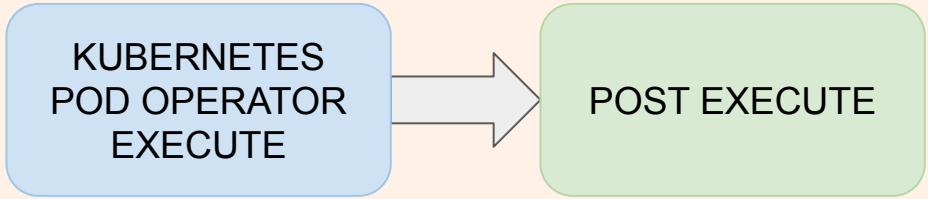
Pre-execute

- Bootstrap the environment
- Enrich the configuration
- Export the configuration to the execution environment pod

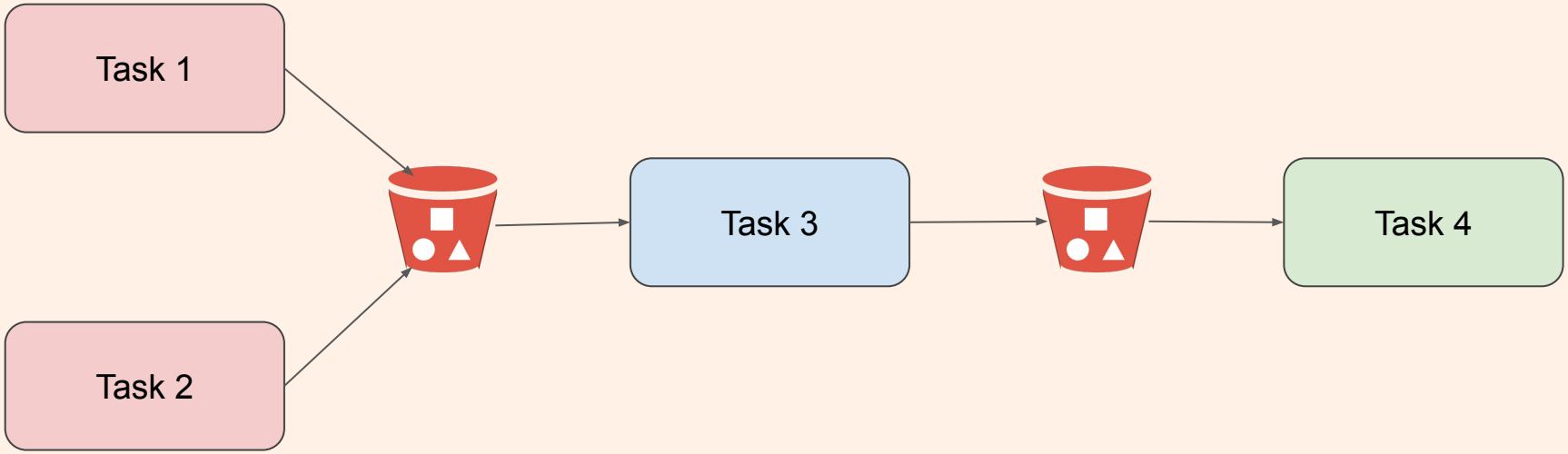


Post-execute

- Handle the execution
- Clear all bootstraps
- Deal with the output



POC with AWS S3 as intermediate storage



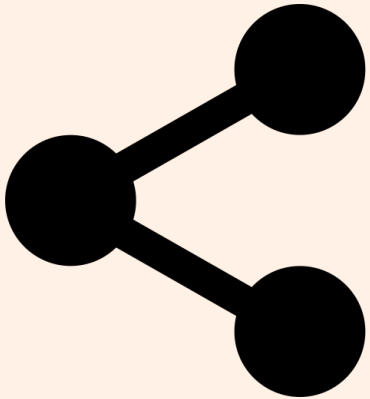
Is this efficient?

