DataOps 3.0

8th of December 2020

The 2nd law of IT

The first 80% of the development are not as hard as **the second 80%**.

Source: bit.ly/3pnwl4C

DataOps

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

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

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 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 pytest, flake8, and regular code reviews. Automate tests.

Workflow managers

Scripts with Cron

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

Python Schedule

```
import time

import schedule


def job():
    print("I'm working...")


schedule.every(10).minutes.do(job)
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)
schedule.every().monday.do(job)
schedule.every().wednesday.at("13:15").do(job)
schedule.every().minute.at(":17").do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

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)

XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.3/" xml:lang="uk">
  <page>
    <title>Фукідід</title>
    <id>1529</id>
    <revision>
      <id>4382</id>
      <timestamp>2006-09-18T22:11:53Z</timestamp>
      <minor />
      <comment>Interwiki</comment>
      <text xml:space="preserve">{{Wikipedia}}
      *Історія — це філософія в прикладах.
      </text>
    </revision>
  </page>
</mediawiki>
```

Source: uk.wikipedia.org/wiki/XML

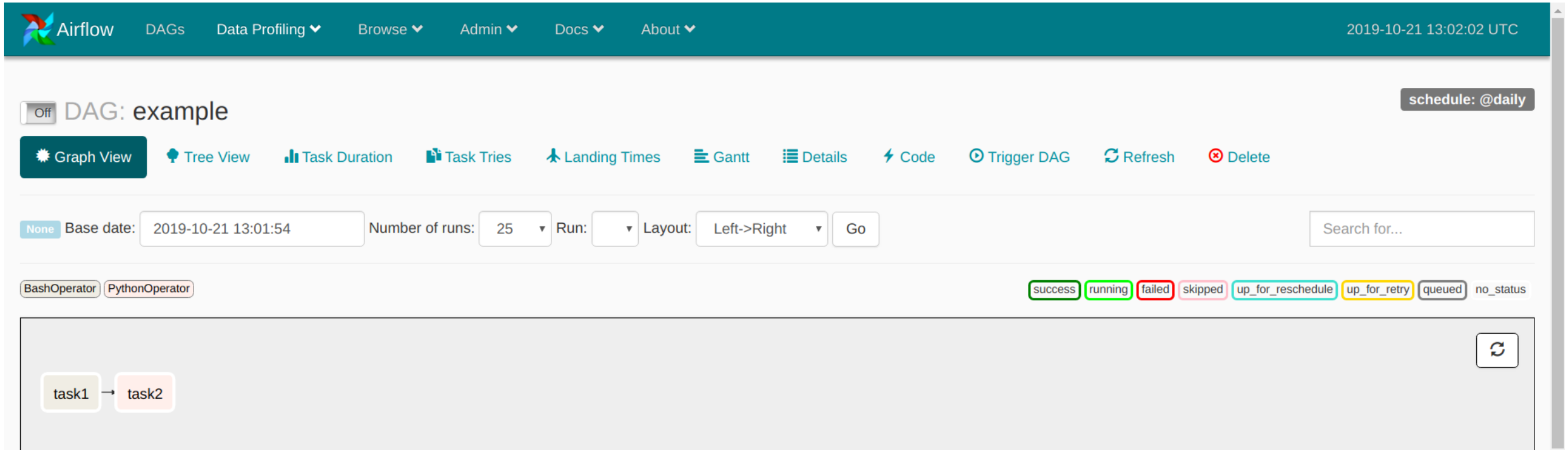
Workflow managers

- 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

DAG

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



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 arguments

```
default_args = {
    'owner': 'Ruslan Korniichuk',
    'start_date': datetime.datetime(2020, 12, 07)
}
```

