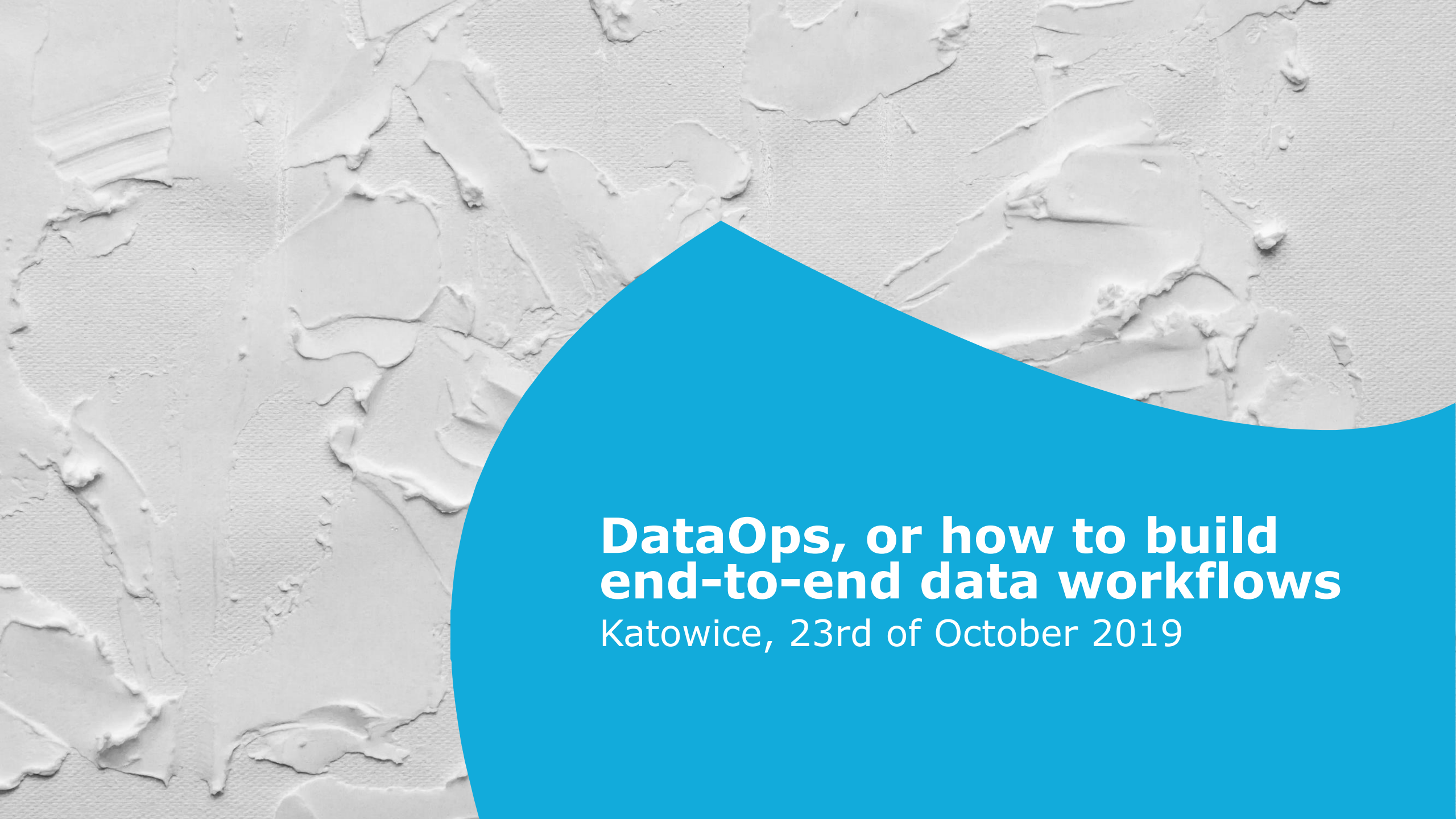




DataOps, or how to build end-to-end data workflows

Katowice, 23rd of October 2019

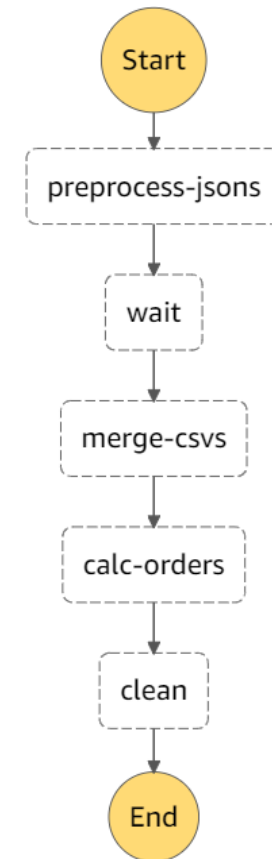
Relationships



Expectation



Reality



DataOps



DevOps improves quality, velocity and scaling of software development.

Borrowing methods from DevOps, **DataOps** offers the same improvements to production Data Science.

Best practices



12 Best Practices

- Alerting
- CI/CD
- Code review
- Integration testing
- Logging
- Metrics collection
- Monitoring
- Quality validation
- Reproducibility
- Scheduling
- Test regressions
- Unit testing



Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.



12 Best Practices

- Alerting
- CI/CD
- Code review
- Integration testing
- Logging
- Metrics collection
- Monitoring
- Quality validation
- Reproducibility
- Scheduling
- Test regressions
- Unit testing



MVP



- Alerting
- **CI/CD**
- Code review
- Integration testing
- Logging
- Metrics collection
- Monitoring
- **Quality validation**
- **Reproducibility**
- Scheduling
- Test regressions
- Unit testing

How to start



How to start

- Share data science code to private git repository.