- ✗ Multiple downloads and uploads
- ✗ Single processing power
- ✗ Always loading the data in memory

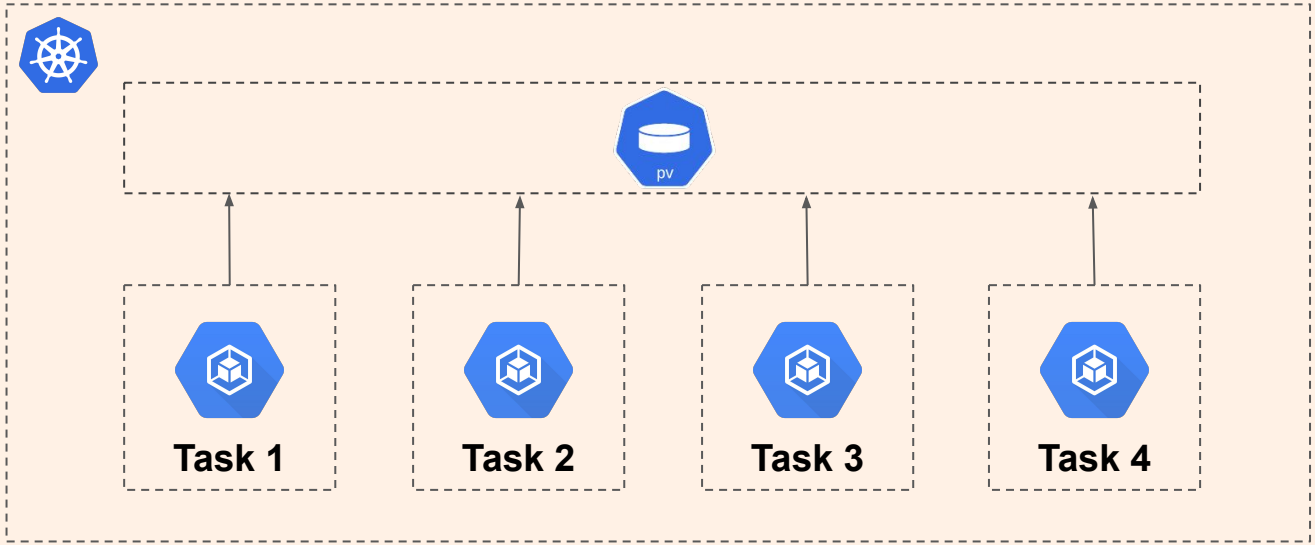
How to evolve the execution environment?

- Remove unnecessary data transfers
- Parallelize the processing
- Provide hot data access

