Beam in production: Working with dataflow flex templates and cloud build

By Ragy Abraham, Monita

BEAM SUMMIT Austin, 2022

Little bit about me...

Live in Sydney, Australia

monita

Cofounder Monita
getmonita.io

Little bit about me...

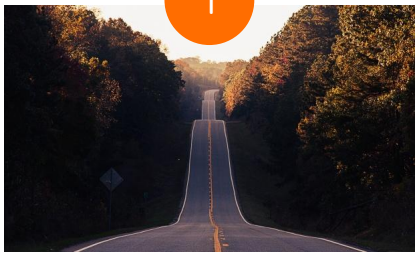
Live in Sydney, Australia

PARTNERS

THE APACHE SOFTWARE FOUNDATION
AKVELON We make BIG software happen
Packt
MANNING
monita
AICamp
Data Council
WOMEN WHO CODE
ChickTECH

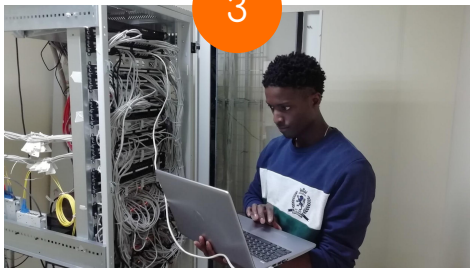
Cofounder Monita
getmonita.io

What we'll cover today...



Our Journey with Beam
Learnings from startup experience

Ensuring reliable pipelines
Continuous Integration



Deploying Beam across multiple-environments
Continuous Deployment

Bring it together
Deploying Word-count App in CICD framework

1. Our Journey with Beam

Learnings from startup experience

BEAM SUMMIT Austin, 2022

Our journey with beam...

- As previously mentioned, using beam since 2020
  - Initially beam solved a huge problem for us -> How do we scalability process a large number of request in real-time whilst ensuring accuracy and completeness
  - At first it was hacky!
    - Mainly concerned with getting it working
      - Lots of manual steps to deploy
      - Testing was only done locally – no automation
  - As we matured and our clients' became bigger (more enterprisee) hacky no longer cut the mustard
  - We need to develop processes to ensure updates to our pipelines were delivered, to the correct env, promptly (in an automated fashion) and with limited bugs (bugs are an inevitable reality, but we want to do our best to reduce them)

We started digging...

- So we started looking into Dataflow Flex Templates and `TestPipeline` framework
  - We had been using Google Cloud Build as our CICD framework for sometime
    - For our web app and backend applications
  - However, due to beam's unique programming paradigm, it wasn't a trivial task to get CICD running for our beam pipelines
    - We went through a considerable amount of pain to get this up so we thought we'd save you all the headache and share our learnings with you

BEAM SUMMIT

Python SDK!

2. Ensuring reliable pipelines

Continuous Integration

BEAM SUMMIT Austin, 2022

Beam TestPipelines

Why we need it...

Beam provides a comprehensive testing framework as part of their SDK

Apache Beam Testing Framework

```
from apache_beam.io.gcp.tests.pubsub_matcher import PubSubMessageMatcher
from apache_beam.runners.runner import PipelineState
from apache_beam.testing import test_utils
from apache_beam.testing.pipeline_verifiers import PipelineStateMatcher
from apache_beam.testing.test_pipeline import TestPipeline
from apache_beam.testing.util import assert_that
from apache_beam.testing.util import equal_to
```

What is TestPipeline

- You can test the individual functions used in your pipeline.
  - User defined **`DoFns`**
- You can test an entire Transform as a unit
  - Combination of several **`DoFns`**
- You can perform an end-to-end test for an entire pipeline.
  - The entire pipeline including I/O
    - For batch processing this is straightforward
    - For streaming, as with everything streaming, it's a bit more complicated