- Use versioning of data science code. Use requirements.txt and add version to scripts.
- Add owner to scripts.
- Write readable Markdown documentation.
- Verify code quality with [unittest](#), [flake8](#), and regular code reviews. Automate tests.



Workflow manager



Scripts with Cron

- No reproducibility.
- No logs, no monitoring, no alerts.
- Code and logic duplication.
- Maintenance time grows exponentially.



Workflow managers

- Pinball 04-03-2015
- Airflow 06-10-2014
- Mistral 11-10-2013
- Azkaban 03-05-2012
- Luigi 17-11-2011
- Oozie 19-08-2011



Workflow managers

- Pinball 04-03-2015 (Pinterest)
- **Airflow** 06-10-2014 (Airbnb -> Apache)
- Mistral 11-10-2013 (OpenStack)
- Azkaban 03-05-2012 (LinkedIn)
- **Luigi** 17-11-2011 (Spotify)
- Oozie 19-08-2011 (Yahoo -> Apache)

Airflow



Airflow. Import

```
import datetime
```

```
from airflow import DAG
```

```
from airflow.operators.bash_operator import BashOperator
```

```
from airflow.operators.python_operator import PythonOperator
```



Airflow. Task #2

```
def task2():  
    print('task2')
```



Airflow. Default

```
default_args = {  
    'owner': 'Ruslan Korniiichuk',  
    'start_date': datetime.datetime(2019, 10, 22)  
}
```

