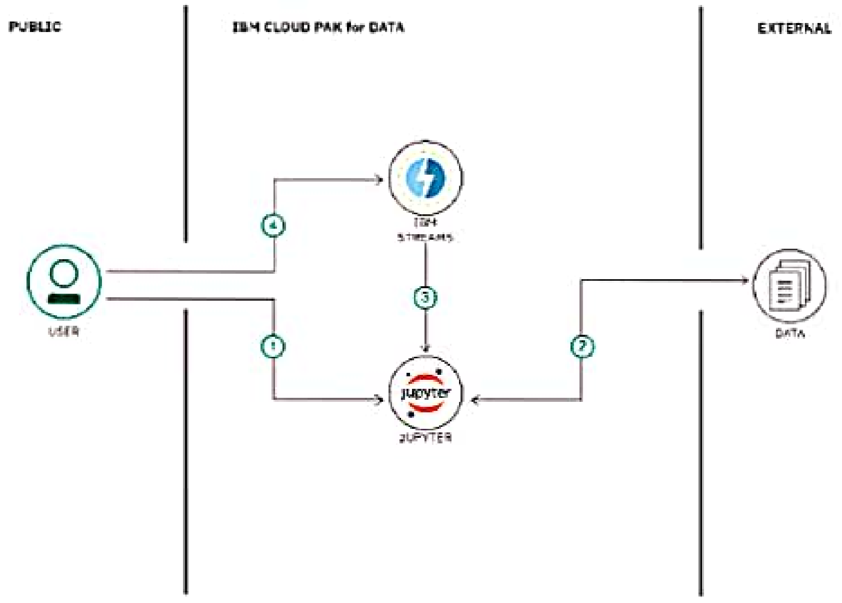
The Python API streamsx allows you to build streaming apps that use IBM Streams, a service that runs on IBM Cloud Pak for Data. The IBM Cloud Pak for Data platform provides additional support, such as integration with multiple data sources, built-in analytics, Jupyter Notebooks, and machine learning. Scalability is increased by distributing processes across multiple computing resources.

In this code pattern, we will build a streaming application by creating a Jupyter Notebook using the streamsx Python API. The app will process a stream of data containing mouse-click events from users as they browse a shopping website.

Flow



1. User runs Jupyter Notebook in IBM Cloud Pak for Data.
2. Clickstream data is inserted into streaming app.
3. Streaming app using the streamsx Python API is executed in the IBM Streams service.
4. User accesses IBM Streams service job to view events.

Instructions

Find the detailed steps for this pattern in the README file. The steps will show you how to:

1. Clone the repo.
2. Add IBM Streams service to IBM Cloud Pak for Data.
3. Create a new project in IBM Cloud Pak for Data.
4. Add a data asset to your project.
5. Add a notebook to your project.
6. Run the notebook.
7. View job status in IBM Streams service panel.
8. Cancel the job.

Python image

Pipelines are executed by using containerized images, often public, that contain the necessary tools to complete their tasks. Unfortunately, Python tools often are not available in these standard public images, even if they can be collected and loaded by the means of a script that tailors each pipeline stage.

A file called `.pipeline-config.yaml` contains several sections, one for each stage, where I can write the aforementioned script as instructions to load tools and complete the Python stack.

