Create component oriented data science pipelines using CLAIMED, Elyra, KubeFlow Pipelines, MLX and Kubernetes

Join us for a paradigm shift in developing data science applications.

Links:
https://elyra.readthedocs.io/
https://github.com/elyra-ai/component-library
https://kubeflow.org/

A lot of state of the art data science applications are developed using jupyter notebooks by sequentially copy and pasting code blocks from the internet.

Then, usually, they either land up in the huge pot of projects which never make it into production (for reasons we will cover in this document) or they are “hardened” by moving the code from jupyter to scripts which then are containerized (e.g. using Docker) and run in a micro services architecture (e.g. on Kubernetes or virtual machines).

Assets not making it into production are dismissed mostly because of a missing path to production. As mentioned above, jupyter notebooks need to be hardened to be considered safe for production. This disconnects the original notebook with the hardened (e.g. containerized) version which aggravates maintenance.

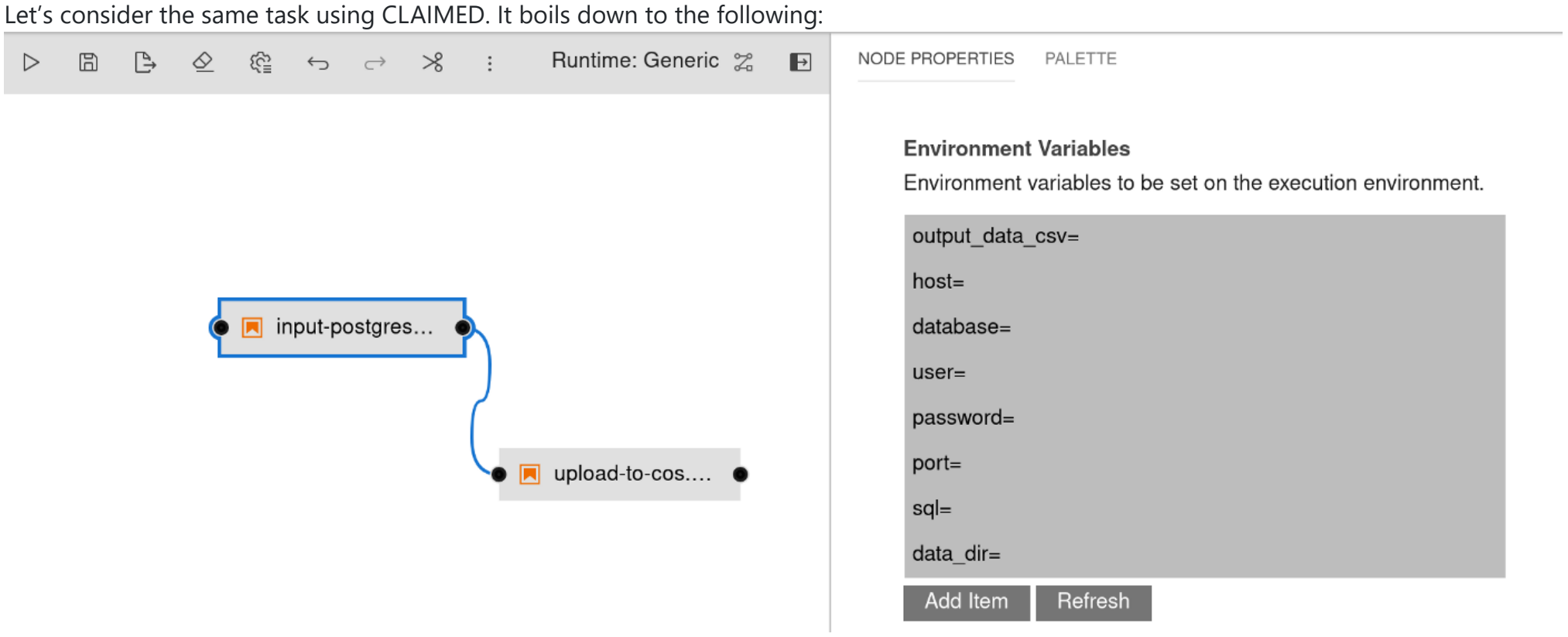
As data science applications are a sequence of code snippets anyway, what would happen if those snippets are hardened and containerized before and their sequential execution orchestrated by a workflow engine?