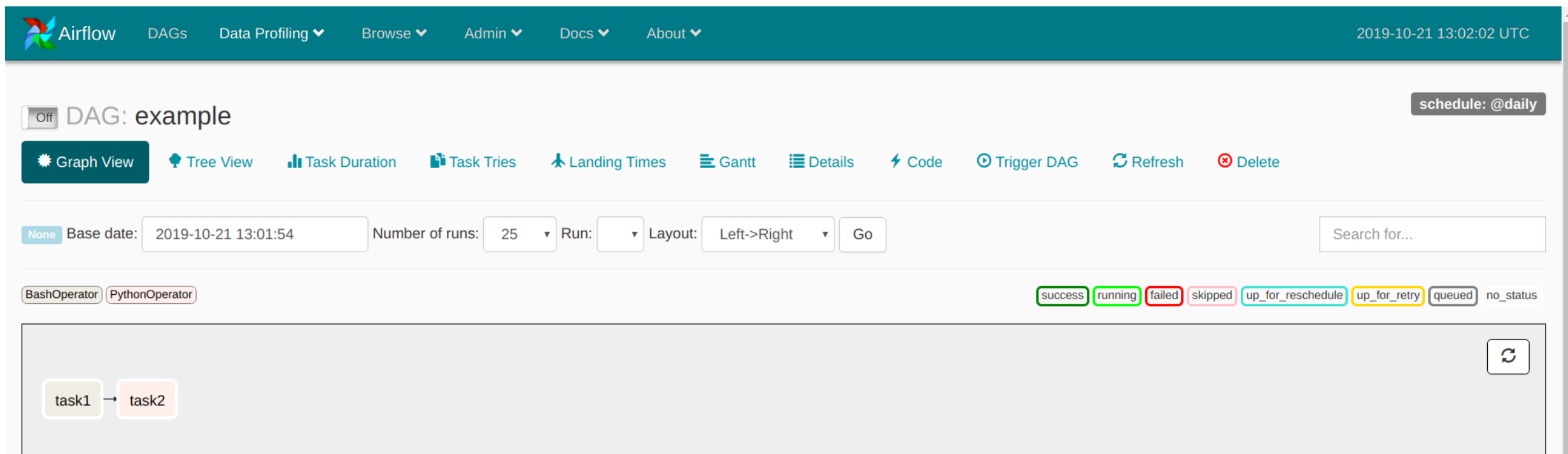
Airflow. DAG



```
with DAG('example',
        default_args=default_args,
        schedule_interval='@daily') as dag:

    task1 = BashOperator(task_id='task1',
                        bash_command='echo task1')
    task2 = PythonOperator(task_id='task2',
                          python_callable=task2)
```

DAG - or a Directed Acyclic Graph - is a collection of all the tasks you want to run, organized in a way that reflects their relationships.



The screenshot shows the Apache Airflow web interface. At the top is a teal navigation bar with the Airflow logo and links for DAGs, Data Profiling, Browse, Admin, Docs, and About. The current date and time are 2019-10-21 13:02:02 UTC.

The main content area displays a DAG named "example". A toggle switch is set to "Off". A "schedule: @daily" button is visible. Below the DAG name is a row of view options: Graph View (selected), Tree View, Task Duration, Task Tries, Landing Times, Gantt, Details, Code, Trigger DAG, Refresh, and Delete.

Below the view options is a filter section with a "None" button, a "Base date:" field (2019-10-21 13:01:54), a "Number of runs:" dropdown (25), a "Run:" dropdown, a "Layout:" dropdown (Left->Right), and a "Go" button. A search bar labeled "Search for..." is also present.

Below the filter section is a row of operator buttons: BashOperator and PythonOperator. To the right is a row of status buttons: success, running, failed, skipped, up_for_reschedule, up_for_retry, queued, and no_status.

The main area shows a simple DAG graph with two tasks: task1 and task2, connected by an arrow. A refresh button is in the bottom right corner.

Airflow. DAG



```
with DAG('example',
         default_args=default_args,
         schedule_interval='@daily') as dag:

    task1 = BashOperator(task_id='task1',
                          bash_command='echo task1')
    task2 = PythonOperator(task_id='task2',
                           python_callable=task2)
```



Airflow. Relationships

```
task1 >> task2
```



Airflow. All-in-one

```
import datetime

from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator

def task2():
    print('task2')

default_args = {
    'owner': 'Ruslan Korniiichuk',
    'start_date': datetime.datetime(2019, 10, 22)
}

with DAG('example',
        default_args=default_args,
        schedule_interval='@daily') as dag:

    task1 = BashOperator(task_id='task1', bash_command='echo task1')
    task2 = PythonOperator(task_id='task2', python_callable=task2)

task1 >> task2
```

Airflow. Graph view



Airflow

DAGs

Data Profiling ▾

Browse ▾

Admin ▾

Docs ▾

About ▾

2019-10-22 21:20:49 UTC

On

DAG: example

schedule: @daily



Graph View



Tree View



Task Duration



Task Tries



Landing Times



Gantt



Details



Code



Trigger DAG



Refresh



Delete

success

Base date:

2019-10-21 18:03:28

Number of runs:

25 ▾

Run:

manual__2019-10-21T18:03:27.971523+00:00 ▾

Layout:

Left->Right ▾

Go

Search for...

