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

By Ragy Abraham, Monita
Little bit about me...

Live in Sydney, Australia

Cofounder Monita
getmonita.io
Little bit about me...

Live in Sydney, Australia

Cofounder Monita
getmonita.io
What we’ll cover today...

1. Our Journey with Beam
   Learnings from startup experience

2. Ensuring reliable pipelines
   Continuous Integration

3. Deploying Beam across multiple-environments
   Continuous Deployment

4. Bring it together
   Deploying Word-count App in CICD framework

Austin, 2022
1. Our Journey with Beam

Learnings from startup experience
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 enterprise) 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
Python SDK!
2. Ensuring reliable pipelines
Beam Test Pipelines

Why we need it...

Beam provides a comprehensive testing framework as part of their SDK.
Apache Beam Testing Framework

```python
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 DoFs
● You can test an entire Transform as a unit
  ○ Combination of several DoFs
● 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
Unit Testing - Basic

```python
class CountTest(unittest.TestCase):
    def test_count(self):
        # Our static input data, which will make up the initial PCollection.
        WORDS = ['hi', 'there', 'hi', 'sue', 'bob',
                 'hi', 'sue', '', '', '20w', '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], ['20w', 1]])
        
```

**Summary:**

Example code defining a test case for a Count transform in a PCollection. The test verifies that the Count transform correctly counts the occurrences of each word. The input data contains various words, and the test asserts that the output matches the correct counts for each word.
Unit Testing - DoFs (Composite)

```python
# Example Python code for unit testing with DoFs

def test_count_words(self):
    # Our input data, which will make up the initial PCollection.
    WORDS = ['hi', 'there', '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 | beam.CountWords()
        assert_that(output, equal_to(EXPECTED_COUNTS), label='CheckOutput')

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

---

**Output:**

```
.../tests/test_composite_transform.py:89: DeprecationWarning: test_count_words has a docstring, and the WARNING_OUTPUT flag is enabled. This will be ignored when the flag is removed in a later release.

-terminal, -debug, -console, -git, -visual, -file, -history, -jupyter, -variables, -comments

tests/test_composite_transform.py

tests/test_composite_transform.py ...
```

---

**Notes:**

- The code snippet demonstrates how to use DoFs for unit testing in a Python script.
- It shows how to create a PCollection, apply transformations, and assert the expected output.
- The example includes a warning about deprecation, indicating that this approach will be removed in a future release.

---

**Author:**

Austin, 2022
Integration Testing - Pipelines (Batch)

```python
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'\d+(?:\d+)?', line)
                if match is not None:
                    results.append((match.group(1), int(match.group(2))));
            else:
                line.strip()
        self.assertEqual(line.strip(), 'word', count)
        self.assertEqual(sorted(results), sorted(expected_words.items()))
```

---

**TERMINAL**

```
DEPLOY [1] via v3.9.12 (env) on rga@rna.digital
pytest test_wordcount_it.py --disable-warnings
```

**CLIENT**

```
platform darwin -- Python 3.9.12, pytest-7.1.2, pluggy-1.0.0
plugins: anyio-3.6.1
collected 1 item
```

1 passed, 20 warnings in 2.48 s
Dataflow Flex Templates

Why we need it...

Flex templates are a way to package and execute custom pipelines in Dataflow
Anyone who wants to deploy/update pipeline must:

- Install beam locally
- Getting beam running locally
  - Install all dependencies locally
  - If your local environment is different this will impact end result
  - You are essentially deploying your local code to a remote runner -> as is!
- Launching pipelines is now limited to people who have access to code base

**USUAL WORKFLOW**

- Developers Commit Code
- Push
- Google Cloud Dataflow Custom Template

- 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
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
1. Create variables in cloud build

2. Make variables available to Dockerfile

3. Build a dynamic .env file in docker

4. Import .env in code
4. Bringing it together (Demo)

Live Demo Deploying CICD framework
The workflow we used

- Env specific metadata held in Cloud build Triggers

Cloud Build YAML File → Docker File → Pipeline Code

- project id
- bucket
- region
- image title

Dynamic .env file creation

- env variables are used to access secrets

Env specific credentials held in Secret Manager

- Credentials are returned securely

Austin, 2022
Gotchas...

Timeout in polling result
Service account: dataflow-service-account@tag-monitoring-dev.iam.gserviceaccount.com
Image URL: gcr.io/tag-monitoring-dev/adobe-streaming-pipeline:e8354c20-3bd9-49fd-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