Code Deep Dive - Testing

BEAM SUMMIT Austin, 2022

Unit Testing - Basic

```
2 hours ago | 1 author (You)
lass CountTest(unittest.TestCase):

  def test_count(self):
      # Our static input data, which will make up the initial PCollection.
      WORDS = [
          "hi", "there", "hi", "hi", "sue", "bob",
          "hi", "sue", "", "", "ZOW", "bob", ""
      ]
      # Create a test pipeline.
      with TestPipeline() as p:

          # Create an input PCollection.
          input = p | beam.Create(WORDS)

          # Apply the Count transform under test.
          output = input | beam.combiners.Count.PerElement()

          # Assert on the results.
          assert_that(
              output,
              equal_to([
                  ("hi", 4),
                  ("there", 1),
                  ("sue", 2),
                  ("bob", 2),
                  ("", 3),
                  ("ZOW", 1)]))
```

```
TERMINAL   DEBUG CONSOLE   GITLENS: VISUAL FILE HISTORY   JUPYTER: VARIABLES   COMMENTS

apache-beam-cicd/src on  DEPLOY via  v3.9.12 (env) on  ragy@rna.digital took 5s
❯ pytest tests/test_basic.py --disable-warnings
============================= test session starts ==============================
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/14385898/Documents/RNA/Code/apache-beam-cicd/src
plugins: anyio-3.6.1
collected 1 item

tests/test_basic.py .

============================== 1 passed, 20 warnings in 2.05s ==============================
```

Unit Testing - DoFns (Composite)

```
m.ptransform_fn
ef CountWords(pcoll):
    return (
        pcoll
        | 'ExtractWords' >> beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x))
        | beam.combiners.Count.PerElement()
    )


You, 8 hours ago | 1 author (You)
class WordCountTest(unittest.TestCase):

    def test_count_words(self):
        # Our input data, which will make up the initial PCollection.
        WORDS = [
            "hi", "there", "hi", "hi", "sue", "bob",
            "hi", "sue", "", "", "ZOW", "bob", ""
        ]
        # Our output data, which is the expected data that the final PCollection must match.
        EXPECTED_COUNTS = [
            ('hi', 4), ('there', 1),
            ('sue', 2), ('bob', 2), ('ZOW', 1)]
        with TestPipeline() as p:
            input = p | beam.Create(WORDS)
            output = input | CountWords()
            assert_that(output, equal_to(EXPECTED_COUNTS), label='CheckOutput')

    # The pipeline will run and verify the results.


if __name__ == '__main__':
    logging.getLogger().setLevel(logging.INFO)
    unittest.main()
```

```
TERMINAL   DEBUG CONSOLE   GITLENS: VISUAL FILE HISTORY   JUPYTER: VARIABLES   COMMENTS

apache-beam-cicd/src on  DEPLOY via  v3.9.12 (env) on  ragy@rna.digital
❯ pytest tests/test_composite_transform.py --disable-warnings
============================================================================ test session starts =========
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/14385898/Documents/RNA/Code/apache-beam-cicd/src
plugins: anyio-3.6.1
collected 1 item

tests/test_composite_transform.py .

============================================================================ 1 passed, 20 warnings in 1.98s ==
```

Integration Testing - Pipelines (Batch)

```
2 days ago | 1 author (You)
class WordCountTest(unittest.TestCase):

  SAMPLE_TEXT = "beam summit 2022"

  def create_temp_file(self, contents):
      with tempfile.NamedTemporaryFile(delete=False) as f:
          f.write(contents.encode('utf-8'))
          return f.name

  def test_basics(self):
      import wordcount
      temp_path = self.create_temp_file(self.SAMPLE_TEXT)
      expected_words = collections.defaultdict(int)
      for word in re.findall(r'[\w]+', self.SAMPLE_TEXT):
          expected_words[word] += 1
      wordcount.run(
          [
              '--input=%s*' % temp_path,
              '--output=%s.result' % temp_path
          ]
      )

      print("==runs after pipeline==")
      # Parse result file and compare.
      results = []
      with open_shards(temp_path + '.result-*') as result_file:
          for line in result_file:
              match = re.search(r'(\S+),([0-9]+)', line)
              if match is not None:
                  results.append((match.group(1), int(match.group(2))))
              elif line.strip():
                  self.assertEqual(line.strip(), 'word,count')
      self.assertEqual(sorted(results), sorted(expected_words.items()))
```