BashOperator

PythonOperator

success

running

failed

skipped

up_for_reschedule

up_for_retry

queued

no_status





Luigi



Luigi. Import

```
import luigi
from luigi.contrib.external_program import ExternalProgramTask
```

Luigi. Tasks



```
class Task1(ExternalProgramTask):

    dst = 'task1'

    def program_args(self):
        return ['touch', self.dst]

    def output(self):
        return luigi.LocalTarget(self.dst)
```

```
class Task2(luigi.Task):

    def requires(self):
        return Task1()

    def run(self):
        dst = self.input().path
        with open(dst, 'a') as f:
            f.write('task2')
```



Luigi. All-in-one

```
import luigi
from luigi.contrib.external_program import ExternalProgramTask

class Task1(ExternalProgramTask):

    dst = 'task1'

    def program_args(self):
        return ['touch', self.dst]

    def output(self):
        return luigi.LocalTarget(self.dst)

class Task2(luigi.Task):

    def requires(self):
        return Task1()

    def run(self):
        dst = self.input().path
        with open(dst, 'a') as f:
            f.write('task2')
```



Luigi. Execution

...

```
if __name__ == '__main__':  
    luigi.run()
```

Luigi. Graph view



Luigi Task Status



Task List

Dependency Graph

Workers

Resources

Running

Task2__99914b932b

Show task details

Show Upstream Dependencies ☐

Visualisation Type

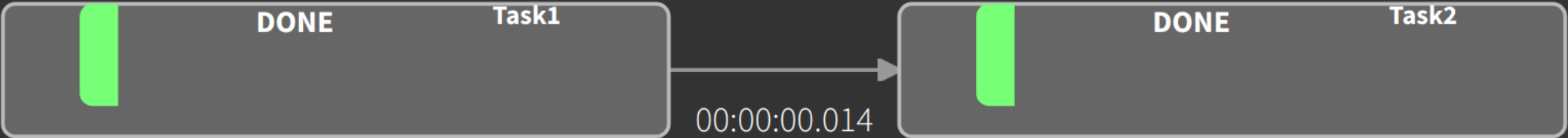
D3

SVG

Hide Done ☐

Task2()

Dependency Graph



Airflow vs Luigi

Airflow vs Luigi



Airflow

pros

- everything in Python (no XML)
- DAGs catalog
- annoying scheduler
- readable DAG concept
- built-in features (e.g. BashOperator)
- readable relationships
- DAG's owner
- open-source

cons

- delay between task execution
- cannot test/run task and all its relationships
- hard maintenance
- cross-communication via XCom

[example](#)

Luigi

pros

- everything in Python (no XML)
- rapid prototyping
- cross-communication
- built-in Target
- built-in features (e.g. ExternalProgramTask)
- easy-to-start
- open-source

cons

- no DAG concept and DAGs catalog
- no scheduling
- maintenance

[example](#)

Airflow vs Luigi



Airflow

pros

- ~~everything in Python (no XML)~~
- DAGs catalog
- annoying scheduler
- readable DAG concept
- ~~built-in features (e.g. BashOperator)~~
- readable relationships
- DAG's owner
- ~~open source~~

cons

- delay between task execution
- cannot test/run task and all its relationships
- ~~hard maintenance~~
- cross-communication via XCom

[example](#)

Luigi

pros

- ~~everything in Python (no XML)~~
- rapid prototyping
- cross-communication
- built-in Target
- ~~built-in features (e.g. ExternalProgramTask)~~
- easy-to-start
- ~~open source~~

cons

- no DAG concept and DAGs catalog
- no scheduling
- ~~maintenance~~

[example](#)

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept
- readable relationships
- DAG's owner

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)

Airflow. DAGs catalog



 Airflow

DAGs

Data Profiling ▾

Browse ▾

Admin ▾

Docs ▾

About ▾

2019-10-21 17:37:28 UTC

DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks 	Last Run 	DAG Runs 	Links
	 Off	example	@daily	Ruslan Korniiichuk				          

