Advanced DAG Configuration

Dynamic Task Generation

Create dynamic Airflow tasks

- Available in Airflow 2.3+
- Can write DAGs that dynamically generate parallel tasks at runtime

Dynamic task concepts

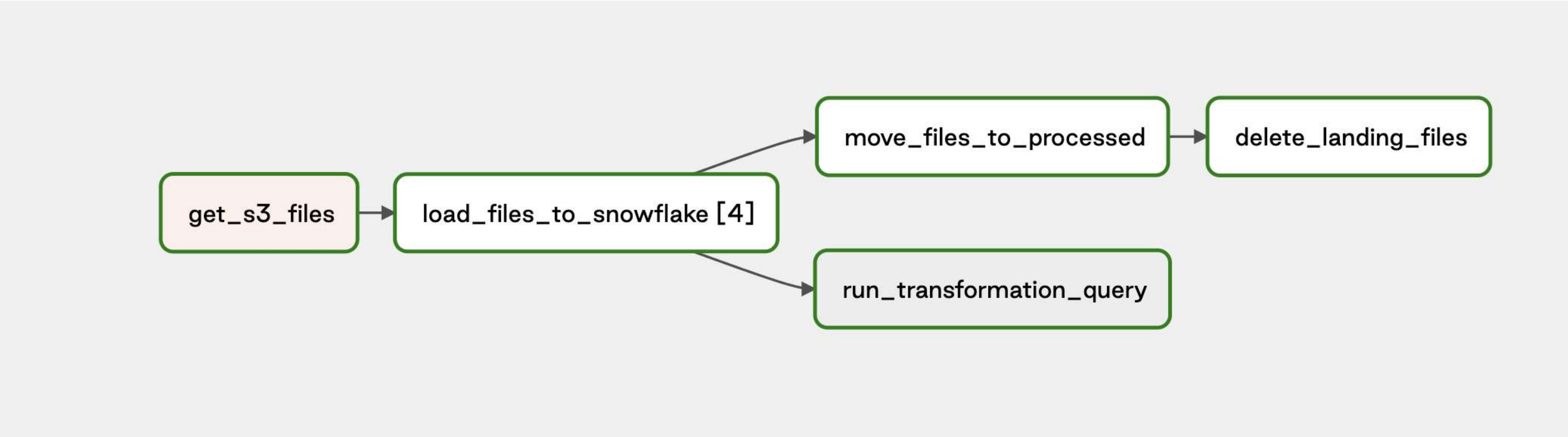
- Airflow tasks have two new functions available
  - expand():
    - Passes the parameters that you want to map
    - A separate parallel task is created for each input
  - partial():
    - Passes any parameters that remain constant across all mapped tasks which are generated by expand()

Dynamic task concepts

```python
@task
def add(x: int, y: int):
    return x + y

added_values = add.partial(y=10).expand(x=[1, 2, 3])
```

