KCD UK Edinburgh
22nd October, 2025

No YAML? No Problem: Orchestrate Kubernetes Workflows the Easy Way with Python

An introduction to Argo Workflows and Hera

Speaker

ELLIOT GUNTON

pipekit

Senior Software Engineer

Hera maintainer

About Pipekit

Pipekit helps you scale Argo & Kubernetes

Serving **startups & Fortune 500** companies since 2021:

- **Enterprise Support for Argo**
  → Ideal for Platform Eng teams scaling with Argo
- **Control Plane for Argo Workflows**
  → Ideal for data teams, granular RBAC, and multi-cluster architectures

Direct support from 40% of the active **Argo Workflows maintainers** in the world

Save engineering time and **up to 60%** on compute costs

Outline

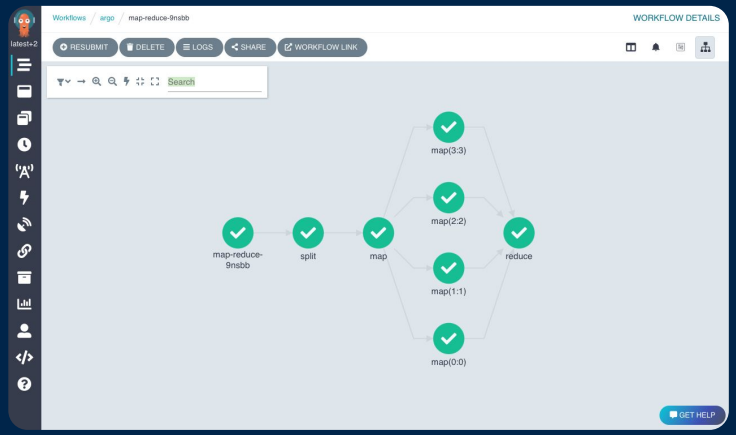
- What is Kubernetes?
- What is Argo Workflows?
- All things Hera
- Features and Examples
- Demo
- Takeaways

What is Kubernetes?

1. The cloud-native standard
2. Allows applications to scale more easily
3. Container-native
4. Active and growing community
5. Long-term support
6. Vendor-neutral

So What is Argo Workflows?

0. Workflow Orchestrator built on Kubernetes
1. The de facto Kubernetes Workflow Orchestration Standard
2. Allows workloads to scale more easily
3. Container-native
4. Active and growing community
5. Long-term support
6. Vendor-neutral



When Would You Need Workflow Orchestration?

- Machine Learning (re)training pipelines
- All kinds of data processing
- Scheduled/batch jobs

You need features like

- Scalable resources
- Automatic retries on failure
- Integrated artifact storage (S3 etc)

Anatomy of a Workflow

- Workflows are Kubernetes Custom Resource Definitions
- Workflows house a collection of “templates” which are analogous to functions in a library
- Templates can be arranged through a DAG
- The entrypoint acts like a “main” function

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
 generateName: dag-diamond-
spec:
 entrypoint: diamond
 templates:
 - name: diamond
   dag:
     tasks:
     - name: A
       template: echo
       arguments:
         parameters: [{name: message, value: A}]
     - depends: A
       name: B
       template: echo
       arguments:
         parameters: [{name: message, value: B}]
     - depends: A
       name: C
       template: echo
       arguments:
         parameters: [{name: message, value: C}]
     - depends: B && C
       name: D
       template: echo
       arguments:
         parameters: [{name: message, value: D}]
 - name: echo
   container:
     command: ["echo", "{{inputs.parameters.message}}"]
     image: alpine:3.7
   inputs:
     parameters:
     - name: message
```

The Average K8s/Argo Developer Experience

YAML is a barrier to entry

Hard to test

Hard to reuse

Hard to maintain long Workflows

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
 generateName: fine-tune-llm-
spec:
 entrypoint: fine-tune
 onExit: exit
 templates:
 - dag:
     tasks:
     - name: create-ssd-storage-class
       template: create-ssd-storage-class
     - depends: create-ssd-storage-class
       name: create-etcd-stateful-set
       template: create-etcd-stateful-set
     - depends: create-ssd-storage-class
       name: create-etcd-load-balancer
       template: create-etcd-load-balancer
       arguments:
         ...
```

I’m a Pythonista, Get Me Out of Here!

You:

- Work primarily in Python
- Think YAML is just funky JSON
  (aka it’s for data storage and APIs)
- Know some buzzwords about
  Kubernetes
- Need to run stuff on cloud native

This was me, about 3 years ago

Introducing Hera

The Python SDK for Argo Workflows

Write your templates as functions, orchestrate and test – all in Python

Interact with Argo Workflows entirely through Python

The Best of Both Worlds!

PYTHON

- Developer experience
- Extensive libraries
- Testing
- Code completion
- Preferred scripting language
- Packaging and versioning

ARGO

- The de facto Kubernetes Workflow Orchestration Standard
- Container-native
- Resource management
- Parallelised, isolated workloads
- Artifact storage integrations

Seamless Developer Experience

All-in-one solution

Business logic in functions

Workflow orchestration logic of these functions

Submission of workflows

Extensive documentation

hera.readthedocs.io

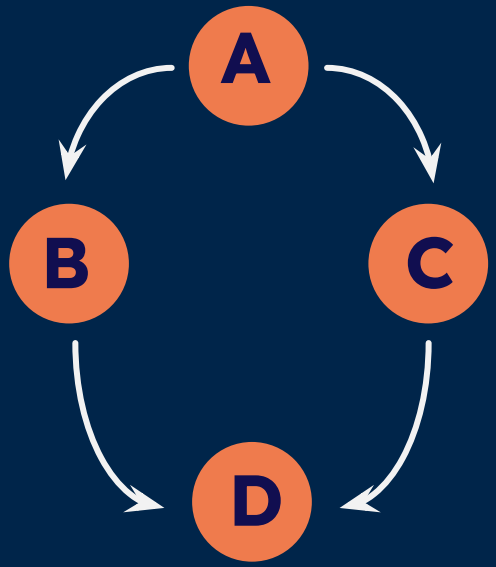
Walkthrough assumes zero knowledge

User guides for more complex features

Enough talk, let's code!