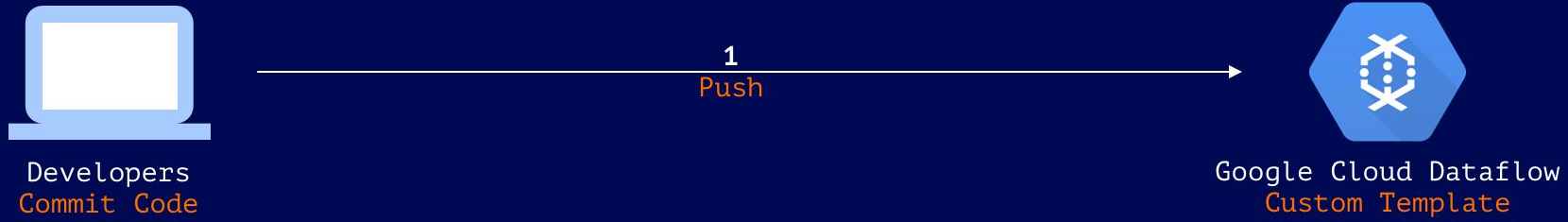
```
48
TERMINAL   DEBUG CONSOLE   GITLENS: VISUAL FILE HISTORY   JUPYTER: VARIABLES   COMMENTS

apache-beam-cicd/src/tests on  DEPLOY [!] via  v3.9.12 (env) on  ragy@rna.digital
❯ pytest test_wordcount_it.py --disable-warnings
============================= test session starts ==========
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/14385898/Documents/RNA/Code/apache-beam-cicd/src
plugins: anyio-3.6.1
collected 1 item

test_wordcount_it.py .

============================== 1 passed, 20 warnings in 2.48s ===
```

Dataflow Flex Templates

Why we need it...

Flex templates are a way to package and execute custom pipelines in Dataflow

USUAL WORKFLOW



- ❌ No Automation via CICD
- ❌ Only local testing
- ❌ Must have access to code base
- ❌ Local environment must be set up with Dataflow dependencies

What is a flex-template...

- A template is a convenient way to package and distribute beam pipelines
- A flex template is a user-defined template based on user custom code
  - This code is then templated and staged in GCS ready to be launched
- Templating has 2 phases:

1. Construction:
   a. Implementing the pipeline and compiling it into execution graph and staging it in GCS
2. Execution:
   a. Executing the pipeline: this is the only step you would need to do in the GC to get a template up and running
      i. Note: running the pipeline does not require recompilation of code
      ii. Can be done in a number of ways
         1. Google Cloud console, Google Cloud CLI, REST API or **Cloud Build commands**

Why flex-template are important...

- You can run your pipelines without the development environment and associated dependencies
- Templates separate the pipeline construction (performed by developers) from the running of the pipeline. Hence, there's no need to recompile the code every time the pipeline is run.
- Non-technical users can run templates with the Google Cloud console, Google Cloud CLI, or the REST API.

How a flex-template works...

1. Package a user defined pipeline as a Docker image
2. Stage the image on your project's container registry
3. Create spec.json file -> template specification file stored on GCS
4. The spec.json file can then be used to launch the pipeline on DF

3. Deploying Beam across multiple environments

Continuous Deployment
Code deep dive

BEAM SUMMIT Austin, 2022

Rules...

- We do not want to store any of this information in GIT
- We do not want to rely on static .env files
  - .env files must be built dynamically and variable are specific to the environment
- We need all environment variables to made available by the environment
  - That way we don't have to think about "where is this being deployed"
- Sensitive information is stored in secret manager
  - E.g. database credentials

CICD WORKFLOW

Users
can launch
jobs from UI

Cloud Repository
Dataflow Source Code

1
Push

Developers
Commit Code

2
Trigger

Cloud Build
YAML File

3
Build Steps

docker

GCS
Template
Staging
spec.json

Google Container
Registry
Build + Store
Docker Image

Google Cloud Dataflow
Flex Template

4
Dataflow
Job is
live

env variables
used to access
secrets

credentials
are returned
securely

Cloud Secret
Manager
Holds Env Specific
Credentials

1. Create variables in cloud build

Variable 1 * _PROJECT_ID	Value 1 [redacted]
Variable 2 * _REGION	Value 2 us-central1
Variable 3 * _SETUP_FILE	Value 3 /dataflow/template/setup.py
Variable 4 * _TARGET_GCR_IMAGE	Value 4 [redacted]streaming-pipeline
Variable 5 * _TEMPLATE_GCS_LOCATION	Value 5 gs://[redacted]'dataflow/template/spec.json

```yaml
steps:
  - name: 'gcr.io/cloud-builders/docker'
    id: 'build-docker-container'
    entrypoint: 'bash'
    args:
    - '-c'
    - |
      docker build -t gcr.io/$PROJECT_ID/$_TARGET_GCR_IMAGE:$BUILD_ID \
      --build-arg PROJECT_ID=$PROJECT_ID \
      --build-arg IMAGE=$_TARGET_GCR_IMAGE \
      --build-arg REGION=$_REGION \
      --build-arg BUCKET=$_TEMPLATE_GCS_LOCATION .
```

