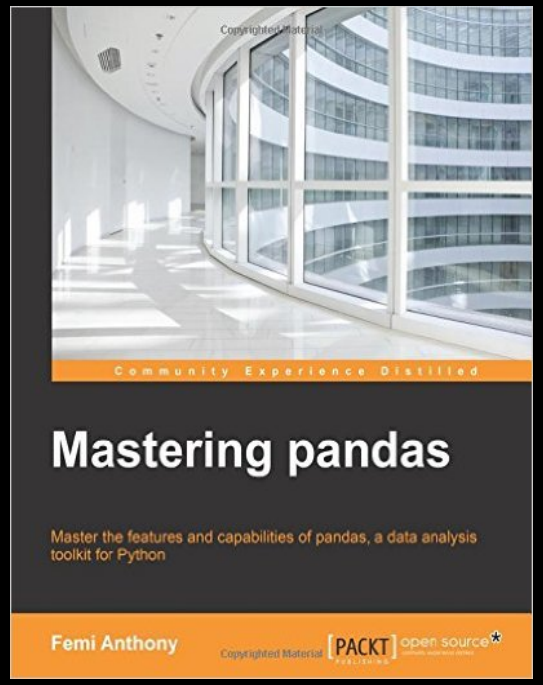
Creating Data Pipelines in the Cloud

Femi Anthony

Who am I

- Femi Anthony
- Twitter : @dataphanatik
- Email: femibyte@gmail.com
- Data Engineer, Capital One
- Book with PacktPub - Mastering Pandas



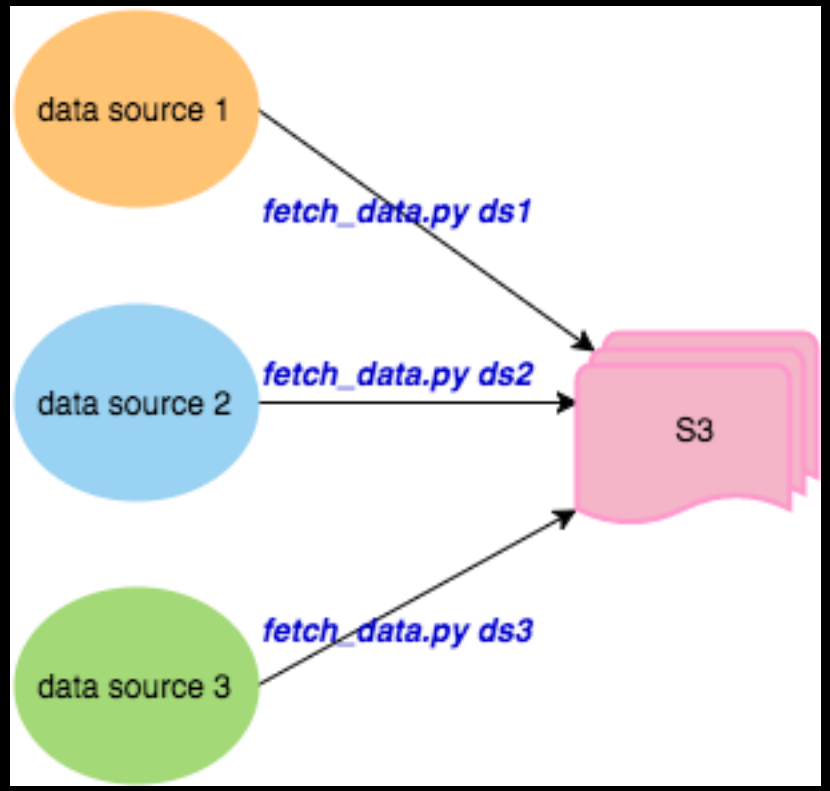
Overview

- Definition and Rationale for Data Pipelines
- Data Pipeline Tools - Luigi, Airflow, AWS Data Pipeline
- Usage in the cloud and comparative analysis