Showing 1 to 1 of 1 entries

«

<

1

>

»

[Hide Paused DAGs](#)

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept
- readable relationships
- DAG's owner

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)



Airflow. Relationships

```
task1.set_downstream(task2)
```

or

```
task2.set_upstream(task1)
```



Airflow. Relationships

```
task1.set_downstream(task2)
```

or

```
task2.set_upstream(task1)
```

or

```
dag.set_dependency('task1', 'task2')
```



Airflow. Relationships

```
task1.set_downstream(task2)
```

or

```
task2.set_upstream(task1)
```

or

```
dag.set_dependency('task1', 'task2')
```

or

```
task1 >> task2
```



Luigi. Relationships

The `requires()` method is used to specify dependencies on other `Task` object:

```
def requires(self):  
    return Task1()
```



Luigi. Relationships

The `requires()` method is used to specify dependencies on other `Task` object:

```
def requires(self):  
    return Task1()
```

We can add Airflow-like relationships readability:

```
# Task1 >> Task2
```

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept
- ~~readable relationships~~
- DAG's owner

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept
- DAG's owner

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)

Airflow. DAG's owner



Airflow

DAGs

Data Profiling ▾

Browse ▾

Admin ▾

Docs ▾

About ▾

2019-10-21 17:37:28 UTC

DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	example	@daily	Ruslan Korniiichuk				

Showing 1 to 1 of 1 entries

«

<

1

>

»

Hide Paused DAGs

Ownership



Airflow

```
default_args = {  
    'owner': 'Ruslan Kornichuk'  
}
```

Luigi

We can add Airflow-like ownership:

```
# Owner: Ruslan Kornichuk
```

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept
- ~~DAG's owner~~

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

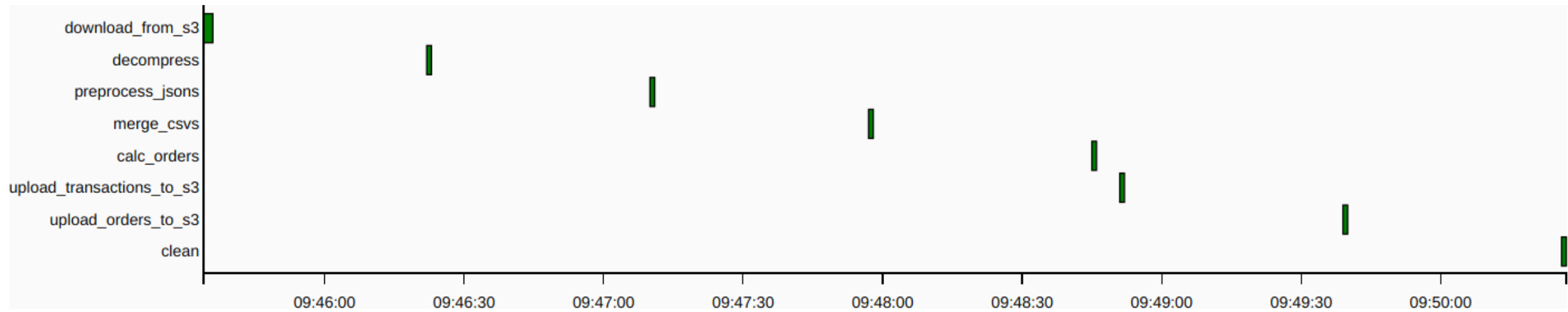
cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)



Airflow. Delay between task execution



Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept

cons

- delay between task execution
- cannot test/run task and all its relationships
- cross-communication via XCom

[example](#)

Luigi

pros

- rapid prototyping
- cross-communication
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)



XCom lets tasks exchange messages, allowing more nuanced forms of control and shared state. The name is an abbreviation of "cross-communication".

XCom can be "pushed" (sent) or "pulled" (received). When a task pushes an XCom, it makes it generally available to other tasks.