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)
```

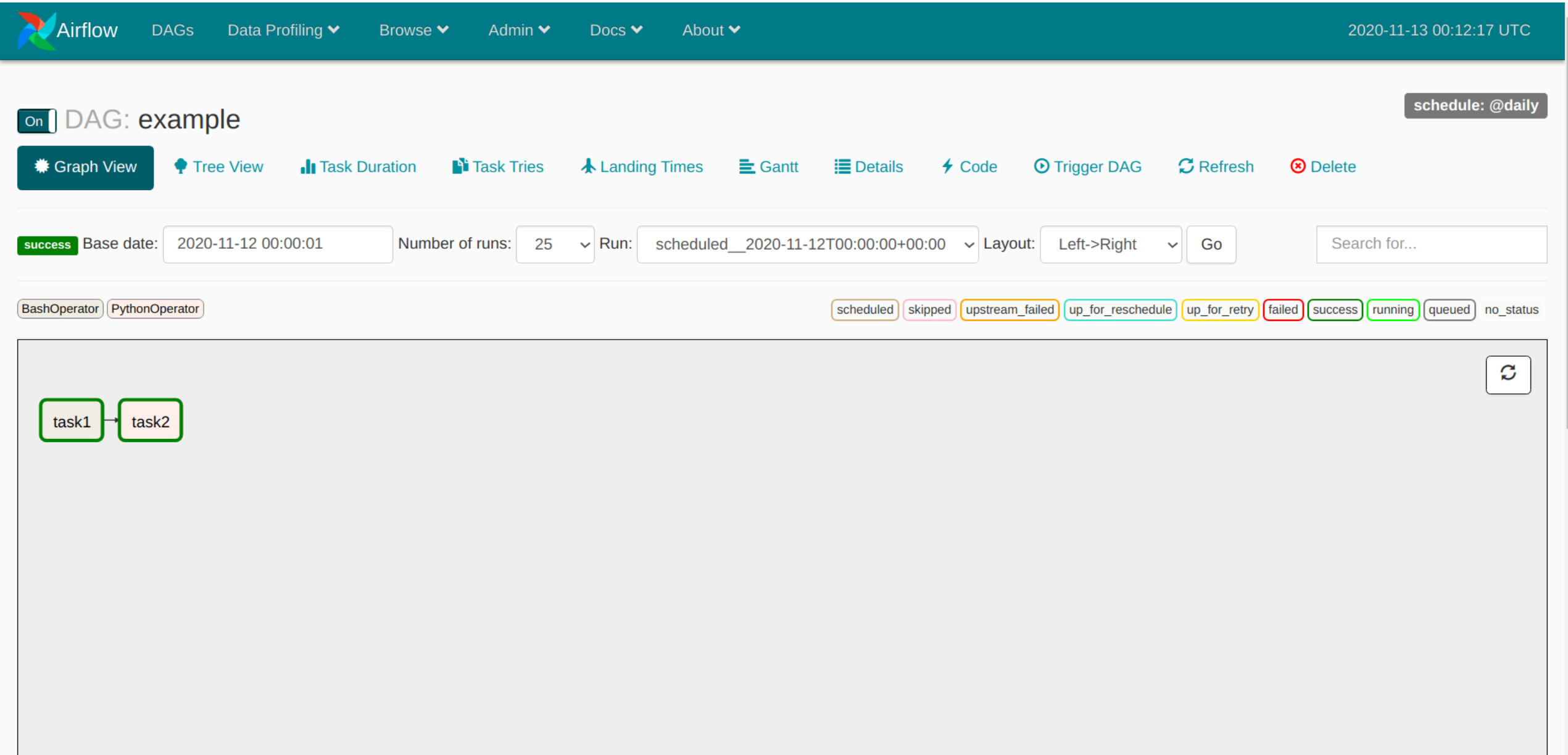
Airflow. Relationships

```
task1 >> task2
```

Airflow. All-in-one

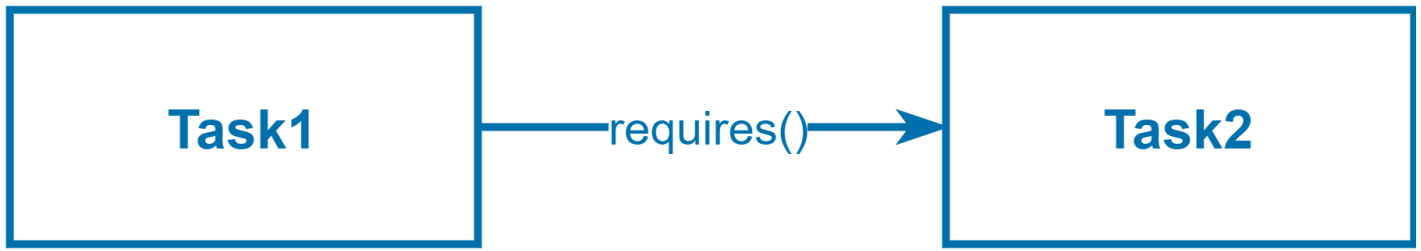
```python
import datetime