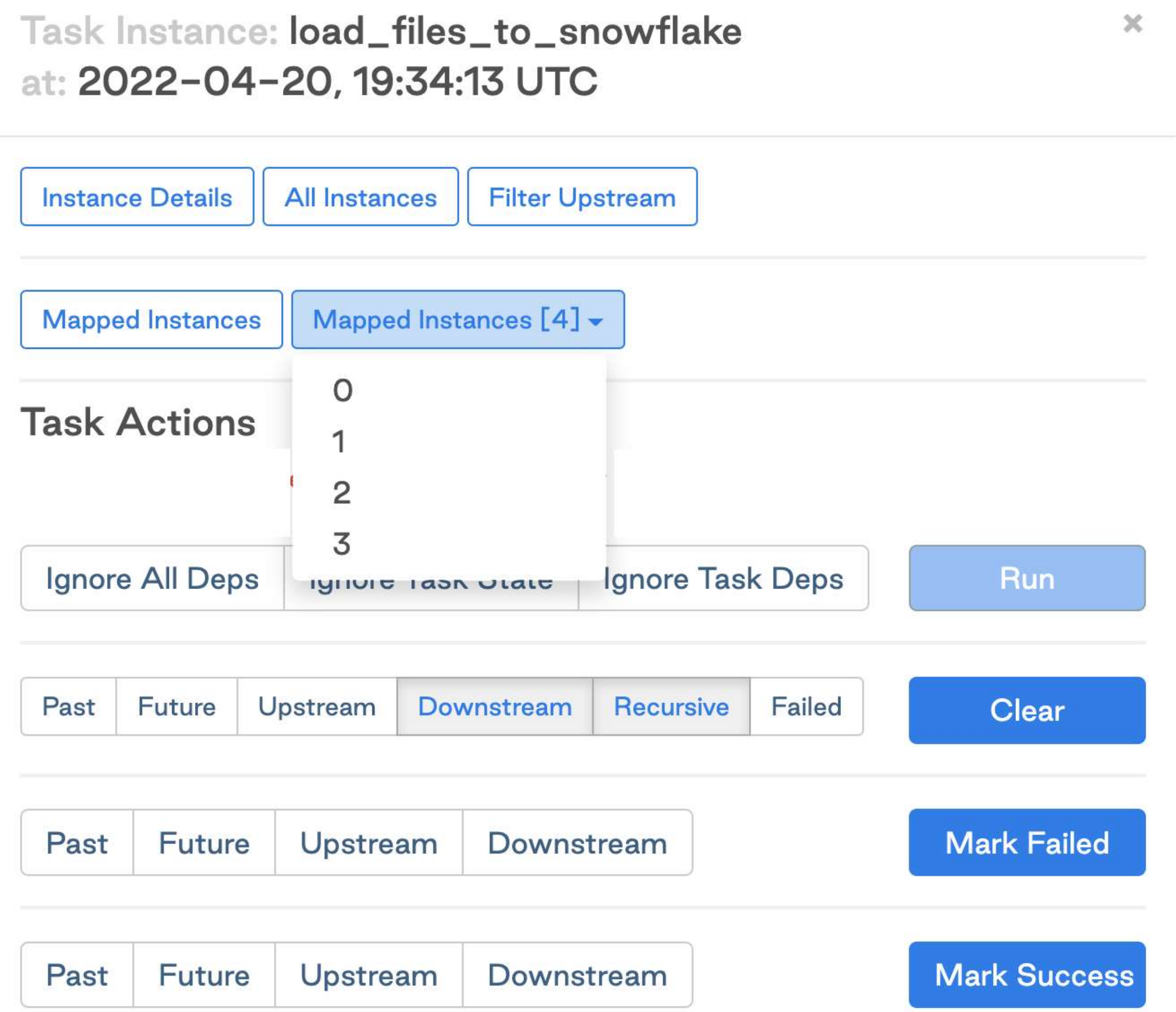
```python
def add_function(x: int, y: int):
    return x + y

added_values = PythonOperator.partial(
    task_id="add",
    python_callable=add_function,
    op_kwargs={"y": 10}
).expand(op_args=[[1],[2],[3]])

added_values
```