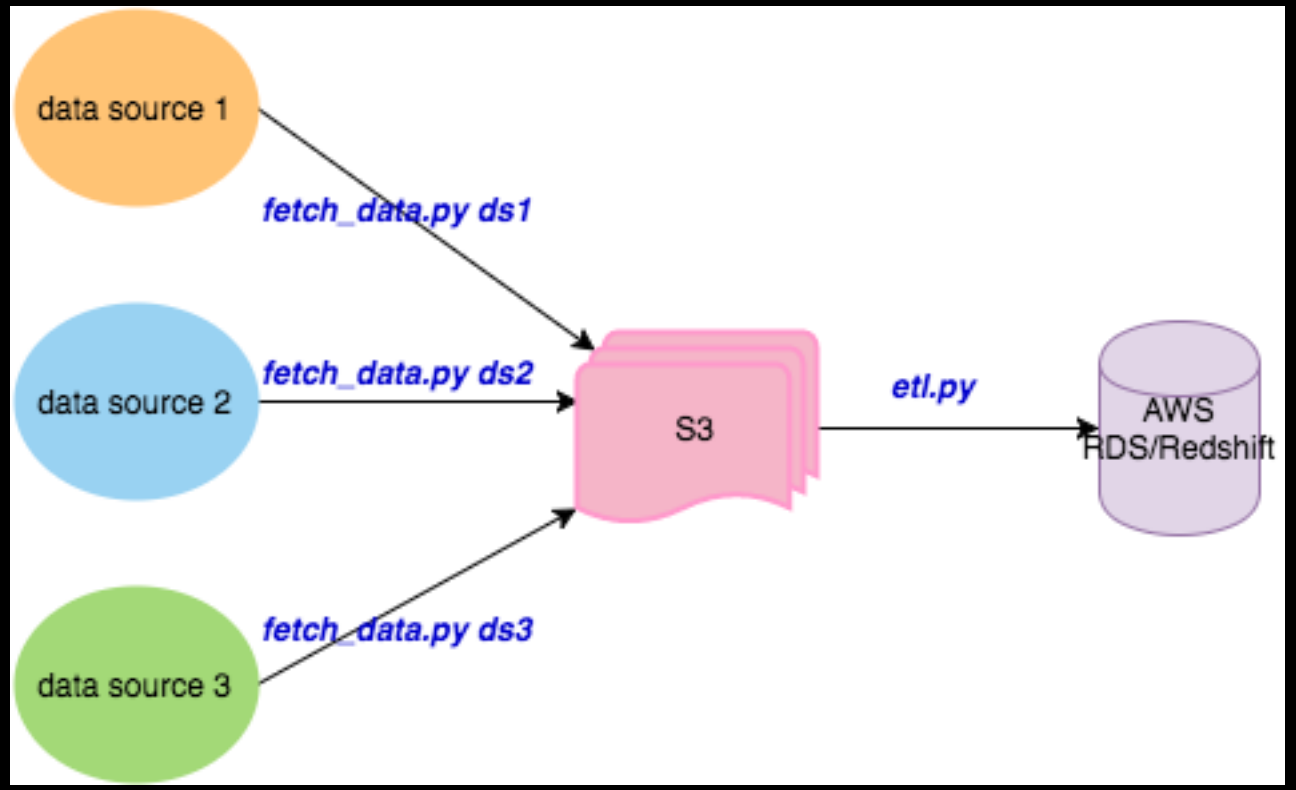
What is a data pipeline ?

- A data pipeline consists of a series of software tasks that source and extract data from possibly disparate sources, move it to a centralized location, process and transform it in a logically consistent manner and produce a set of meaningful results that can be consumed by a client/end user.