Welcome to CLAIMED — the component library for AI, Machine Learning, ETL and Data Science.

CLAIMED is a library of jupyter notebooks encapsulating hardened code for granular tasks. CLAIMED introduces a new, higher level of abstraction. It moves up from function calls to components.

Let’s consider a small example task of running a SQL statement against a PostgreSQL database and storing the result to a CSV file. If a data scientist doesn’t know this code by heart, most probably he would consult a search engine and search for something like “python connect postgresql” and “store postgresql resultset to csv”. After a while he would have identified the relevant code snippets (e.g. from stackoverflow.com), pasted them to a jupyter notebook and after some adjustments the code should be working in about 5–10 minutes.

Let’s consider the same task using CLAIMED. It boils down to the following:



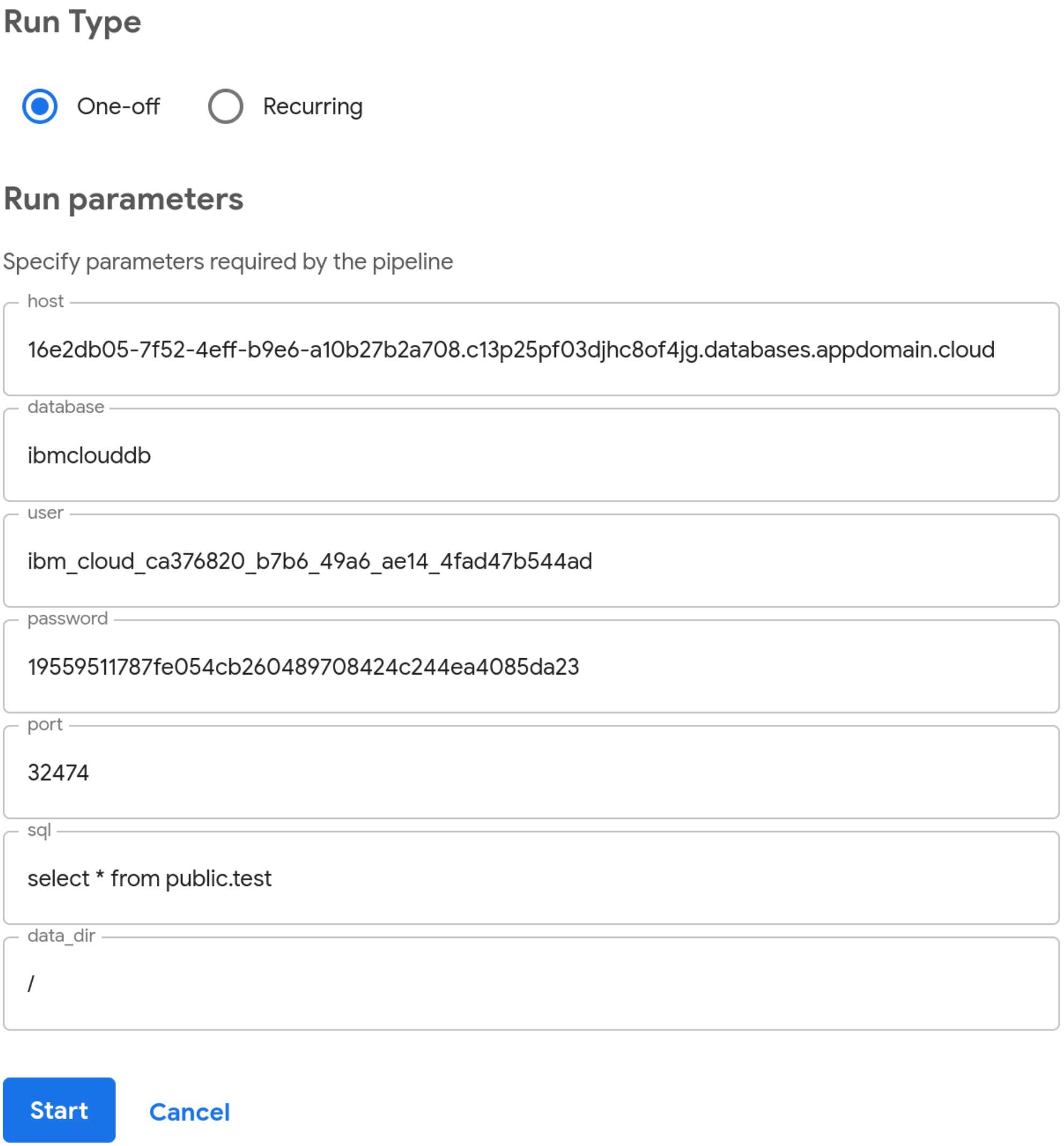
What you are seeing here is the Elyra Pipeline Editor where two notebooks have been dragged and dropped to the canvas. The first one reads data from PostgreSQL, the second writes it to Cloud Object Store. Each notebook is a so called “component” for CLAIMED. You can see that eight parameters have to be specified. That’s all. No searching, no coding, no bug fixing. Elyra — on the fly — transpiles to either AirFlow or KubeFlow pipelines out of the box. That way, reproducible and scalability are built-in.