Airflow. Cross-communication

```
def task1():
    return 'task1'

def task2(**context):
    text = context['ti'].xcom_pull(task_ids='task1')
    with open('task2', 'w') as f:
        f.write(text)

...
task1 = PythonOperator(task_id='task1', python_callable=task1)
task2 = PythonOperator(task_id='task2', python_callable=task2,
                        provide_context=True)

...
```



Luigi. Cross-communication

```
class Task1(ExternalProgramTask):  
  
    dst = 'task1'  
  
    def program_args(self):  
        return ['touch', self.dst]  
  
    def output(self):  
        return luigi.LocalTarget(self.dst)
```

```
class Task2(luigi.Task):  
  
    def requires(self):  
        return Task1()  
  
    def run(self):  
        dst = self.input().path  
        with open(dst, 'a') as f:  
            f.write('task2')
```

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept

cons

- delay between task execution
- cannot test/run task and all its relationships
- ~~cross-communication via XCom~~

[example](#)

Luigi

pros

- rapid prototyping
- ~~cross-communication~~
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept

cons

- delay between task execution
- cannot test/run task and all its relationships

[example](#)

Luigi

pros

- rapid prototyping
- built-in Target
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)



Luigi. Target

The `output()` method returns one or more Target objects.:

```
def output(self):  
    return luigi.LocalTarget('file.txt')
```



Luigi. Target

The `output()` method returns one or more Target objects.:

```
def output(self):  
    return luigi.LocalTarget('file.txt')
```

or

```
def output(self):  
    return S3Target('s3://bucket/file.txt')
```



Airflow. Luigi-like Target

```
os.path.exists('file.txt')
```



Airflow. Luigi-like Target

```
os.path.exists('file.txt')
```

or

```
s3 = boto3.resource('s3')
```

```
try:
```

```
    s3.Object('bucket', 'file.txt').load()
```

```
except botocore.exceptions.ClientError as e:
```

```
    if e.response['Error']['Code'] == '404':
```

```
        # The object does not exist
```

```
    else:
```

```
        raise
```

```
else:
```

```
    # The object does exist
```

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept

cons

- delay between task execution
- cannot test/run task and all its relationships

[example](#)

Luigi

pros

- rapid prototyping
- ~~built-in Target~~
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)

Airflow vs Luigi



Airflow

pros

- DAGs catalog
- annoying scheduler
- readable DAG concept

cons

- delay between task execution
- cannot test/run task and all its relationships

[example](#)

Luigi

pros

- rapid prototyping
- easy-to-start

cons

- no DAG concept and DAGs catalog
- no scheduling

[example](#)



AWS

AWS. Simple Workflow Service (SWF)



Services ▾

Resource Groups ▾



korniichuk ▾

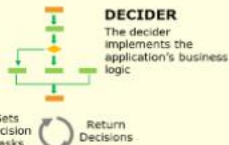
Ireland ▾

Support ▾

Amazon Simple Workflow Service Dashboard

Welcome to Amazon Simple Workflow Service (Amazon SWF)

Amazon Simple Workflow Service (Amazon SWF) is a workflow service for building scalable, resilient applications. With Amazon SWF, you can build many kinds of applications as workflows. Amazon SWF maintains the execution state of the workflow consistently and reliably so that you can focus on building and running your application. For more details on Amazon SWF, [click here](#).



Amazon SWF

- Maintains distributed application state
- Tracks workflow executions
- Ensures consistency of execution history
- Provides visibility into executions
- Holds and dispatches tasks
- Provides control over task distribution
- Retains workflow execution history

Get Activity Tasks
Return Results

Cloud



Workers for Activity 1

Get Activity Tasks
Return Results

Mobile



Workers for Activity 2

Get Activity Tasks
Return Results

On Premises



Workers for Activity 3

Run a Sample Workflow

The sample workflow introduces you to basic concepts in Amazon SWF. Running this sample will introduce you to basic concepts of Amazon SWF and also create resources such as a domain and types that you can reuse.

Launch sample walkthrough

Create your own domain

You can create a domain and register your own workflows and activities. Click below to get started.

Create a new Domain

AWS. Step Functions



Using **Step Functions**, you can design and run workflows that stitch together services such as AWS Lambda and Amazon ECS.