Dynamic task concepts



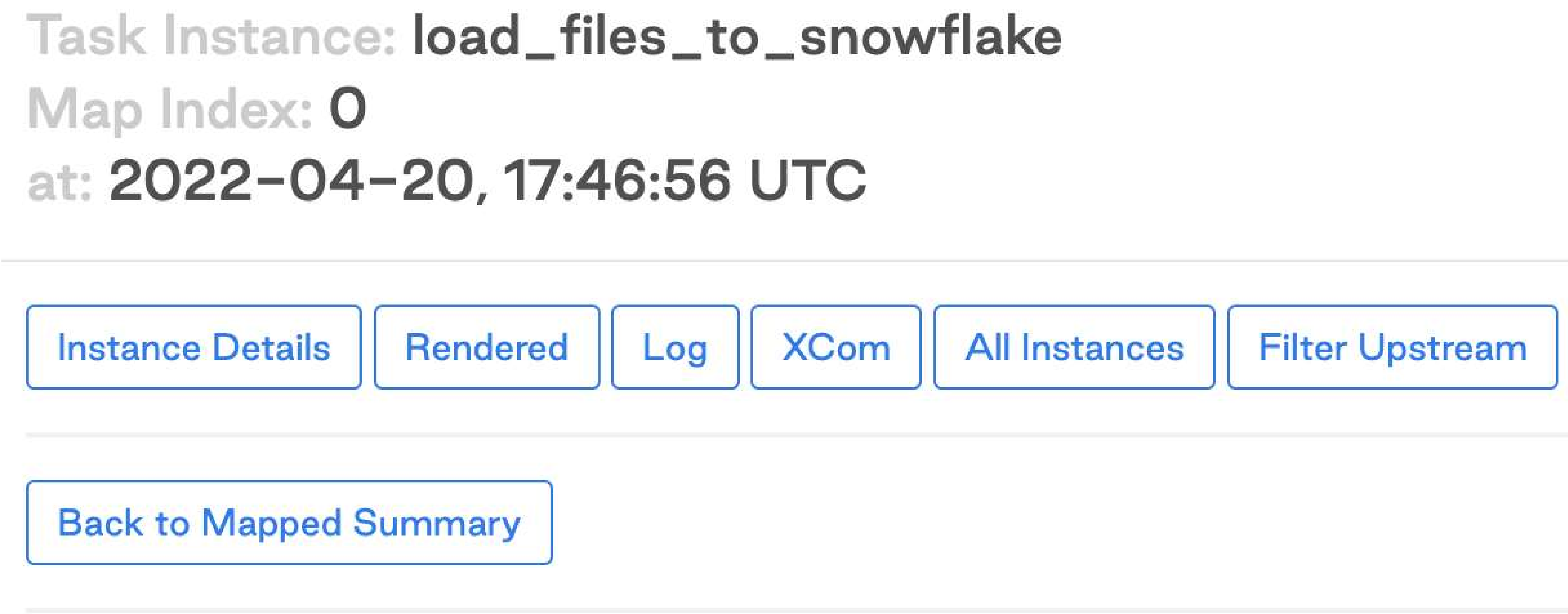
Dynamic task concepts

- Click the mapped task to display the Mapped Instances list
- Select a specific mapped task run to perform actions on.

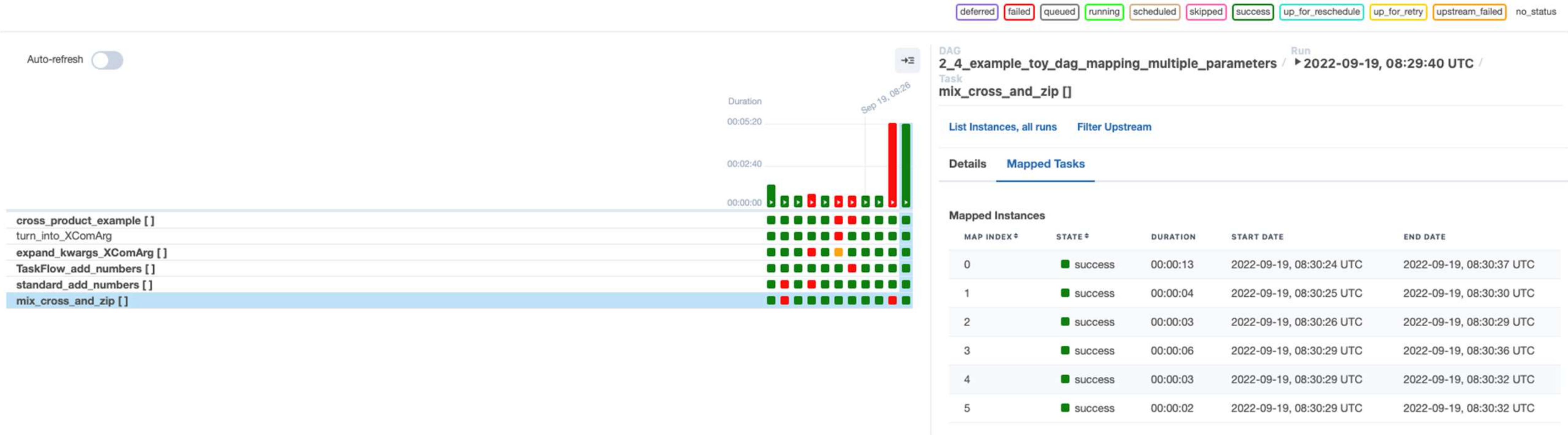


Dynamic task concepts

- Select one of the mapped instances to access links to other views such as Instance Details, Rendered, Log, XCom, and so on.