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 Korniichuk',
    'start_date': datetime.datetime(2020, 12, 07)
}

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

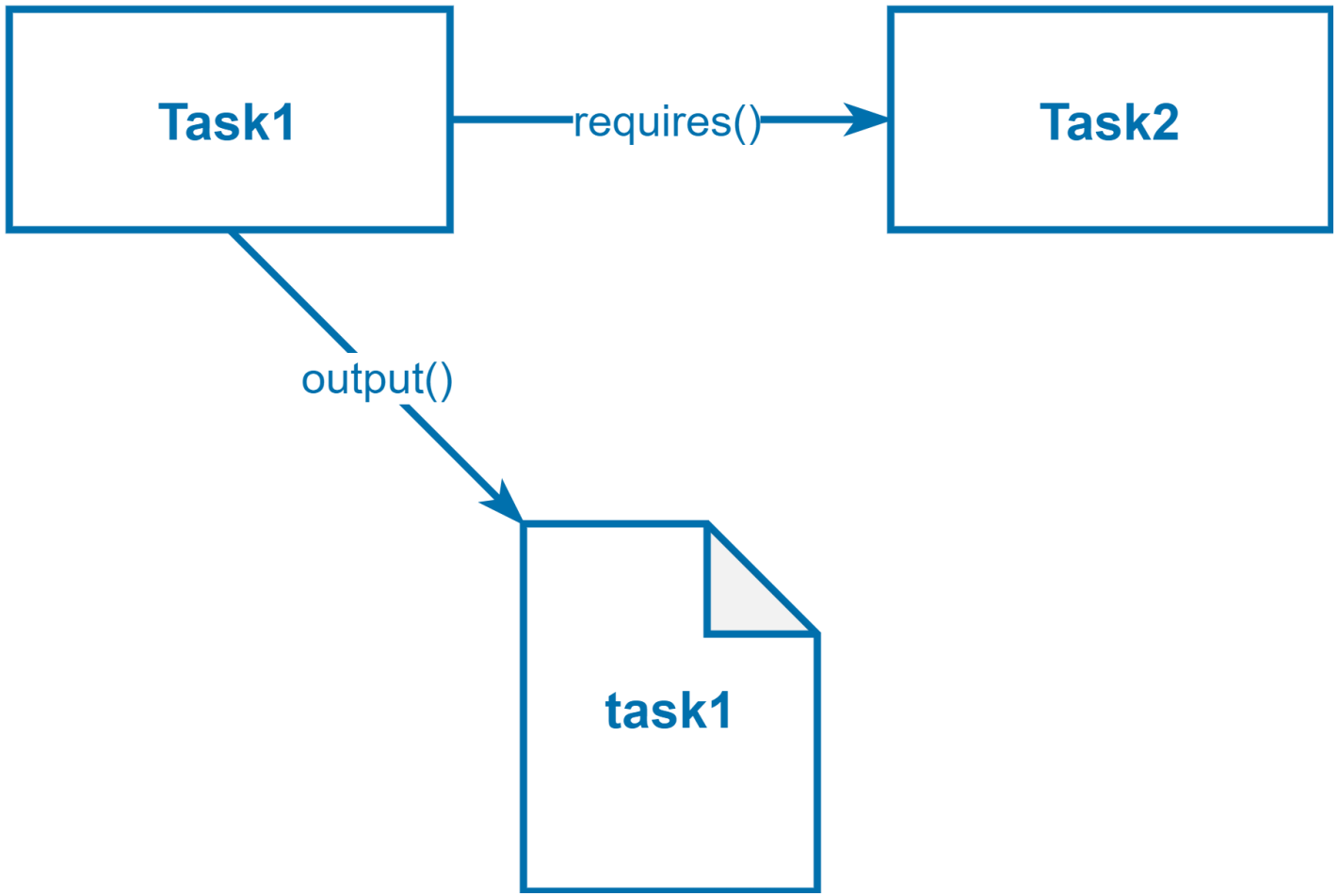