Elyra is a set of AI centric extensions for jupyter lab but the pipeline editor is also supported on VSCode.

Now imagine, for some reason, you don’t wanna use Elyra and visual editing. No problem, since all CLAIMED components are notebooks, you can just call them — they are designed to accept configuration parameters as command line arguments:

*ipython ./claimed/component-library/input/input-postgresql.ipynb output_data_csv="data.csv" host="16e2db05–7f52–4eff-b9e6-a10b27b2a708.c13p25pf03djhc8of4jg.databases.appdomain.cloud" database="ibmclouddb" user="ibm_cloud_ca376820_b7b6_49a6_ae14_4fad47b544ad" password="19559511787fe054cb260489708424c244ea4085" port="32474" sql="select * from public.test" data_dir="/tmp/data/"*

As all meta information is self-contained in the notebook using python syntax, every data scientist can read, understand and modify the notebooks / components. In addition, the meta data can be used to transpile a notebook into a component for a workflow engine like KubeFlow, AirFlow, Galaxy, Apache Nifi or similar. This way, with python skills only, such components can be created.



The screenshot above shows the job launcher UI of Kubeflow. You can clearly see that the parameters to the component match the one from the jupyter notebook. And yes, no manual coding was necessary to turn the notebook into a KubeFlow Pipeline component. A tool called C3 (CLAIMED Component Compiler) does the job for you.

So how does such a notebook look like? First of all, it is an ordinary notebook, no cell magics or cell metadata is required to keep things simple and understandable by data scientists. It is based on design by contract principles — so let's first have a look at a notebook diagram first:

Jupyter notebook

Title <<markdown>>

Description <<markdown>>

Requirements (pip install...)

imports

Input and Output parameters

1st code/markdown cell

▪ ▪ ▪

(n-1)th code/markdown cell

nth code/markdown cell

As we can see, there are a couple of cells at the beginning of the notebook with a specific purpose. So the order and the actual position of the cell matters. The cell positions, order and specific purpose are as follows:

Cell 1

Title / name of the component (e.g. input-postgresql) (as markdown cell type)

Cell 2

Description of the component (as markdown cell type)

Cell 3

Requirements of the component — this cell need to contain code for installing requirements via pip or conda

Cell 4

All imports needed for execution of this notebook

Cell 5

Input / Output parameters — this cell type is the most complex as it defines all variables the component needs during runtime. So in case of the PostgreSQL connector the database host, user name and password for examples

Cell 6+

All subsequent cells are freely usable by the creator of the component. The only thing one have to remember if the component is not a Sink (last processing step in a pipeline) it needs to pass on data to the next component. This is called the “components output”. The name and type of the component’s output is also defined in Cell 4 (by having it’s environment variable name start with “output_”), so the only thing which needs to be done is to write the result to a file as specified in Cell 4 after all processing is done.