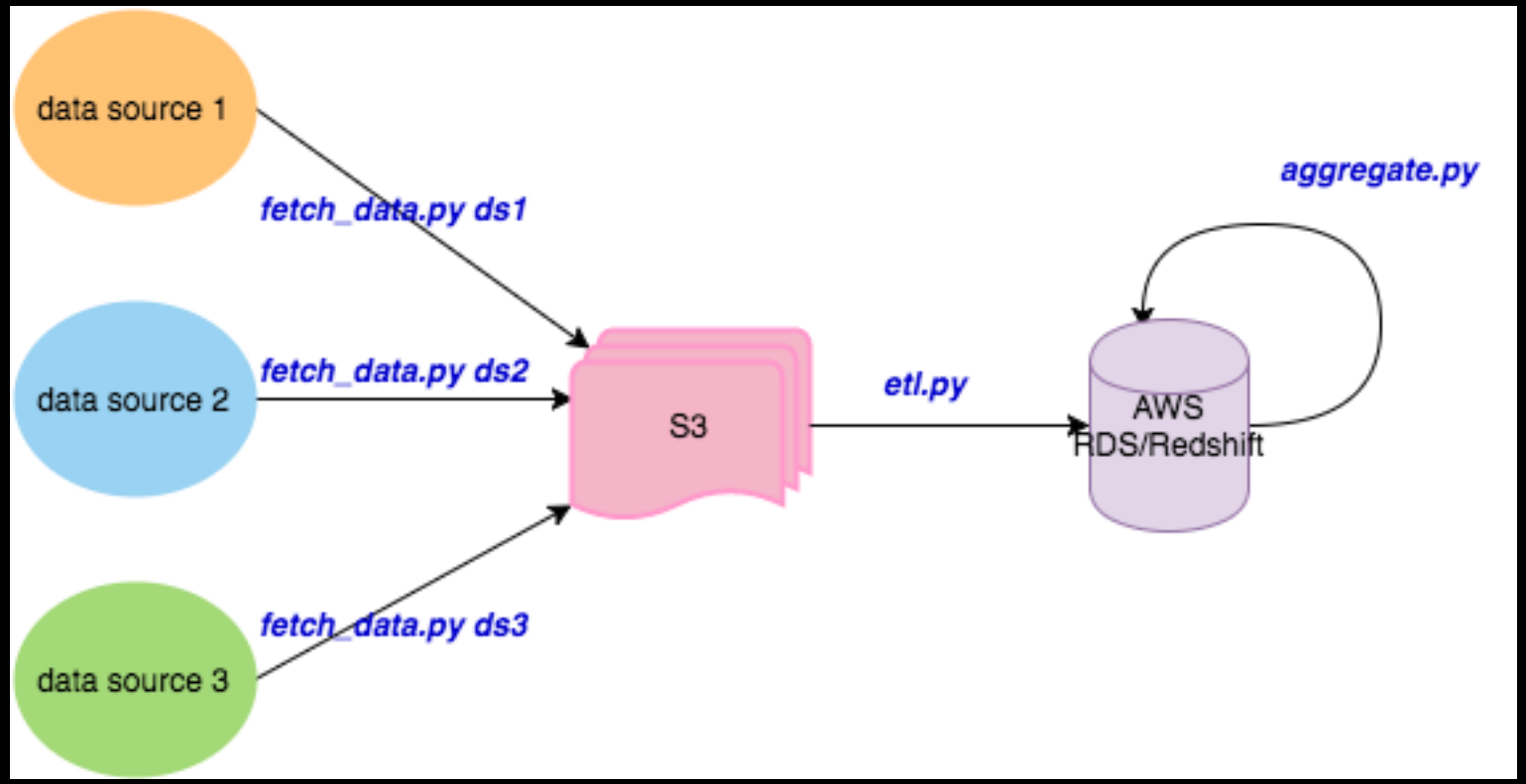
Data Pipeline Step 1 – Data Retrieval



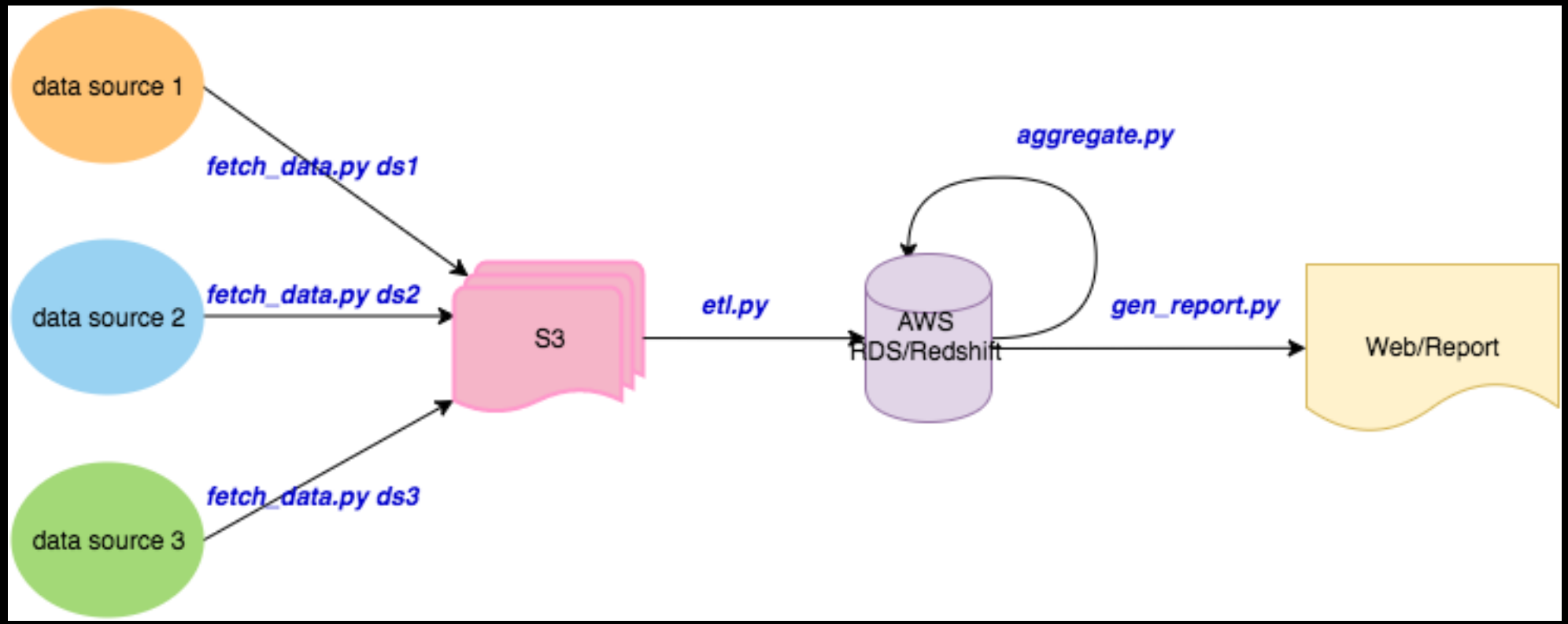
Data Pipeline Step 2 – Extract Transform Load