 Services ▾ Resource Groups ▾ 

 korniichuk ▾ Ireland ▾ Support ▾

Step Functions 

State machines

Activities

Step Functions > State machines

State machines (1)

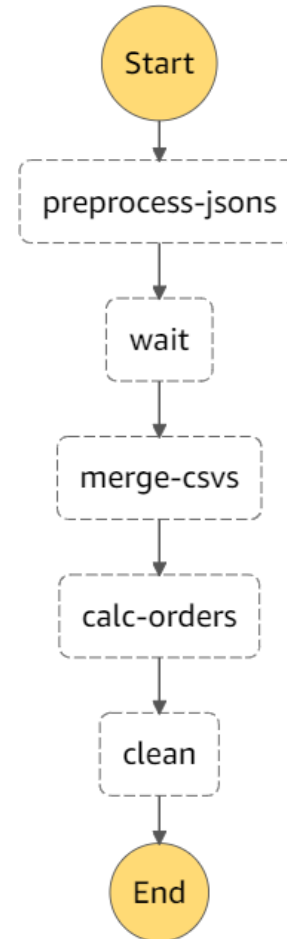
 View details Edit Copy to new Delete **Create state machine**

< 1 >



	Name ▾	Creation date ▾	Status	Running	Succeeded	Failed	Timed out	Aborted
	orders	Aug 19, 2019 11:12:49.871 PM	Active	0	10	5	0	0