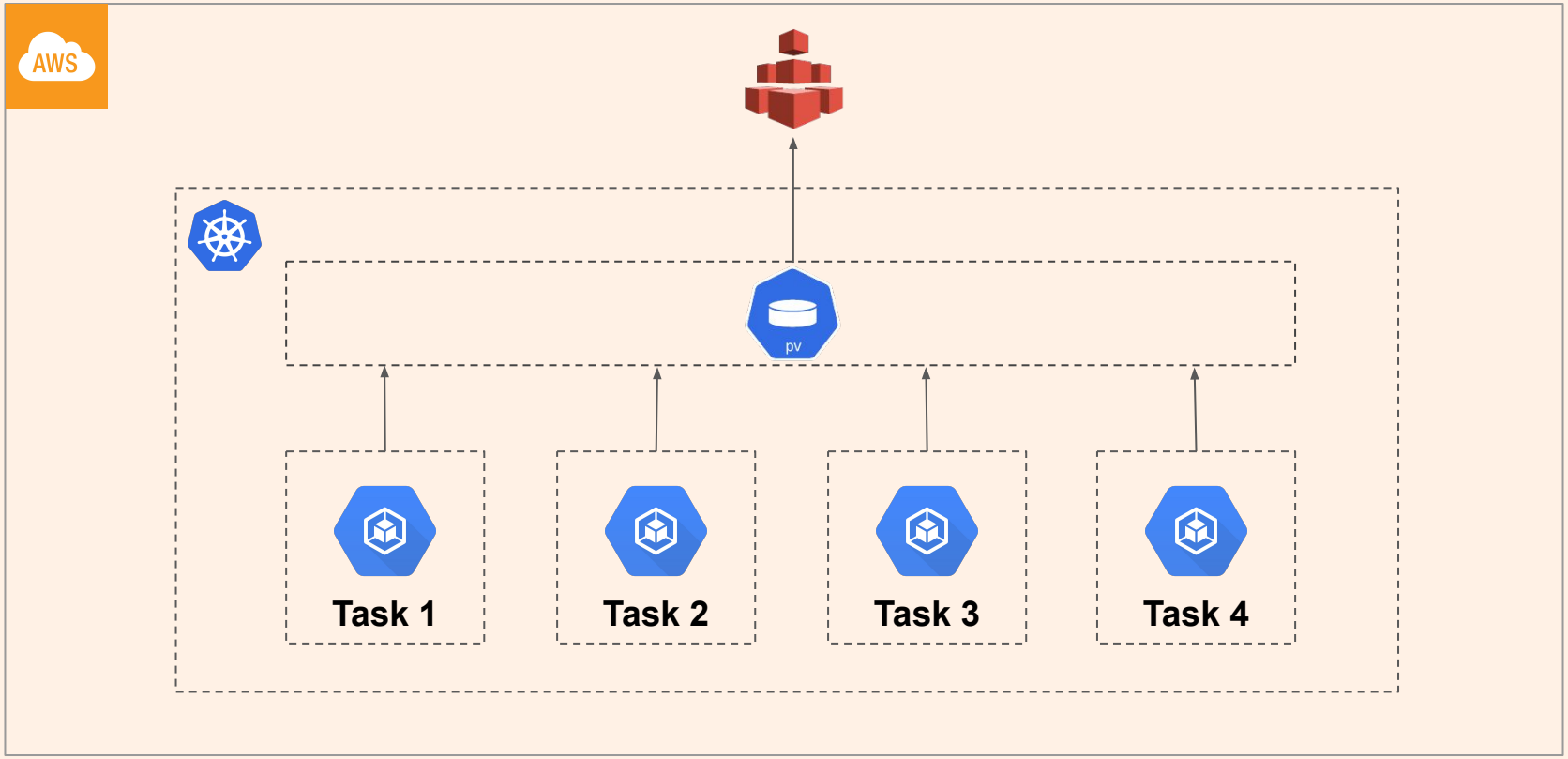
Shared file system



Kubernetes persistent volume



Kubernetes persistent volume with EFS



So far so good

- Remove unnecessary data transfers
- Parallelize the processing
- Provide hot data access

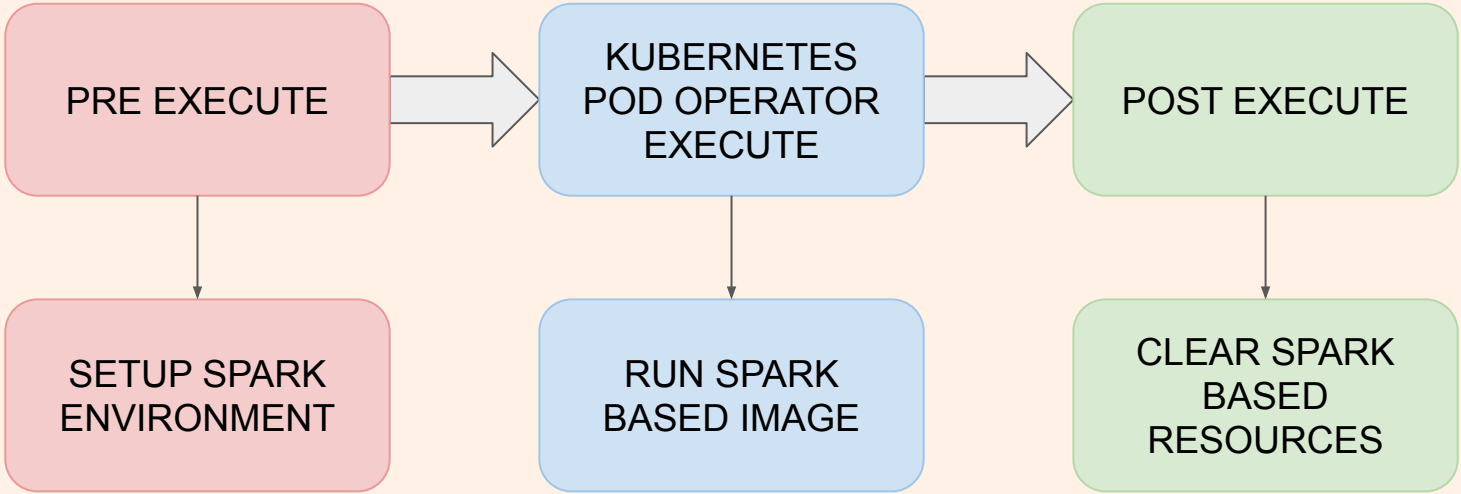
One worker?