Data Pipeline Step 3 – Aggregate/Predict



Data Pipeline Step 4 – Write Results



Naïve approach : separate cron tasks

- Run the scripts in desired order as cron jobs and hope for the best:

```
0 8 * * * /home/ubuntu/bin/fetch_data.py ds1
0 8 * * * /home/ubuntu/bin/fetch_data.py ds2
0 8 * * * /home/ubuntu/bin/fetch_data.py ds3
30 8 * * * /home/ubuntu/bin/etl.py
0 9 * * * /home/ubuntu/bin/aggregate.py
0 10 * * * /home/ubuntu/bin/gen_report.py
```

Less Naïve approach

- Execute scripts in order in sequential order via single bash script :

```
#!/usr/bin/bash
dataset_loc1 = ...
dataset_loc2 = ...
dataset_loc3 = ...
workdir = ...
cd $workdir
# retrieve data
python fetch_data.py dataset_loc1 && python fetch_data.py dataset_loc2 \
&& python fetch_data.py dataset_loc3
# Do ETL
python run_etl.py s3_location
# Do aggregation
python aggregate.py
# Generate results
python gen_results.py
```

Less Naïve approach with Error handling

- Check the status of each task upon completion before running the next.

```
# Retrieve data
python fetch_data.py dataset_loc1 && python fetch_data.py dataset_loc2 \
&& python fetch_data.py dataset_loc3
# Do ETL
if [[ $? == 0 ]]; then
   python run_etl.py s3_location
# Do aggregation
else
   echo “error: $?” && exit -1
…
if [[ $? == 0 ]]; then
...
```