AWS. Step Functions

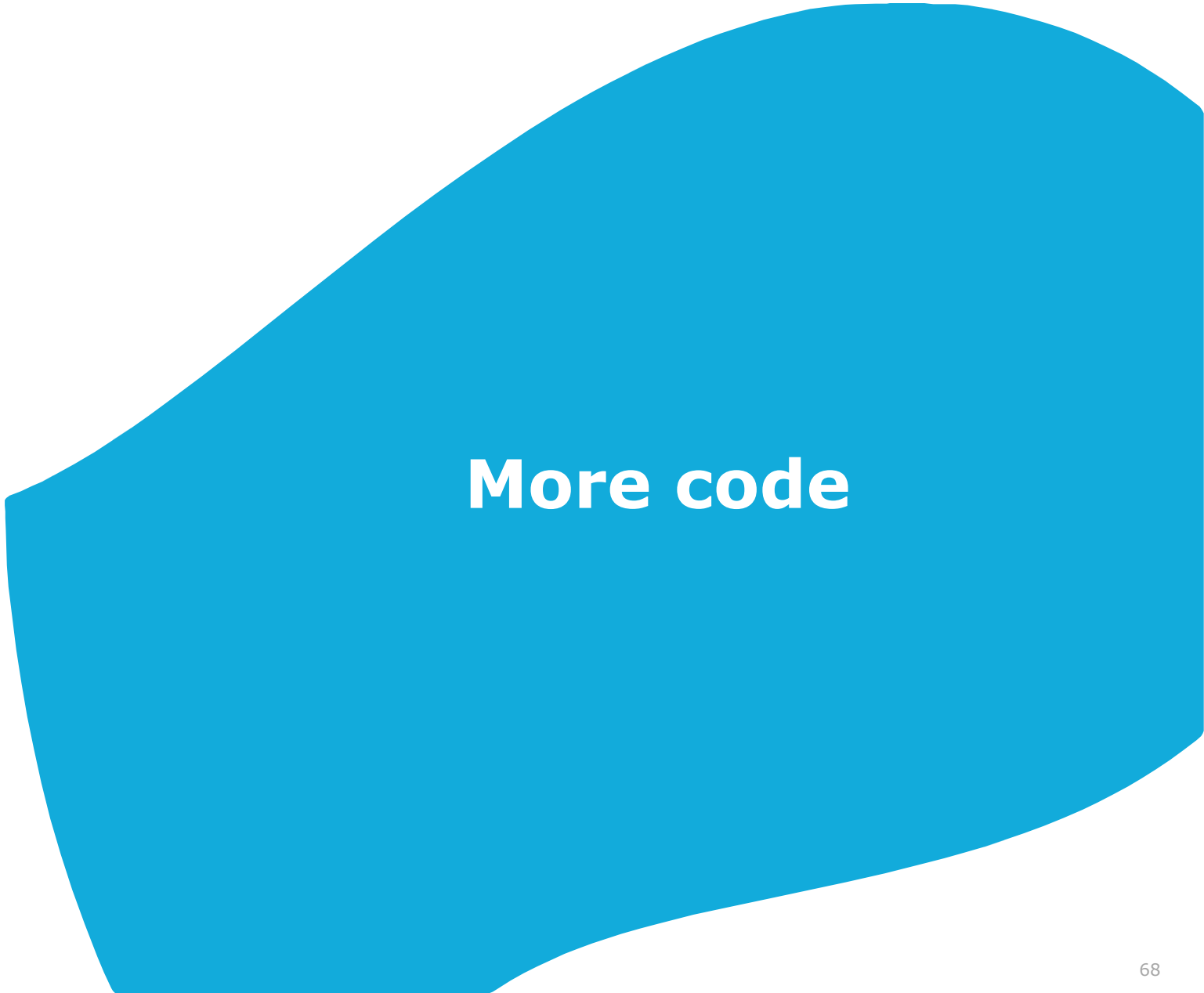




Amazon States Language

```
{
  "Comment": "orders ver. 0.1a1 by Ruslan Korniiichuk",
  "StartAt": "preprocess-jsons",
  "States": {
    "preprocess-jsons": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-1:539199393808:function:workflow-preprocess-jsons",
      "Next": "wait"
    },
    "wait": {
      "Type": "Wait",
      "Seconds": 5,
      "Next": "merge-csvs"
    },
    "merge-csvs": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-1:539199393808:function:workflow-merge-csvs",
      "Next": "calc-orders"
    }
  }
}
```

```
    "calc-orders": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-1:539199393808:function:workflow-calc-orders",
      "Next": "clean"
    },
    "clean": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-1:539199393808:function:workflow-clean",
      "End": true
    }
  }
}
```



More code

[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)[korniichuk / workflow](#)[Unwatch](#)

1

[★ Star](#)

0

[Fork](#)

0

[Code](#)[Issues](#) 0[Pull requests](#) 0[Projects](#) 0[Wiki](#)[Security](#)[Insights](#)[Settings](#)

In progress... Workflow management platforms comparison <http://www.korniichuk.com>

[Edit](#)[dataops](#)[airflow](#)[luigi](#)[aws](#)[Manage topics](#)[55 commits](#)[1 branch](#)[0 releases](#)[1 contributor](#)[Unlicense](#)[Branch: master](#)[New pull request](#)[Create new file](#)[Upload files](#)[Find file](#)[Clone or download](#)[korniichuk](#) [workflow][luigi] Add Airflow-like relationships

Latest commit 8c690c3 yesterday

input	[workflow] Add input data	3 months ago
workflow-airflow	[workflow][airflow] Change owner to 'Ruslan Korniichuk'	yesterday
workflow-aws	[workflow][aws] Add owner to Step Fucntions	2 months ago
workflow-luigi	[workflow][luigi] Add Airflow-like relationships	yesterday
.gitignore	[workflow][aws] Add workflow-calc-orders Lambda	2 months ago
LICENSE	Initial commit	3 months ago
README.md	Initial commit	3 months ago
fabfile.py	[workflow][aws] Fix repo name	2 months ago

[README.md](#)

About Capgemini

A global leader in consulting, technology services and digital transformation, Capgemini is at the forefront of innovation to address the entire breadth of clients' opportunities in the evolving world of cloud, digital and platforms. Building on its strong 50-year heritage and deep industry-specific expertise, Capgemini enables organizations to realize their business ambitions through an array of services from strategy to operations. Capgemini is driven by the conviction that the business value of technology comes from and through people. It is a multicultural company of over 200,000 team members in more than 40 countries. The Group reported 2018 global revenues of EUR 13.2 billion.

Learn more about us at

www.capgemini.com



People matter, results count.

This presentation contains information that may be privileged or confidential and is the property of the Capgemini Group.
Copyright © 2019 Capgemini. All rights reserved.

Ruslan Korniiichuk

Senior Consultant

Capgemini

ruslan.korniichuk@capgemini.com

+48 791 065 877