Grid View


deferred
failed
queued
running
scheduled
skipped
success
up_for_reschedule
up_for_retry
upstream_failed
no_status
Auto-refresh
Duration
00:05:20
00:02:40
00:00:00
Sep 19, 08:26
cross_product_example [ ]
turn_into_XComArg
expand_kwargs_XComArg [ ]
TaskFlow_add_numbers [ ]
standard_add_numbers [ ]
mix_cross_and_zip [ ]
DAG
2_4_example_toy_dag_mapping_multiple_parameters
Run
2022-09-19, 08:29:40 UTC
Task
mix_cross_and_zip []
List Instances, all runs
Filter Upstream
Details
Mapped Tasks
Mapped Instances
MAP INDEX
STATE
DURATION
START DATE
END DATE
0 success 00:00:13 2022-09-19, 08:30:24 UTC 2022-09-19, 08:30:37 UTC
1 success 00:00:04 2022-09-19, 08:30:25 UTC 2022-09-19, 08:30:30 UTC
2 success 00:00:03 2022-09-19, 08:30:26 UTC 2022-09-19, 08:30:29 UTC
3 success 00:00:06 2022-09-19, 08:30:29 UTC 2022-09-19, 08:30:36 UTC
4 success 00:00:03 2022-09-19, 08:30:29 UTC 2022-09-19, 08:30:32 UTC
5 success 00:00:02 2022-09-19, 08:30:29 UTC 2022-09-19, 08:30:32 UTC

Mapping over the result of another operator

- You can use the output of an upstream operator as the input data for a dynamically mapped downstream task.

```
@task
def one_two_three_TF():
    return [1, 2, 3]

@task
def plus_10_TF(x):
    return x + 10

plus_10_TF.partial().expand(x=one_two_three_TF())
```

Mapping over multiple parameters

- Cross-product
- Sets of keyword arguments

Cross-product

```python
cross_product_example = BashOperator.partial(
    task_id="cross_product_example"
).expand(
    bash_command=[
        "echo $WORD", # prints the env variable WORD
        "echo `expr length $WORD`", # prints the number of letters in WORD
        "echo ${WORD//e/X}" # replaces each "e" in WORD with "X"
    ],
    env=[
        {"WORD": "hello"},
        {"WORD": "tea"},
        {"WORD": "goodbye"}
    ]
)
```

Sets of keyword arguments

```
# input sets of kwargs provided directly as a list[dict]
t1 = BashOperator.partial(task_id="t1").expand_kwargs(
    [
        {"bash_command": "echo $WORD", "env" : {"WORD": "hello"}},
        {"bash_command": "echo `expr length $WORD`", "env" : {"WORD": "tea"}},
        {"bash_command": "echo ${WORD//e/X}", "env" : {"WORD": "goodbye"}}
    ]
)
```

Repeated mapping

- You can dynamically map an Airflow task over the output of another dynamically mapped task.

```
@task
def multiply_by_2(num):
    return num * 2

@task
def add_10(num):
    return num + 10

@task
def multiply_by_100(num):
    return num * 100

multiplied_value_1 = multiply_by_2.expand(num=[1, 2, 3])
summed_value = add_10.expand(num=multiplied_value_1)
multiply_by_100.expand(num=summed_value)
```

Mapping over task groups

```python
# creating a task group using the decorator with the dynamic input my_num
@task_group(group_id="group1")
def tg1(my_num):
    @task
    def print_num(num):
        return num

    @task
    def add_42(num):
        return num + 42

    print_num(my_num) >> add_42(my_num)

# creating 6 mapped task group instances of the task group group1
tg1_object = tg1.expand(my_num=[19, 23, 42, 8, 7, 108])
```