Luigi

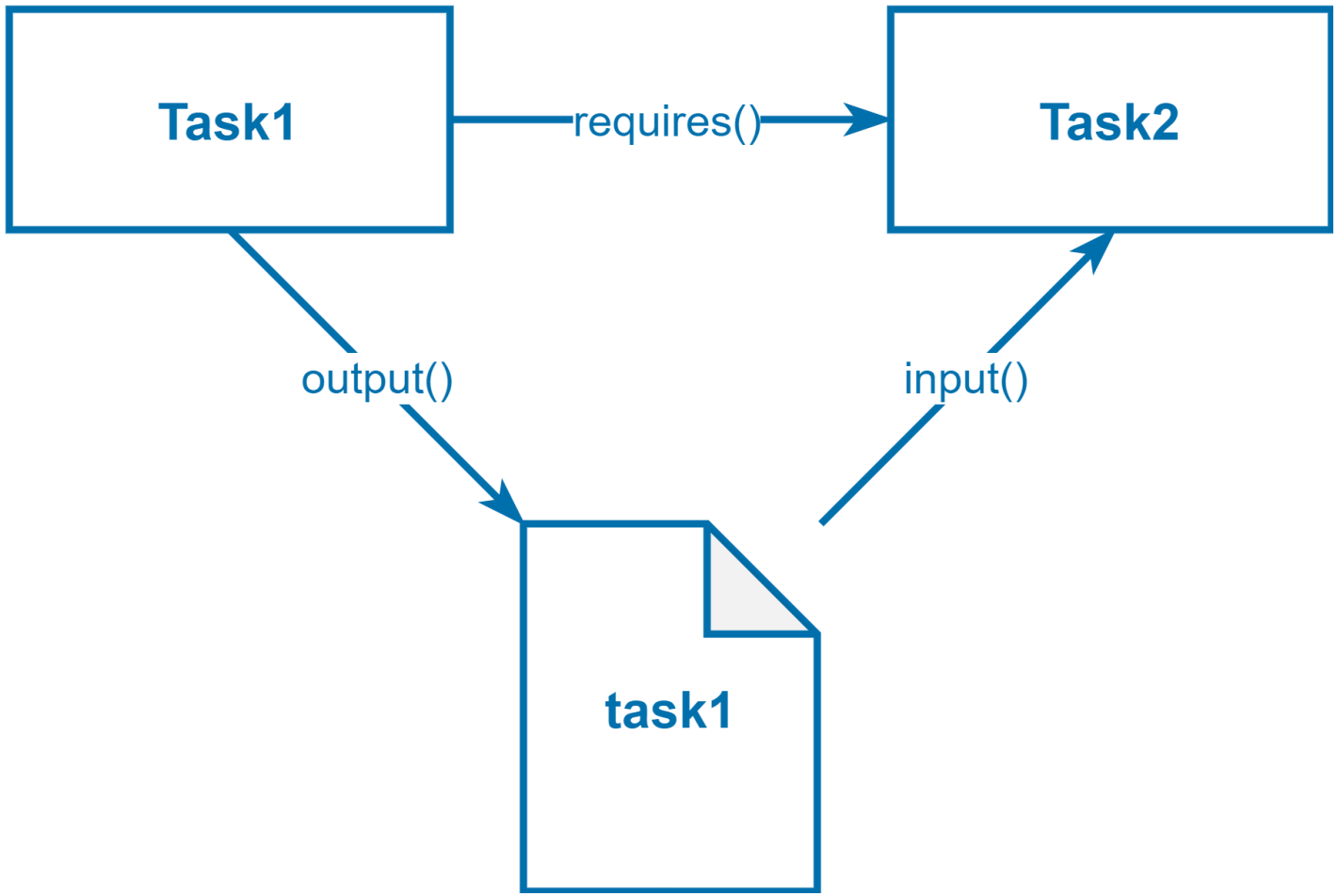
Luigi. Core idea



Luigi. Core idea



Luigi. Core idea



Luigi. Import

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

Luigi. Tasks