APACHE
Spark
TM

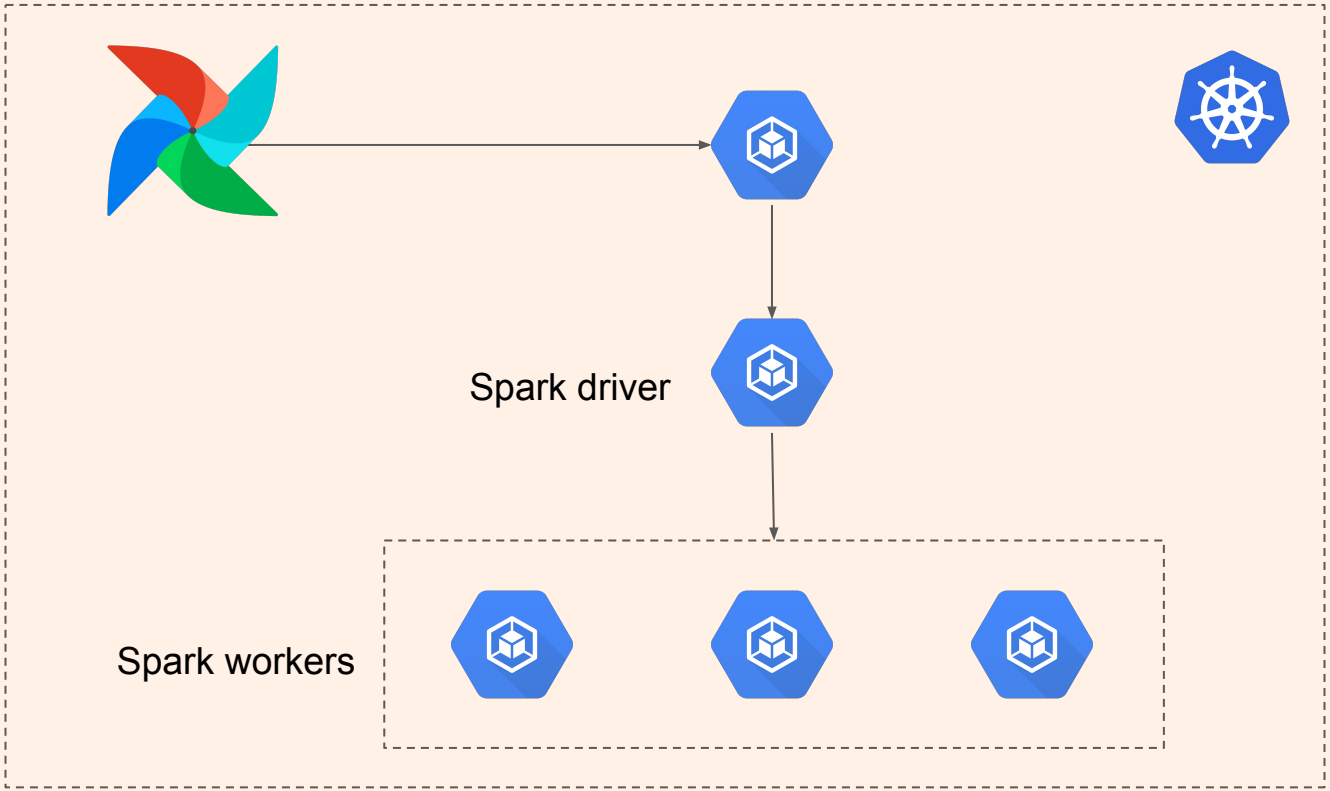
Benefits from Spark

- Runs perfectly in Kubernetes
- Supports many distributed storages
- Allows faster data processing
- Supports multiple languages
- Easy to use