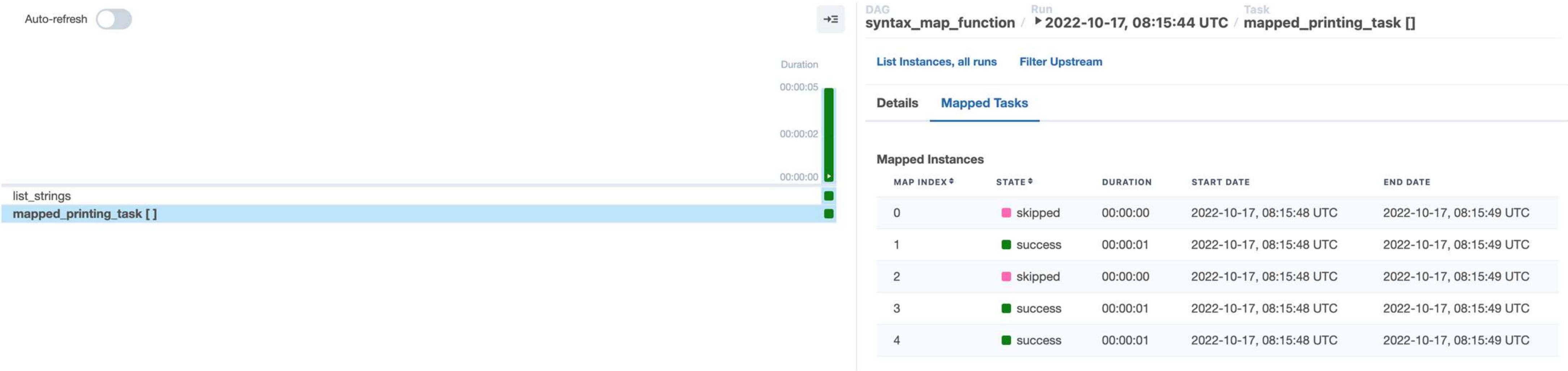
Transform outputs with .map

- There are use cases where you want to transform the output of an upstream task before another task dynamically maps over it

```python
# an upstream task returns a list of outputs in a fixed format
@task
def list_strings():
    return ["skip_hello", "hi", "skip_hallo", "hola", "hey"]

# the function used to transform the upstream output before
# a downstream task is dynamically mapped over it
def skip_strings_starting_with_skip(string):
    if len(string) < 4:
        return string + "!"
    elif string[:4] == "skip":
        raise AirflowSkipException(f"Skipping {string}; as I was told!")
    else:
        return string + "!"

# transforming the output of the first task with the map function.
transformed_list = list_strings().map(skip_strings_starting_with_skip)

# the task using dynamic task mapping on the transformed list of strings
@task
def mapped_printing_task(string):
    return "Say " + string

mapped_printing_task.partial().expand(string=transformed_list)
```

Transform outputs with .map

- In the grid view you can see how the mapped task instances 0 and 2 have been skipped.



Deferrable Operators & Triggers

Why not Sensors?

- Standard Operators and Sensors take up a full worker slot for the entire time they are running,
  - Even if they are idle;
- for example, if you only have 100 worker slots available to run Tasks, and you have 100 DAGs waiting on a Sensor that’s currently running but idle, then you cannot run anything else - even though your entire Airflow cluster is essentially idle
- This is where Deferrable Operators come in.

Deferrable Operators

- Ability to suspend itself and free up the worker when it knows it has to wait
- Hand off the job of resuming it to something called a Trigger
- As a result, while it is suspended (deferred), it is not taking up a worker slot
- Triggers are small, asynchronous pieces of Python code

Using Deferrable Operators

- Two steps:
  - Ensure your Airflow installation is running at least one triggerer process, as well as the normal scheduler
  - Use deferrable operators/sensors in your DAGs

Writing Deferrable Operators

- Refer to Hands-on

Triggering Deferral

- Refer to Hands-on

Writing Triggers

- A Trigger is written as a class that inherits from BaseTrigger
- Implements three methods
    - __init__
        - to receive arguments from Operators instantiating it
    - Run
        - an asynchronous method that runs its logic and yields one or more TriggerEvent instances as an asynchronous generator
    - Serialize
        - which returns the information needed to re-construct this trigger, as a tuple of the classpath, and keyword arguments to pass to __init__

View triggerer logs

- The triggerer generates logs that are available together with logs of other components.

Monitor triggerer

- In addition to monitoring the triggerer, you can check the number of deferred tasks in the Unfinished Task metrics on the Monitoring dashboard of your environment.

Triggerer metrics

Note: Some metrics related to triggerer are provided through Airflow metrics.

Name	API	Description
Active triggerers	`composer.googleapis.com/environment/active_triggerers`	Number of active triggerer instances.

XCom

What is an Airflow XCom?