Solution: Use Workflow framework manager

- Luigi

- Airflow

- AWS Data Pipeline

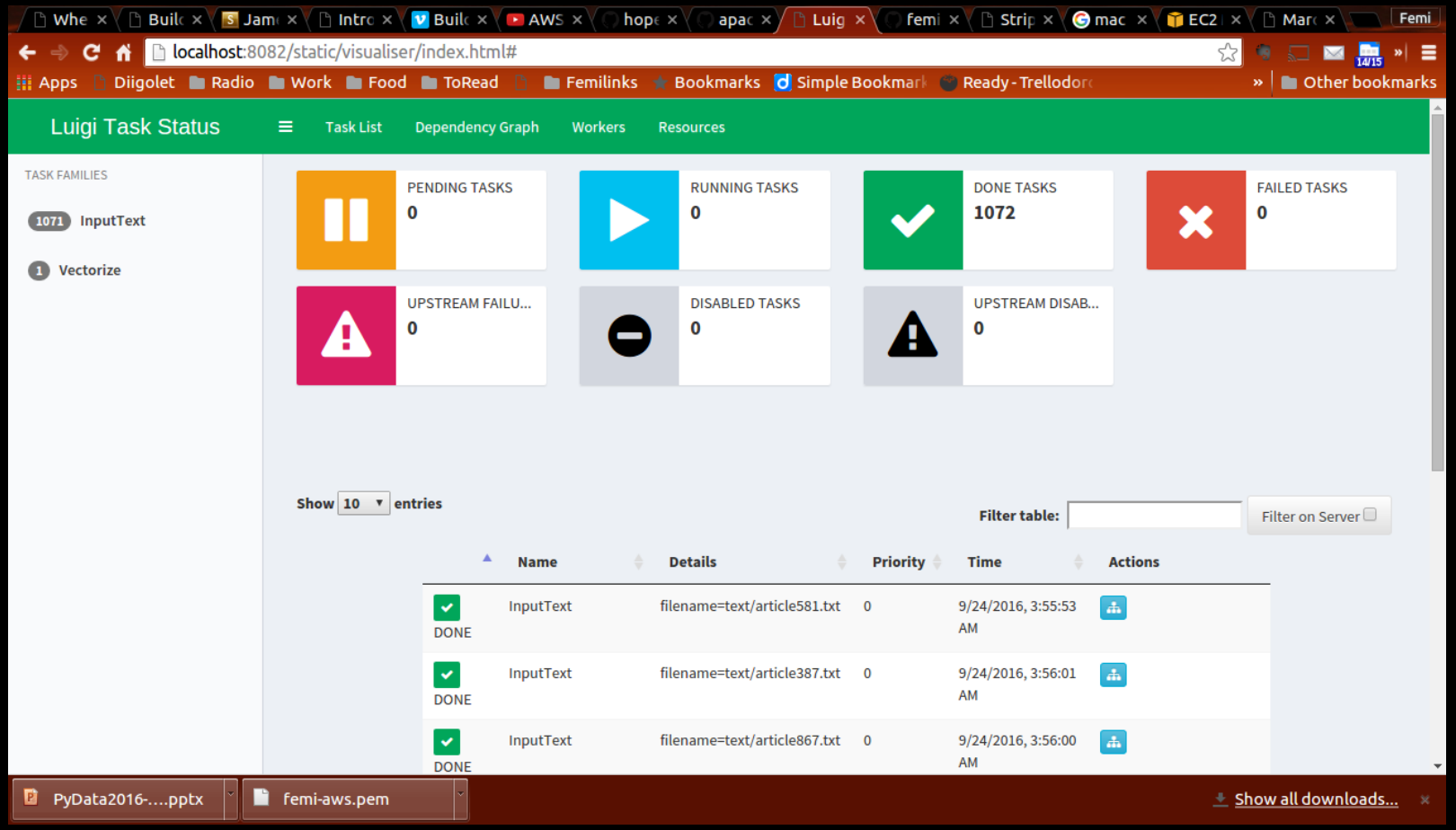
Introduction to Luigi

- Python package that helps build complex pipelines of batch jobs.
- It provides dependency resolution, workflow management, visualization, handling failures, command line integration.
- Created by folks at Spotify
- Maintained by Erik Bernardson.
- First open-source Python based workflow manager.
- Named after the world's 2nd most famous plumber

Philosophy and Concepts

- Similar to GNU Make where one can define tasks and tasks depend on other tasks
- Dependency graph is specified in Python code.
- Provides UI, can be run in server mode or with local-scheduler mode
- Two main abstractions :
  - **Task** – Responsible for execution of a task. It is subclassed from the abstract class *`luigi.Task`*.
  - **Target** – Corresponds to a location for writing output data from a Task run. Examples : file on disk, file in S3/HDFS, checkpoint entry in database.

Luigi UI


localhost:8082/static/visualiser/index.html#
Luigi Task Status
Task List
Dependency Graph
Workers
Resources
TASK FAMILIES
1071 InputText
1 Vectorize
PENDING TASKS 0
RUNNING TASKS 0
DONE TASKS 1072
FAILED TASKS 0
UPSTREAM FAILU... 0
DISABLED TASKS 0
UPSTREAM DISAB... 0
Show 10 entries
Filter table:
Filter on Server
Name
Details
Priority
Time
Actions
DONE InputText filename=text/article581.txt 0 9/24/2016, 3:55:53 AM
DONE InputText filename=text/article387.txt 0 9/24/2016, 3:56:01 AM
DONE InputText filename=text/article867.txt 0 9/24/2016, 3:56:00 AM
PyData2016-....pptx
femi-aws.pem
Show all downloads...

Task class Implementation

- Implement a Task, subclass *luigi.Task* and implement the following 3 methods:
  - `run()`
    - contains the logic needed to achieve purpose of Task
  - `output()`
    - returns one or more Target objects.
  - `requires()`
    - specifies dependencies on a preceding Task object.

Task class Example


```
class ReportTask(luigi.Task):
    self.end_date= luigi.Parameter(end_date='latest')

    def run(self):
        gen_report(end_date)

    def requires(self):
        return AggregateTask()

    def output(self):
       report_path = ("%s/%s.txt" %(bucket, end_date))
       return luigi.s3.s3Target(report_path)
if __name=="__main__":
    luigi.run()
```

Examples of Target

- MyTarget(luigi.Target)
- luigi.s3Target
- luigi.HDFSTarget
- luigi.LocalTarget

Advantages

- Mature, been around the block
- Platform agnostic – not limited to say just Hadoop like frameworks such as Oozie.
- Native support for HDFS, S3, postgres, MySQL, Redshift, Spark, BigQuery etc luigi.postgres,
- Decent visualization tool (UI) to track progress of job.
- Decentralized dependencies.