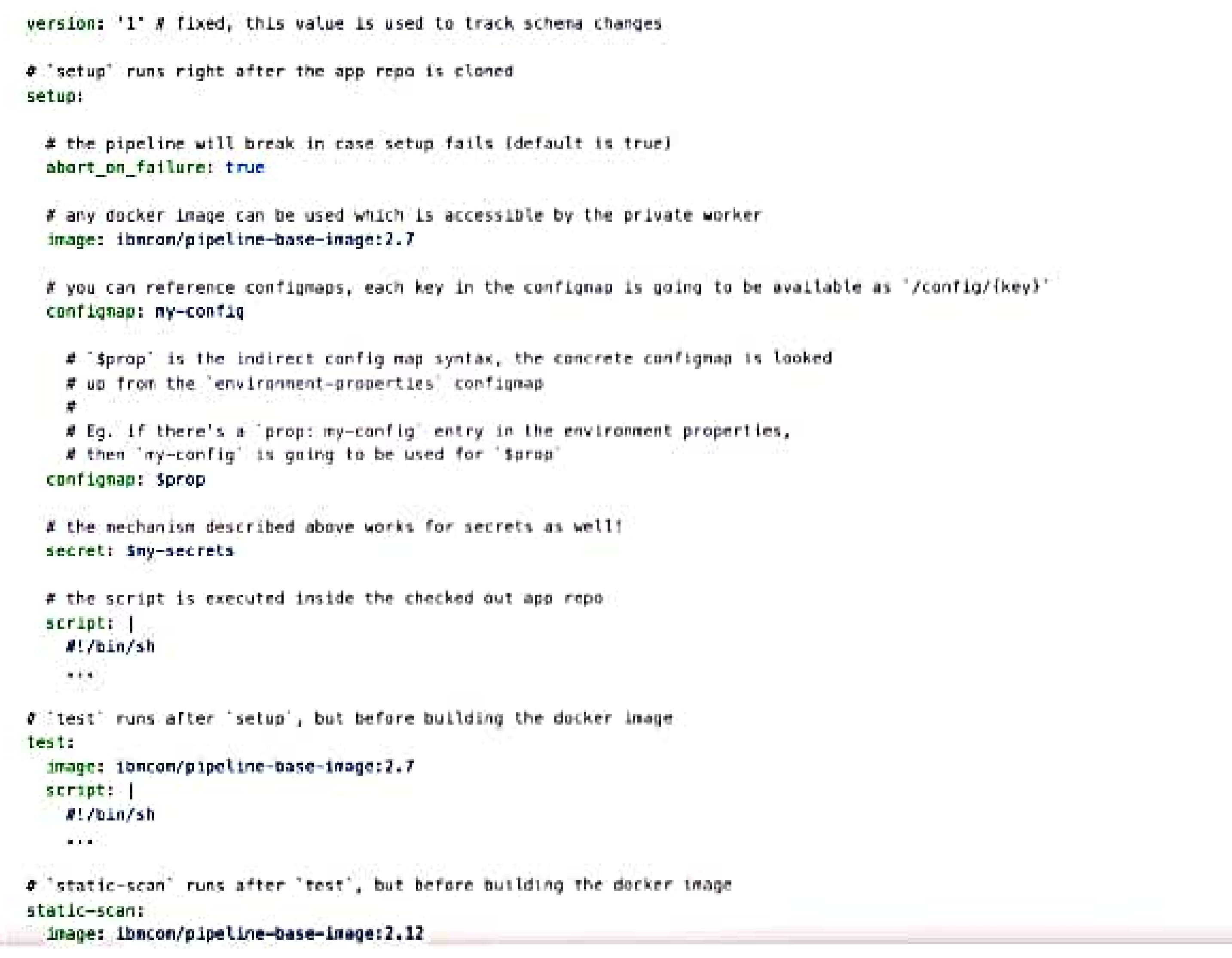
```
FROM ubuntu:20.04

RUN apt update && apt install software
RUN add-apt-repository ppa:deadsnakes/
RUN apt install -y python3 python3-pip
RUN apt-get update && apt-get install
RUN ln -s /usr/bin/python3 /usr/bin/py

RUN useradd -m compliance && echo "com
RUN mkdir -p /home/compliance
```

Show more

If this management is too onerous, you can modify any execution stage through the YAML configuration file. The YAML configuration file usage is described in detail within the [IBM Cloud product documentation](). Hereafter, it's appearing at the initial state.



```yaml
version: '1' # fixed, this value is used to track schema changes

# 'setup' runs right after the app repo is cloned
setup:

  # the pipeline will break in case setup fails (default is true)
  abort_on_failure: true

  # any docker image can be used which is accessible by the private worker
  image: ibmcom/pipeline-base-image:2.7

  # you can reference configmaps, each key in the configmap is going to be available as '/config/{key}'
  configmap: my-config

    # `$prop` is the indirect config map syntax, the concrete configmap is looked
    # up from the `environment-properties` configmap
    #
    # Eg. if there's a `prop: my-config` entry in the environment properties,
    # then `my-config` is going to be used for `$prop`
  configmap: $prop

  # the mechanism described above works for secrets as well!
  secret: $my-secrets

  # the script is executed inside the checked out app repo
  script: |
    #!/bin/sh
    ...

# `test` runs after `setup`, but before building the docker image
test:
  image: ibmcom/pipeline-base-image:2.7
  script: |
    #!/bin/sh
    ...

# `static-scan` runs after `test`, but before building the docker image
static-scan:
  image: ibmcom/pipeline-base-image:2.12
```

```
# devrequirements.txt
flake8>=3.7.0
flake8-2020>=1.3.0
flake8-broken-line>=0.1.1
flake8-bugbear>=19.8.0
flake8-builtins>=1.4.1
flake8-commas>=2.0.0
flake8-comprehensions>=2.2.0
flake8-docstrings>=1.4.0
flake8-eradicate>=0.2.1
flake8-import-order>=0.0.11
flake8-mutable>=1.2.0
flake8-pep3101>=1.2.1
flake8-print>=3.1.0
flake8-quotes>=2.1.0
flake8-string-format>=0.2.3
flake8-use-fstring>=1.0
```

```
static-scan:
  image: icr.io/continuous-delivery/
  abort_on_failure: true
  script: |
    echo '===static scan==='
    cd /workspace/app/$app_name

    pip3 install -r devrequirements.tx
    pip3 list
    flake8 *
    bandit -r *
```

Conclusion

Even if a Python package does not need to continuously maintain a running service and its code might be more gradually developed, you should always look for the highest quality possible. Thus, all of the checks are important to build high-quality software and can help you to deliver packages with zero defects.

A pipeline forces all involved developers to pay attention to their interaction, to create a clean commit history, and to maintain a clear vision of the software design and architecture. Moreover, the focus of different tests, such as unit, systems, and finally acceptance, can drive development to quickly clarify which ones are planned features and enhancements. This prevents the writing of out-of-scope procedures and makes following agile techniques easier.