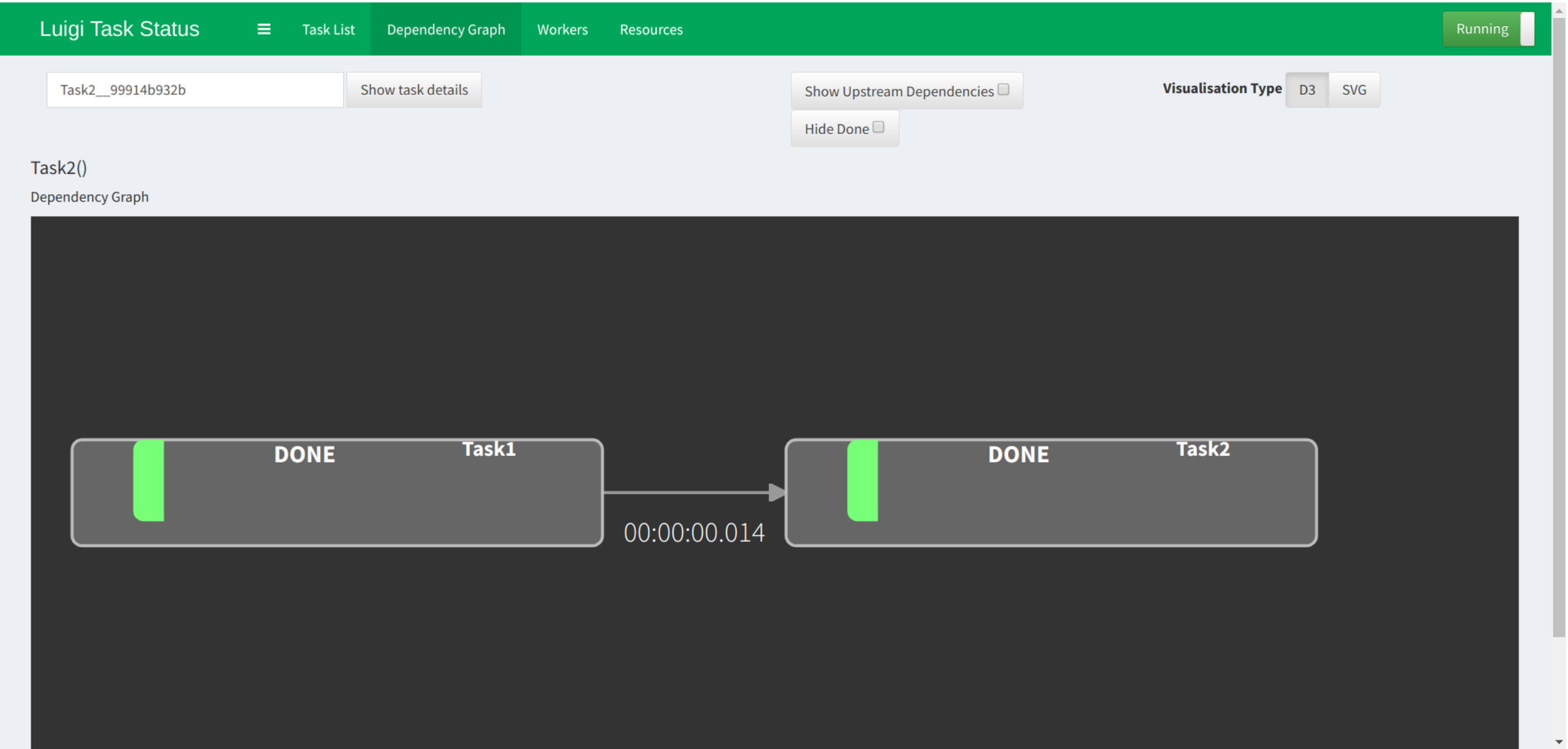
```python
class Task1(ExternalProgramTask):

    dst = 'task1'

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

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