Limitations

- No built-in task triggering/scheduling. Reliant on cron or similar mechanism to trigger pipeline runs.
- Focus is on batch processing so not as useful for real-time pipelines or stream processing.
- Relatively small number of Tasks, because requires writing subclasses for unique tasks.
- Requires writing to file at every stage - cannot write output from 1 stage to another via in-memory buffer.

Introduction to Airflow

- Workflow management and orchestration tool
- Written in Python.
- Created by folks at Airbnb
- Maintained as an Apache Foundation project
- Large set of features including UI, scheduler, command line interface

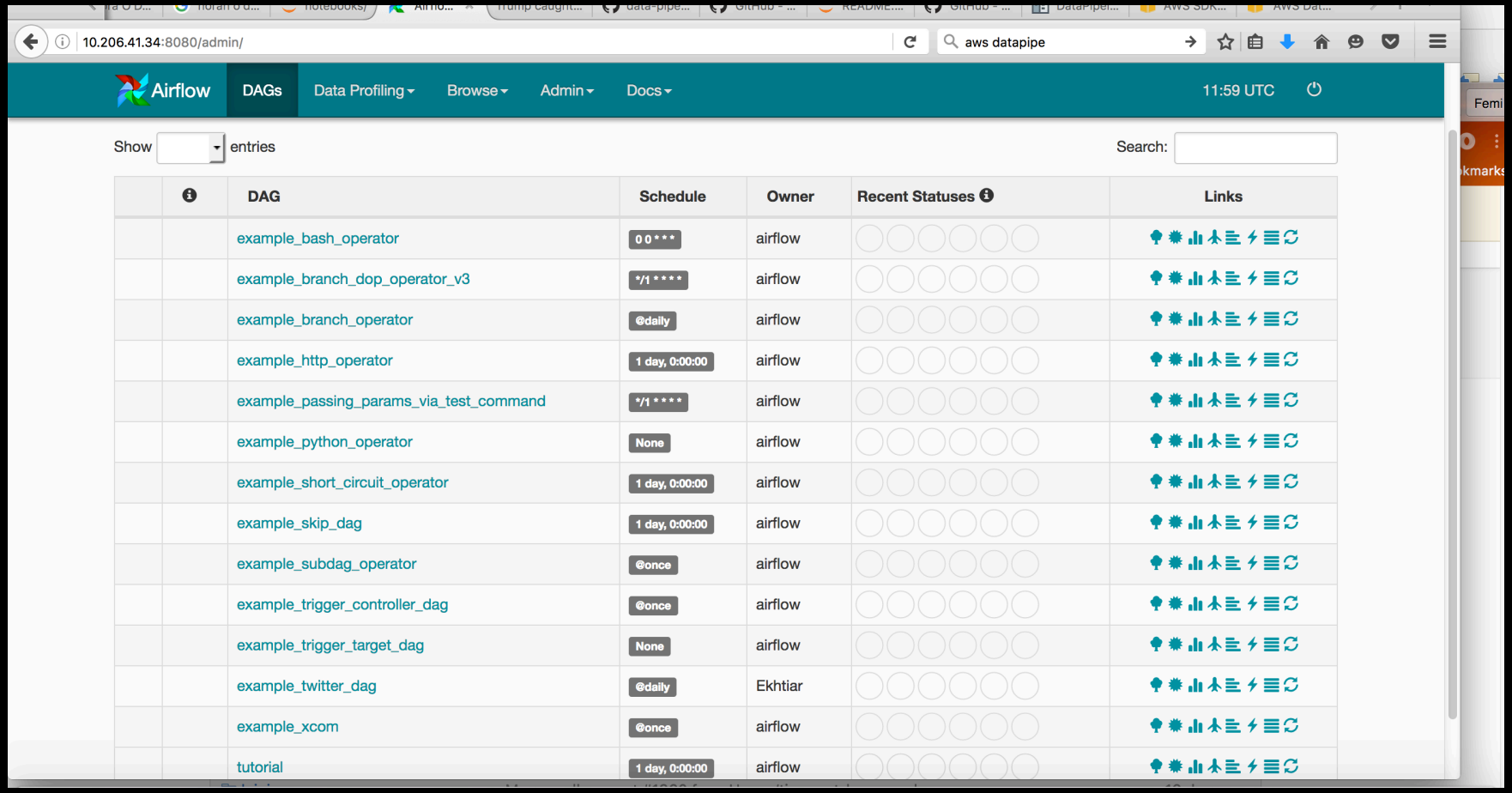
Airflow Design Principles

- Core Abstraction : DAG (directed acyclic graph)
- Pipelines are modeled as DAGs and tasks are nodes on the DAG while dependencies are paths connecting the nodes.
- Cycles are not allowed (pipelines need to run to completion)
- Can run tasks that are independent in parallel
- Task failures in independent parts of the pipeline do not derail other parts.
- Can re-run parts of workflow that have been affected by a failure