2. Make variables available to Dockerfile

3. Build a dynamic .env file in docker

```
RUN echo "PROJECT_ID=${PROJECT_ID}" >> .env
RUN echo "IMAGE=${IMAGE}" >> .env
RUN echo "BUCKET=${BUCKET}" >> .env
RUN echo "REGION=${REGION}" >> .env
```

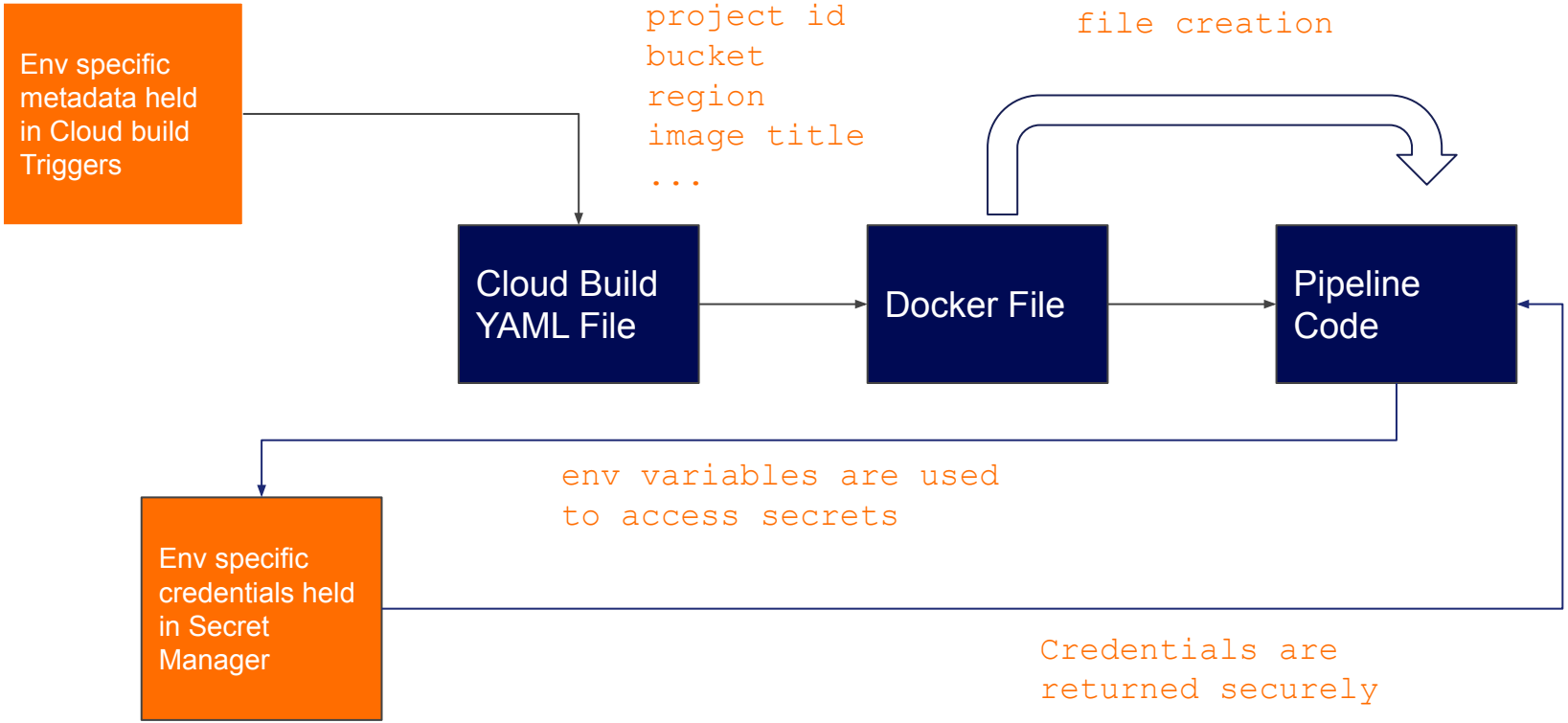
4. Import .env in code

```python
from pathlib import Path
from dotenv import load_dotenv
import os
import logging
import apache_beam as beam
import json
from traceback import format_exc
from apache_beam.options.pipeline_options import PipelineOptions, StandardOptions
from modules.cloud_secrets import access_secret, CreateSecret
import google.auth

env_path = Path('.') / '.env'
load_dotenv(dotenv_path=env_path)
# Load ENV variables
REGION = os.getenv('REGION')
BUCKET = os.getenv('BUCKET')
IMAGE = os.getenv('IMAGE')
LOCAL = os.getenv("LOCAL")
logging.info(f"Local enviornment is {LOCAL}")

if LOCAL == 'True':
    os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "./modules/keys/[redacted]
    credentials, PROJECT_ID = google.auth.default(
        scopes=["https://www.googleapis.com/auth/cloud-platform"]
    )
    runner = 'DirectRunner'
```