```python
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

```python
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. Graph view



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

example

Luigi

pros

- everything in Python (no XML)
- rapid prototyping
- 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~~

example

Luigi

pros

- ~~everything in Python (no XML)~~
- rapid prototyping
- 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

example

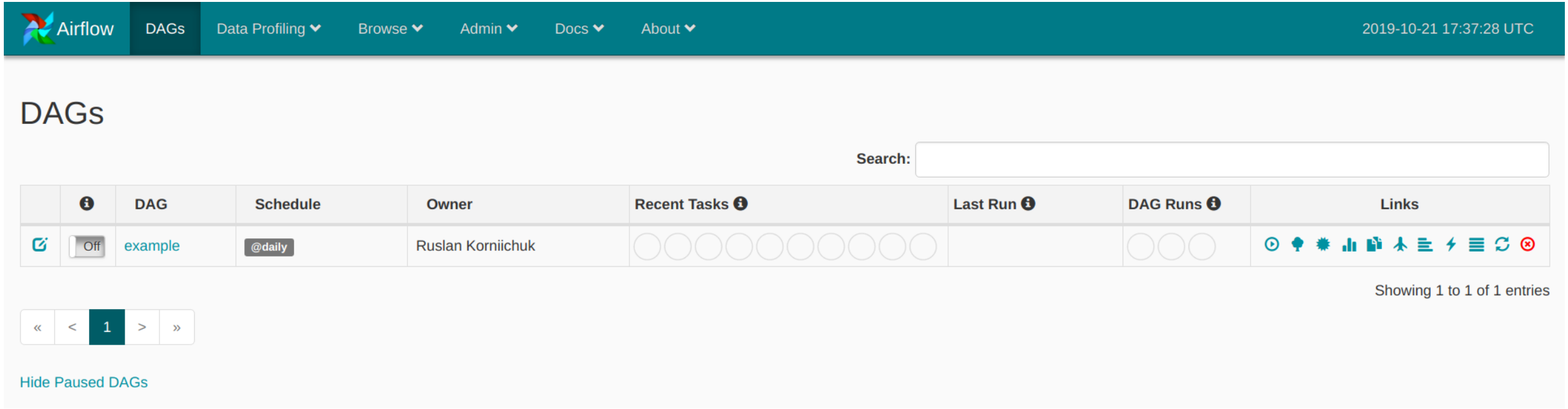
Luigi

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

cons
- no DAG concept and DAGs catalog
- no scheduling

example

Airflow. DAGs catalog



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

example

Luigi

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

cons
- no DAG concept and DAGs catalog
- no scheduling

example

Airflow. Relationships

Can be done with the Python bitshift operators >> and <<. The following two statements are functionally equivalent:

```
task1 >> task2
```

or

```
task2 << task1
```

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

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
- DAG's owner

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

example

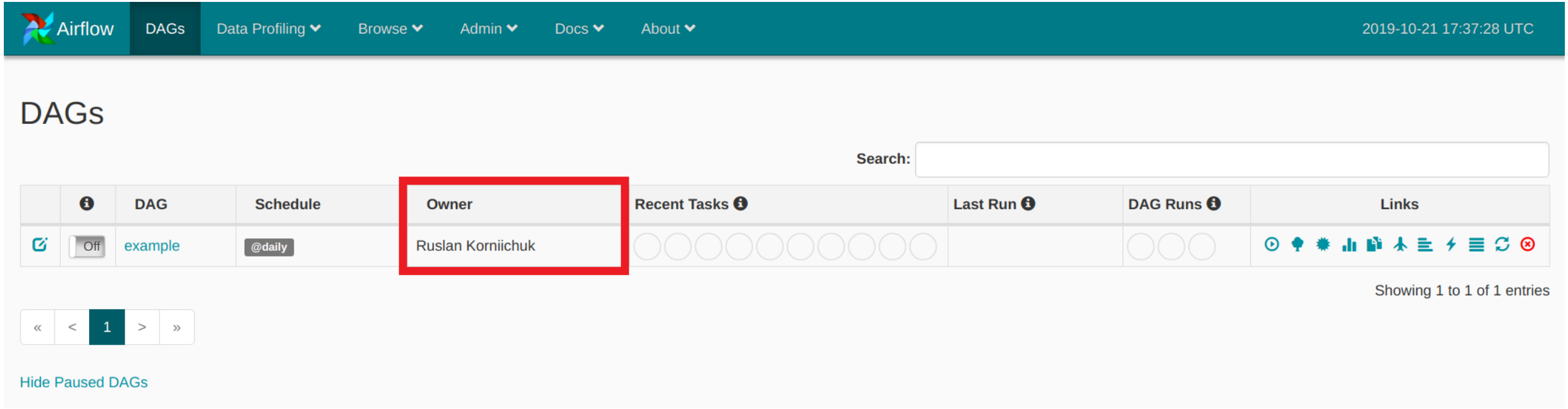
Luigi

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

cons
- no DAG concept and DAGs catalog
- no scheduling

example

Airflow. DAG's owner



Ownership

Airflow

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

Luigi

We can add Airflow-like ownership:

```
    # Owner: Ruslan Korniichuk
```

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

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

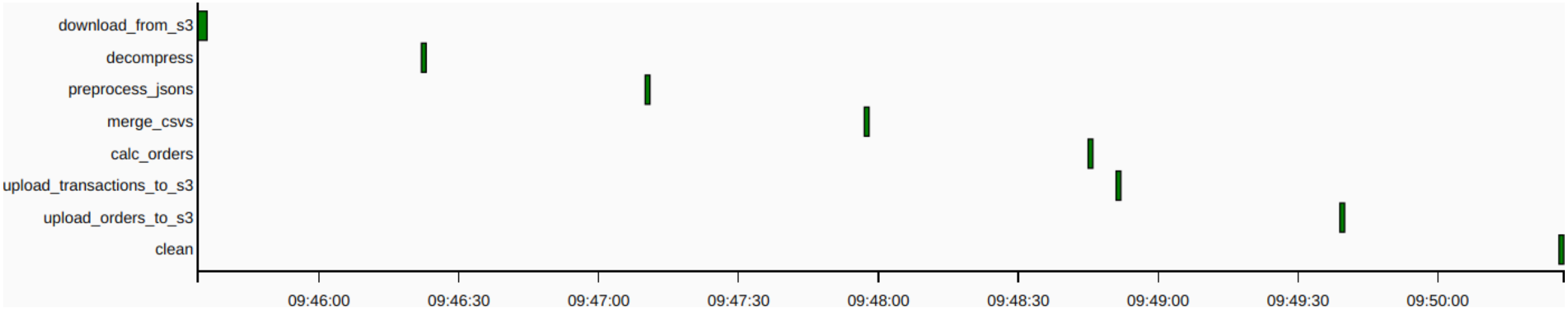
Luigi

pros
- rapid prototyping
- 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

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

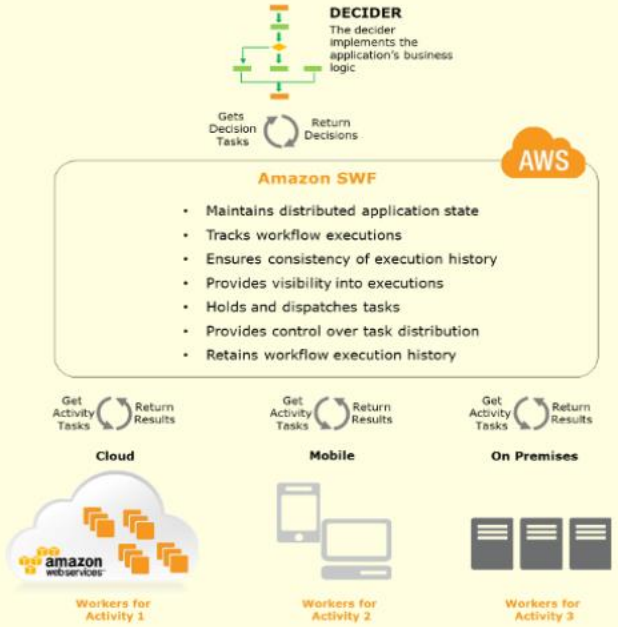
Simple Workflow Service (SWF)

aws 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.



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

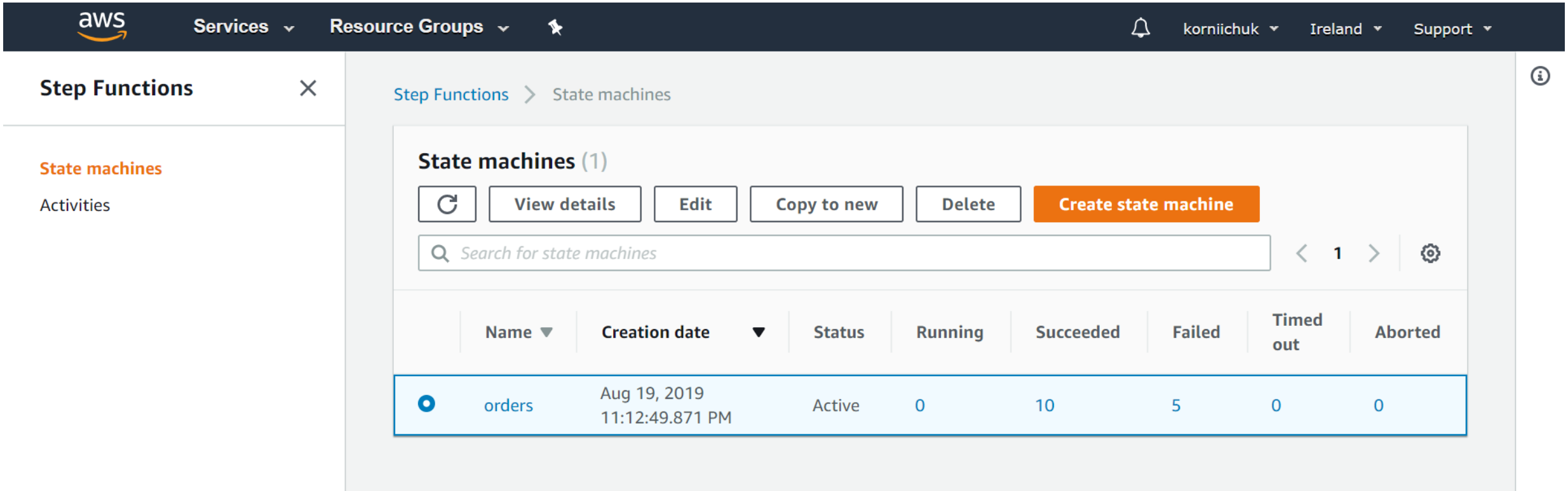
Create your own domain

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

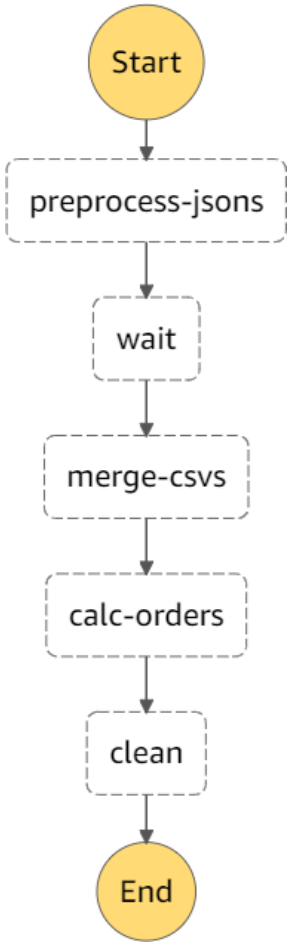
Create a new Domain

Step Functions

Using **Step Functions**, you can design and run workflows that stitch together services such as Lambda, Fargate, Glue.



Step Functions



Amazon States Language