Airflow Components

- **Job definitions** - stored in source control
- **Metadata DB** - MySQL, Postgres used by Airflow to keep track of metadata such as task statuses.
- **Command line interface (CLI)** – to test, run, backfill, describe and clear parts of pipeline definitions.
- **Rich Web interface** – for viewing task dependencies, logs, statuses, and other metadata. Implemented as a Flask web app.
- **Scheduler** – cron replacement that kicks off tasks that are scheduled to run.
- **Workers** – array of processes that run tasks in a distributed fashion.

Airflow UI


10.206.41.34:8080/admin/
aws datapipe
Airflow
DAGs
Data Profiling
Browse
Admin
Docs
11:59 UTC
Show entries
Search:
DAG
Schedule
Owner
Recent Statuses
Links
example_bash_operator
0 0 * * *
airflow
example_branch_dop_operator_v3
*/1 * * * *
airflow
example_branch_operator
@daily
airflow
example_http_operator
1 day, 0:00:00
airflow
example_passing_params_via_test_command
*/1 * * * *
airflow
example_python_operator
None
airflow
example_short_circuit_operator
1 day, 0:00:00
airflow
example_skip_dag
1 day, 0:00:00
airflow
example_subdag_operator
@once
airflow
example_trigger_controller_dag
@once
airflow
example_trigger_target_dag
None
airflow
example_twitter_dag
@daily
Ekhtiar
example_xcom
@once
airflow
tutorial
1 day, 0:00:00
airflow

Airflow Code – Concepts

- **Operators** enable the creation of certain types of tasks. All operators derive from BaseOperator and inherit many attributes and methods.
- Examples of Operators:
  - BashOperator, PythonOperator, EmailOperator
  - See link for more details: https://pythonhosted.org/airflow/code.html#operators
- **Executors** are the mechanism by which tasks get run. All executors inherit from the BaseExecutor class.
- Examples of Executors:
  - LocalExecutor, SequentialExecutor, CeleryExecutor
  - See link for more details: https://pythonhosted.org/airflow/code.html#executors

Airflow Code : Simple Task Creation

- Here we create 2 tasks - Task_A, Task_B with Task_B dependent on Task_A :

```
import airflow.models as afm
from airflow import DAG
mydag = DAG(...)
task_a = afm.PythonOperator(task_id = 'Task_A',
                            python_callable = module.func_a,
                            dag = mydag)

task_b = afm.PythonOperator(task_id = 'Task_B',
                           python_callable = module.func_b,
                           dag = mydag)

task_b.set_upstream(task_a)
```

Airflow Code : Dynamic Task Creation

```
for fname in files:
  task = afm.PythonOperator(task_id='proc_%s' % fname,
                            python_callable = process,
                            op_kwargs={'fname' : filename},
                            dag=DAG)

  task.set_upstream(task_a)
```

Airflow Advantages

- Comprehensive workflow management solution
- Programmatic authoring of pipelines - all dependencies expressed in code rather than config files such as in Oozie (XML).
- Extremely feature rich CLI and web UI.
- Can express complex dependencies in pipelines - e.g. sub DAGs can be expressed within DAGs.

Airflow Disadvantages

- Architectural complexity – lots of moving pieces – metadata DB, scheduler, queue
- Code complexity – not as simple to use as say Luigi.

AWS Data Pipeline

- Web-service based workflow manager for managing data pipelines created via Amazon Web Services AWS
- Similar to SSIS rather than Python based workflow engine like Luigi, Airflow
- Requires SNS endpoint
- Basically ETL process hosted on Amazon's Cloud
- Emphasis is more on UI based pipeline creation and management rather than a programmatic approach.