4.Bringing it together (Demo)

Live Demo Deploying CICD framework

The workflow we used

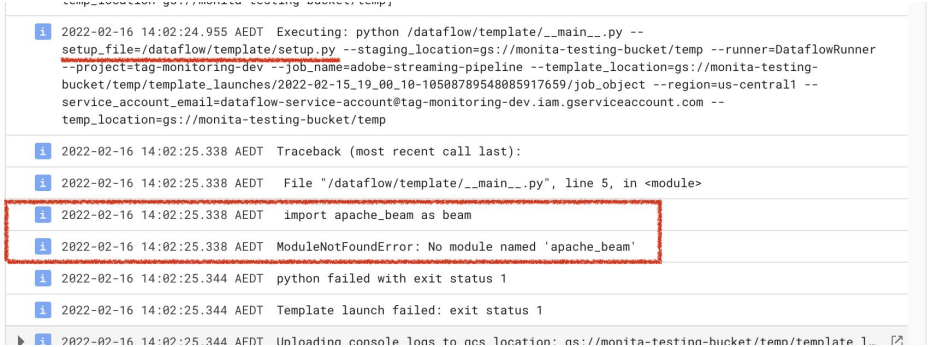


Gotchas...

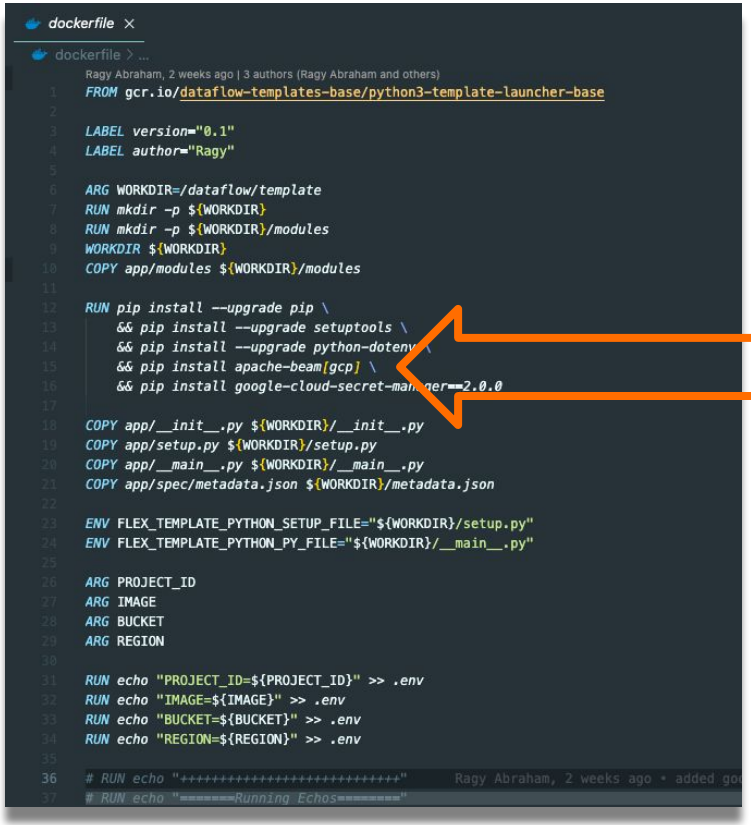
```
Timeout in polling result
file: gs://monita-testing-bucket/temp/template_launches/2022-01-31_20_26_06-14138723572289947670/operation_result.
Service account: dataflow-service-account@tag-monitoring-dev.iam.gserviceaccount.com Image
URL: gcr.io/tag-monitoring-dev/adobe-streaming-pipeline:e8354c20-3bd0-49fd-9c85-b1d8c6dfe78e Troubleshooting guide
at https://cloud.google.com/dataflow/docs/guides/common-errors#timeout-polling
```

Do NOT use a requirements.txt file

Do not install beam in setup.py file



Gotchas...



Beam must be installed in the Dockerfile

Questions?

ragy@getmonita.io
Linkedin
/ragyibrahim/apache-beam-cicd

BEAM SUMMIT Austin, 2022