```
{
  "Comment": "orders ver. 0.1a1 by Ruslan Korniichuk",
  "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

korniichuk / workflow

master 2 branches 0 tags Go to file Add file Code

korniichuk Update requirements.txt 7f86a77 13 seconds ago 56 commits

input	[workflow] Add input data	15 months ago
workflow-airflow	Update requirements.txt	13 seconds ago
workflow-aws	[workflow][aws] Add owner to Step Fucntions	15 months ago
workflow-luigi	[workflow][luigi] Add Airflow-like relationships	13 months ago
.gitignore	[workflow][aws] Add workflow-calc-orders Lambda	15 months ago
LICENSE	Initial commit	16 months ago
README.md	Initial commit	16 months ago
fabfile.py	[workflow][aws] Fix repo name	15 months ago

README.md

worflow

Workflow management platforms comparison

About

Workflow management platforms comparison

www.korniichuk.com

dataops airflow luigi aws aws-step-functions step-functions apache-airflow

Readme

Unlicense License

Languages

Python 100.0%

Cześć! Jesteśmy SaaSem ze Śląska
i tworzymy 60-osobowy

#teamlandingi

Wykorzystujemy **Machine Learning**
oraz takie technologie jak: **Python, React,**
PHP (Symfony framework)

Dołącz do nas: **landingi.com/kariera**

Obserwuj nas na:

@landingi_com @landingi.kariera @landingi

About Capgemini

Capgemini is a global leader in consulting, digital transformation, technology and engineering services. The Group 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. Today, it is a multicultural company of 270,000 team members in almost 50 countries. With Altran, the Group reported 2019 combined revenues of €17billion.

Learn more about us at

www.capgemini.com

People matter, results count.

Ruslan Korniichuk

Information Technology Architect
Capgemini Poland
ruslan.korniichuk@capgemini.com
+48 791 065 877