To further illustrate this concept let’s have a look at a real example notebook component of CLAIMED — again we are using the PostgreSQL connector component:

Input Postgresql Cell 1

Cell 2

This notebook pulls data from a postgresql database as CSV on a given SQL statement

[ ]:
```
!pip install psycopg2-binary==2.9.1 pandas==1.3.1
```
Cell 3

[ ]:
```
import os
import pandas as pd
import psycopg2
import re
import sys
```
Cell 4

[ ]:
```
# path and file name for output
data_csv = os.environ.get('output_data_csv', 'data.csv')

# hostname of database server
host = os.environ.get('host')

# database name
database = os.environ.get('database')

# db user
user = os.environ.get('user')

# db password
password = os.environ.get('password')

# db port
port = int(os.environ.get('port', 5432))

# sql query statement to be executed
sql = os.environ.get('sql')

# temporal data storage for local execution
data dir = os.environ.get('data dir', '../../data/')
```
Cell 5

Let's dive into the different cells again:

Cell 1

Here you can see the title of the component: "Input PostgreSQL" in markdown.

Cell 2

Here you can see the description of the component in markdown.

Cell 3

The '!' symbol turns this command into a shell command which installs the requirements of this notebook using "pip"

Cell 4

All required imports

Cell 5

This is the most important cell as it exposes the interface to the component by defining it's parameters. This configuration looks like pure python code and it actually is. Using design by contract the following rules are used to define different aspects of the configuration.

Let's consider the following block:
```
# hostname of database server
host = os.environ.get('host')
```
This code already contains a lot of information. The comment line "# hostname of database server" is used as description / help text. As the environment variable "host" is read we know that the component expects "host" of type "String" as configuration parameter. This is a mandatory parameter, now we see how an optional parameter is configured:

```python
# db port
port = int(os.environ.get('port', 5432))
```

Again, the description and parameter name are extracted as before. As the whole expression gets cast to “int” we know that parameter “port” is of type “Integer”. Finally, as “5432” is given as default value (in case environment variable “port” isn’t set) this specifies an optional parameter.

The following shows how an output parameter is defined by having the environment variable name starting with “output_” — that’s all:

```python
# path and file name for output
output_data_csv = os.environ.get('output_data_csv', 'data.csv')
```

Let’s finally have a look at C3 (CLAIMED Component Compiler) — which takes such notebooks and turns them into runtime specific components. Currently, only KubeFlow is supported, but support for Apache Airflow, Apache Nifi, NodeRED and Galaxy are on the roadmap already.

The following yaml file is a so called KubeFlow Pipeline component specification which has been created by C3 from the notebook above:

```yaml
name: Input Postgresql
description: This notebook pulls data from a postgresql database as CSV on a given SQL statement

inputs:
- {name: host, type: String, description: 'hostname of database server'}
- {name: database, type: String, description: 'database name'}
- {name: user, type: String, description: 'db user'}
- {name: password, type: String, description: 'db password'}
- {name: port, type: Integer, description: 'db port'}
- {name: sql, type: String, description: 'sql query statement to be executed'}
- {name: data_dir, type: String, description: 'temporal data storage for local execution'}


outputs:
- {name: output_data_csv, type: String, description: 'path and file name for output'}


implementation:
    container:
        image: continuumio/anaconda3:2020.07
        command:
        - sh
        - -ec
        - |
          mkdir -p `echo $0 |sed -e 's/\/[a-zA-Z0-9]*$//'`
          wget https://raw.githubusercontent.com/IBM/claimed/master/component-library/input/input-postgresql.ipynb
          ipython ./input-postgresql.ipynb output_data_csv="$0" host="$1" database="$2" user="$3" password="$4" port="$5" sql="$6"
        - {outputPath: output_data_csv}
        - {inputValue: host}
        - {inputValue: database}
        - {inputValue: user}
        - {inputValue: password}
        - {inputValue: port}
        - {inputValue: sql}
        - {inputValue: data_dir}
```

Understanding this yaml file is beyond the scope of this reading but you can clearly see the power of the concept of CLAIMED and the technology it’s relying on. This way, scalable, robust and reproducible data science applications can be build using the tools data scientists like in a way they are ready for production at any time.