SparkExecutionEnvironmentOperator



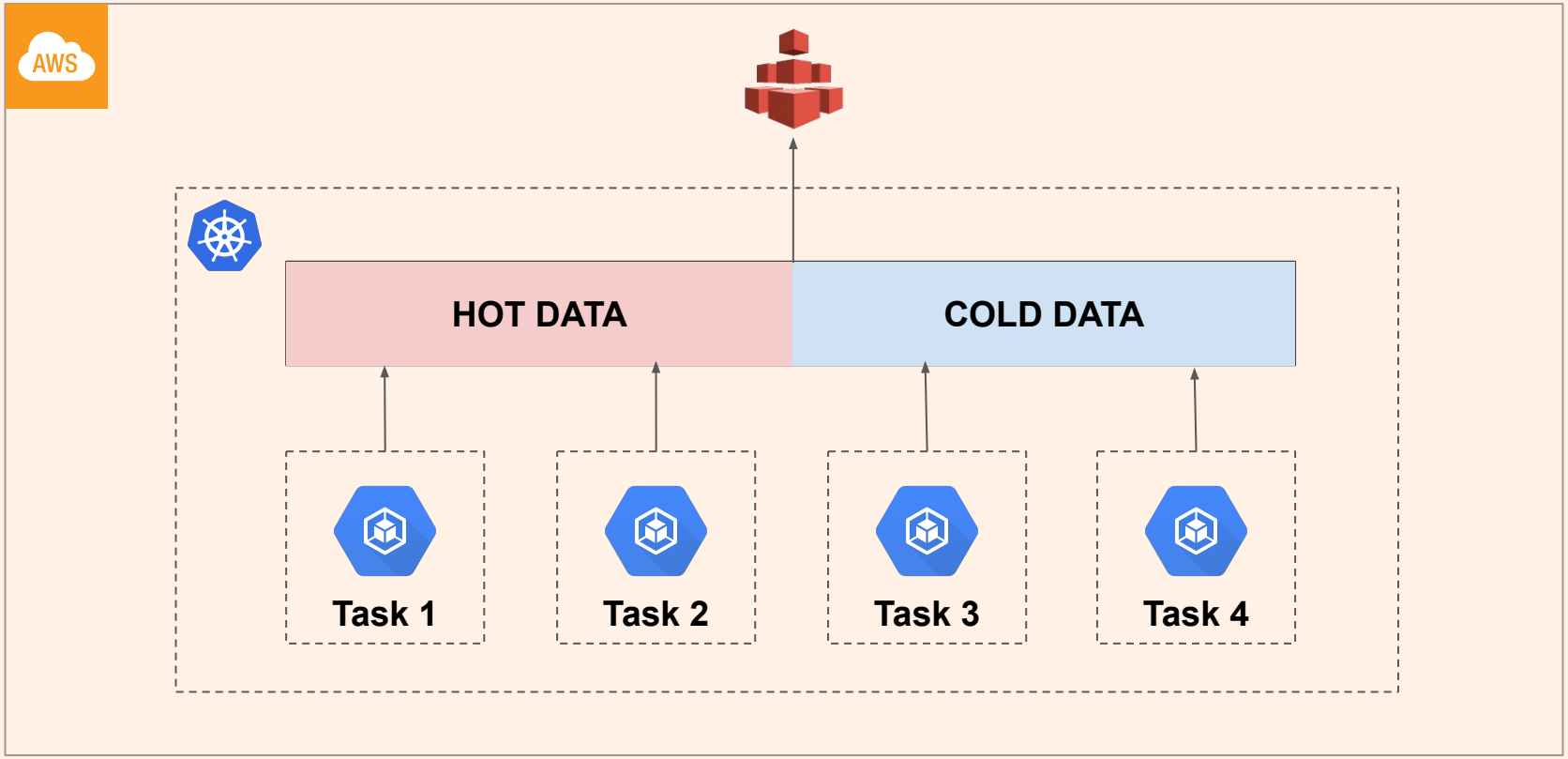
Spark execution environment



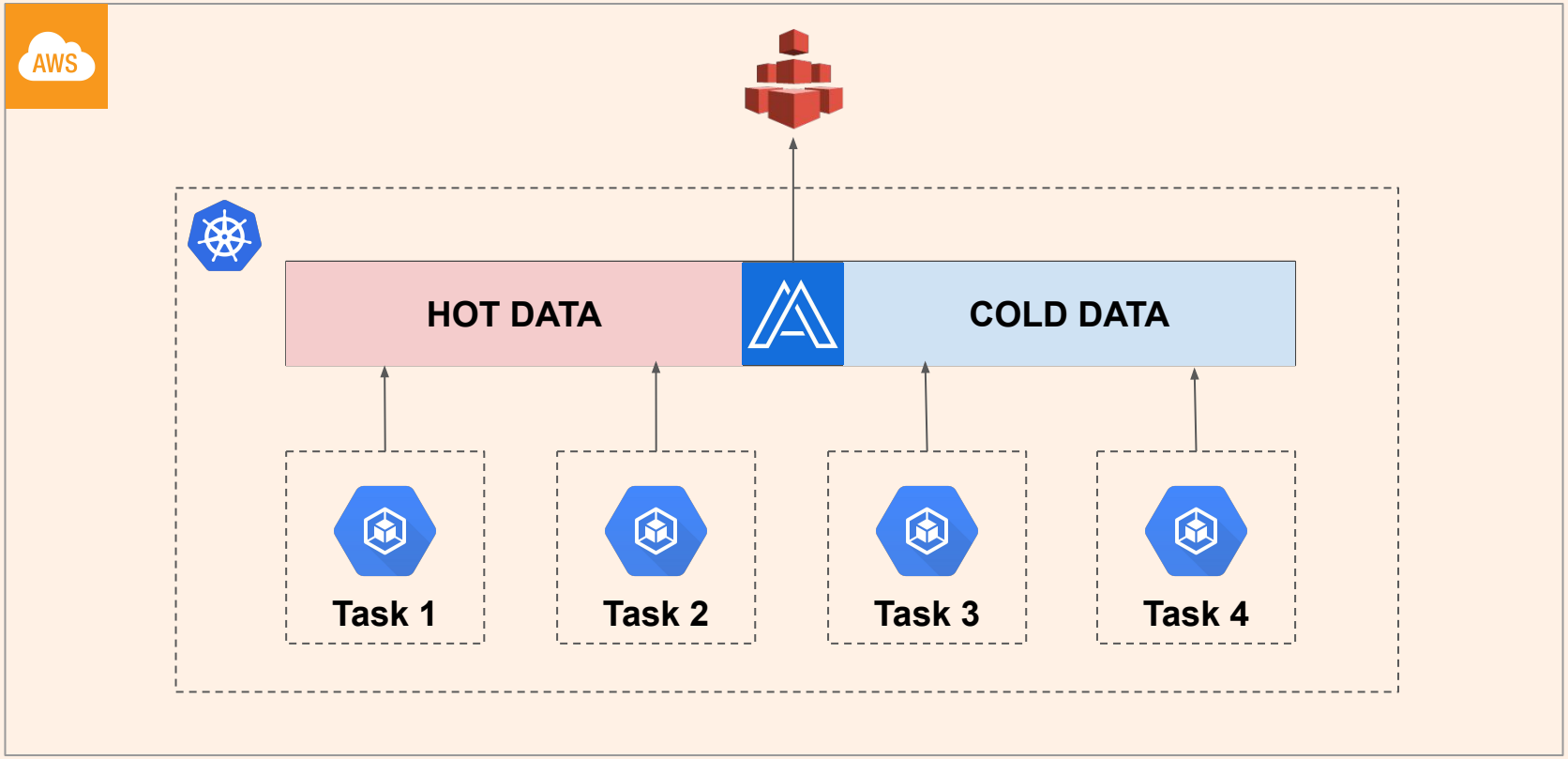
Our current state

- [x] Remove unnecessary data transfers
- [x] Parallelize the processing
- [ ] Provide hot data access

Hot & cold data



Alluxio



Thank you!

#apacheairflow