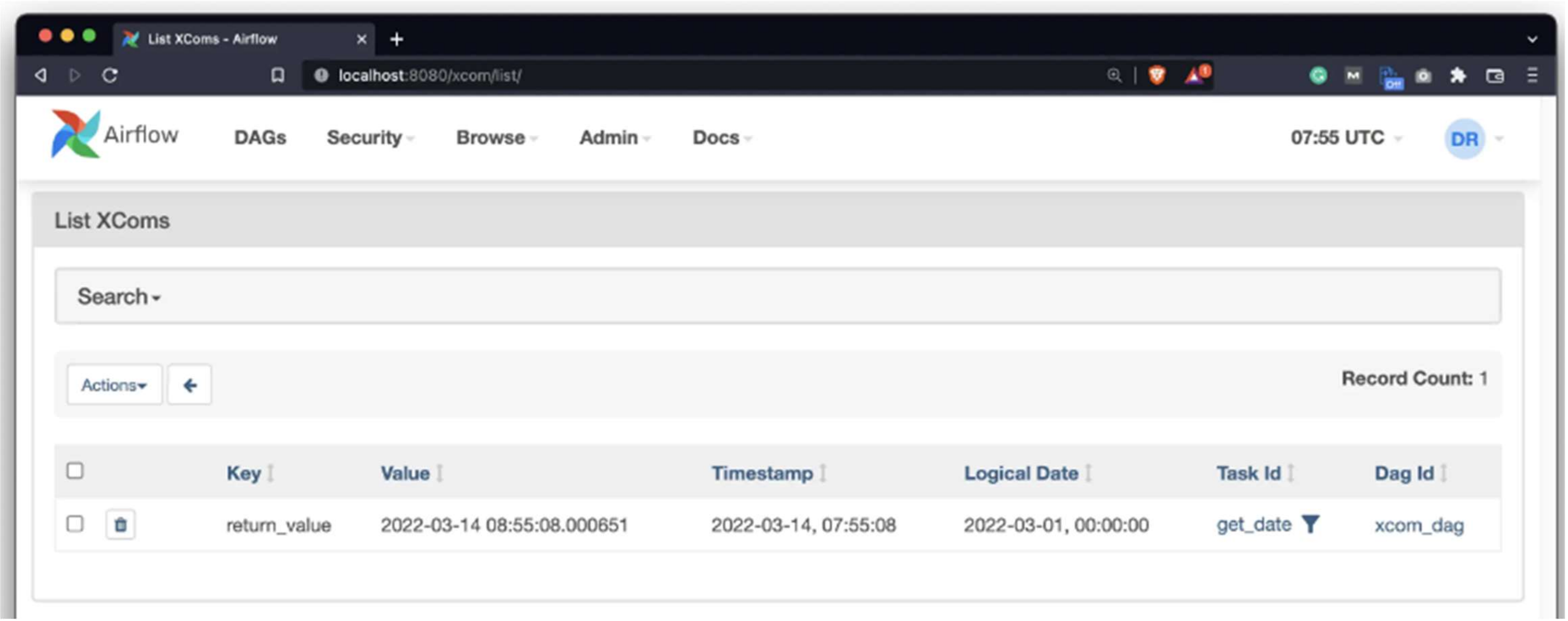
- Mechanism that let Tasks talk to each other
- By default Tasks are entirely isolated
- XComs are explicitly “pushed” and “pulled”

How to use XCom in Airflow

How to Push a Value to Airflow Xcoms?

```
from datetime import datetime
from airflow.models import DAG
from airflow.operators.python import PythonOperator


def get_date() -> str:
    return str(datetime.now())


with DAG(
    dag_id='xcom_dag',
    schedule_interval='@daily',
    start_date=datetime(2022, 3, 1),
    catchup=False
) as dag:

    task_get_date = PythonOperator(
        task_id='get_date',
        python_callable=get_date,
        do_xcom_push=True
    )
```

How to Push a Value to Airflow Xcoms?



How to Get the XCom Value through Airflow

```python
from datetime import datetime
from airflow.models import DAG
from airflow.operators.python import PythonOperator


def get_date() -> str:
    ...


def save_date(ti) -> None:
    dt = ti.xcom_pull(task_ids=['get_date'])
    if not dt:
        raise ValueError('No value currently stored in XComs.')

    with open('/Users/dradecic/airflow/data/date.txt', 'w') as f:
        f.write(dt[0])


with DAG(
    ...
) as dag:

    task_get_date = PythonOperator(
        ...
    )

    task_save_date = PythonOperator(
        task_id='save_date',
        python_callable=save_date
    )
```

XCom limitations

- Avoid sending huge Pandas DataFrames between tasks
- You’re likely to run into memory issues if you try to exchange large datasets between the tasks
- Process big datasets in Spark, and use Airflow only to trigger a Spark job

Thanks