AWS Data Pipeline UI

Create new pipeline | Actions

Filter: All | Filter pipelines ... | 9 pipelines (all loaded)

Pipeline ID	Name	Schedule State	Health Status
df-080527230QFGY4KPNMYK	JobFlow	PENDING	Pipeline is not active
df-06033452OZYFGDKQ3ZGX	ScheduleTest	PENDING	Pipeline is not active
df-03337934JOSA5ROTPKA	CopyMySQL	PENDING	Pipeline is not active
df-00189603TB4MZO0AD74D	CopyRedshift	PENDING	Pipeline is not active
df-0418261LXLUBQEFZ7FX	CopyDataTutorial	PENDING	Pipeline is not active
df-07356562IVEIU7LH9TQG	ApacheWebLogs	PENDING	Pipeline is not active
df-0870198233ZYV7H6T7CH	CrossRegionDDB	PENDING	Pipeline is not active
df-0116154RLHIY7WC387T	DDBPart2	FINISHED Runs every 1 day	HEALTHY
df-09028963KNVMR1DS8042	ImportDDB	FINISHED Runs every 1 day	HEALTHY

AWS Data Pipeline Creation

- Various ways to create a pipeline:
  - **AWS Management Console -** web interface to manage AWS Data Pipeline.
  - **AWS Command Line Interface (AWS CLI)** - with a pipeline definition file in JSON format.
  - **AWS SDKs** — Provides language-specific APIs.
    In Python's case you can use the boto3 module.
    See:
    https://boto3.readthedocs.io/en/latest/reference/services/datapipeline.html
  - **Query API**— Provides low-level APIs that you call using HTTPS requests. Using the Query API is the most direct way to access AWS Data Pipeline

AWS Pipeline Definition


```
{
  "objects": [
    {
      "id": "CSVId1",
      "name": "DefaultCSV1",
      "type": "CSV"
    },
    {
      "id": "RedshiftDatabaseId1",
      "databaseName": "dbname",
      "username": "user",
      "name": "DefaultRedshiftDatabase1",
      "*password": "password",
      "type": "RedshiftDatabase",
      "clusterId": "redshiftclusterId"
    },
    {
      "id": "Default",
      "scheduleType": "timeseries",
      "failureAndRerunMode": "CASCADE",
      "name": "Default",
      "role": "DataPipelineDefaultRole",
      "resourceRole": "DataPipelineDefaultResourceRole"
    },
    {
      "id": "RedshiftDataNodeId1",
      "schedule": {
        "ref": "ScheduleId1"
      },
      "tableName": "orders",
      "name": "DefaultRedshiftDataNode1",
      "createTableSql": "create table StructuredLogs (requestBeginTime CHAR(
      "type": "RedshiftDataNode",
      "database": {
        "ref": "RedshiftDatabaseId1"
      }
```

Advantages

- Seamless integration with AWS Services - Amazon S3, Amazon RDS, HDFS (AWS EMR), Redshift
- Easy to use web UI for creating and monitoring pipelines.
- (To non-programmers) – emphasis is on configuration-based pipeline specification.
- No need to install any additional software for use in the AWS cloud environment unlike tools like Luigi, Airflow.

Limitations

- Very tight coupling with AWS Services. Essentially vendor lock-in.
- Requires paying for services you may not want to use S3, Dynamo RDS etc.
- Not very programmatic - more configuration file-based. Programmability via Python was limited. Cannot specify the dependencies in Python code like you do in Luigi/Airflow.

Launching Pipelines in the cloud - AWS

- For Luigi, Airflow :
  - Launch EC2 instance
  - Install Python – via Anaconda
  - `pip install luigi`
  - `pip install airflow`
- For AWS Data Pipeline :

Luigi vs. Airflow vs. AWS Data Pipeline

Metric	Luigi	Airflow	AWS Data Pipeline
Ease of use	Easy	Moderate	It depends
Programmability	Medium	High	Low
Complexity	Low	Moderate	Moderate
Cloud provider coupling	None	None	High
Cost	Low	Low	Low
Developer learning curve	Low	Moderate	Moderate to High

Conclusions

- Use Luigi if:
  - You have relatively simple workflows.
  - You want a minimalist workflow management solution which is programmatic.
  - You want a tried and tested mature solution which has a vast knowledge base of solutions.

Conclusions

- Use Airflow if:
  - You would like an all in one solution.
  - You have complicated workflows with many dependencies.
  - You prefer encoding your task dependencies in code.

Conclusions

- Use AWS if :
  - You plan on deploying all your workflows within only AWS.
  - You don't mind a configuration/UI based pipeline definition
  - You would like an all in one solution that seamlessly integrates with AWS services.

References

- **Luigi**
  - https://luigi.readthedocs.io/en/latest/ - Read the docs
  - https://github.com/spotify/luigi - Github repo
- **Airflow**
  - https://pythonhosted.org/airflow/start.html - Documentation
  - https://github.com/apache/incubator-airflow - Github repo
- **AWS Data Pipeline**
  - https://aws.amazon.com/documentation/data-pipeline/ - Documentation
  - https://github.com/awslabs/data-pipeline-samples - Data Pipeline Samples