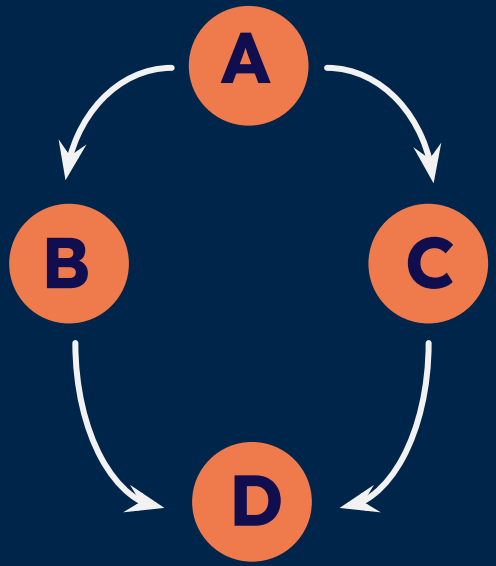
```
pip install hera
```

From the Argo YAML experience…




```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
 generateName: dag-diamond-
spec:
 entrypoint: diamond
 templates:
 - name: diamond
   dag:
     tasks:
     - name: A
       template: echo
       arguments:
         parameters: [{name: message, value: A}]
     - depends: A
       name: B
       template: echo
       arguments:
         parameters: [{name: message, value: B}]
     - depends: A
       name: C
       template: echo
       arguments:
         parameters: [{name: message, value: C}]
     - depends: B && C
       name: D
       template: echo
       arguments:
         parameters: [{name: message, value: D}]
 - name: echo
   container:
     command: ["echo", "{{inputs.parameters.message}}"]
     image: alpine:3.7
   inputs:
     parameters:
     - name: message
```

...to Hera!




```python
from hera.workflows import DAG, Container, Parameter, Task, Workflow

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    echo = Container(
        name="echo",
        image="alpine:3.7",
        command=["echo", "{{inputs.parameters.message}}"],
        inputs=[Parameter(name="message")],
    )
    with DAG(name="diamond"):
        A = Task(name="A", template=echo, arguments={"message": "A"})
        B = Task(name="B", template=echo, arguments={"message": "B"})
        C = Task(name="C", template=echo, arguments={"message": "C"})
        D = Task(name="D", template=echo, arguments={"message": "D"})
        A >> [B, C] >> D
```

...to Hera!

Hera provides many custom classes to help you author your Workflow - with code completion as standard!

```
with Workflow(
    ge
):
    generate_name=
    generation=
    getattr
    GeneratorExit
    deletion_grace_period_seconds=
```

```python
from hera.workflows import DAG, Container, Parameter, Task, Workflow

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    echo = Container(
        name="echo",
        image="alpine:3.7",
        command=["echo", "{{inputs.parameters.message}}"],
        inputs=[Parameter(name="message")],
    )
    with DAG(name="diamond"):
        A = Task(name="A", template=echo, arguments={"message": "A"})
        B = Task(name="B", template=echo, arguments={"message": "B"})
        C = Task(name="C", template=echo, arguments={"message": "C"})
        D = Task(name="D", template=echo, arguments={"message": "D"})
        A >> [B, C] >> D
```

...to Hera!

Hera uses a context manager pattern for Workflows and DAGs, which mirrors the YAML syntax

```python
from hera.workflows import DAG, Container, Parameter, Task, Workflow

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    echo = Container(
        name="echo",
        image="alpine:3.7",
        command=["echo", "{{inputs.parameters.message}}"],
        inputs=[Parameter(name="message")],
    )
    with DAG(name="diamond"):
        A = Task(name="A", template=echo, arguments={"message": "A"})
        B = Task(name="B", template=echo, arguments={"message": "B"})
        C = Task(name="C", template=echo, arguments={"message": "C"})
        D = Task(name="D", template=echo, arguments={"message": "D"})
        A >> [B, C] >> D
```

...to Hera!

Objects are automatically added to the corresponding context - the Container goes into the Workflow, and the Tasks go into the DAG

(And the DAG itself goes into the Workflow!)

```python
from hera.workflows import DAG, Container, Parameter, Task, Workflow

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    echo = Container(
        name="echo",
        image="alpine:3.7",
        command=["echo", "{{inputs.parameters.message}}"],
        inputs=[Parameter(name="message")],
    )
    with DAG(name="diamond"):
        A = Task(name="A", template=echo, arguments={"message": "A"})
        B = Task(name="B", template=echo, arguments={"message": "B"})
        C = Task(name="C", template=echo, arguments={"message": "C"})
        D = Task(name="D", template=echo, arguments={"message": "D"})
        A >> [B, C] >> D
```

With a Sprinkling of Syntactic Sugar

```python
from hera.workflows import DAG, Container, Parameter, Task, Workflow

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    echo = Container(
        name="echo",
        image="alpine:3.7",
        command=["echo", "{{inputs.parameters.message}}"],
        inputs=[Parameter(name="message")],
    )
    with DAG(name="diamond"):
        A = echo(name="A", arguments={"message": "A"})
        B = echo(name="B", arguments={"message": "B"})
        C = echo(name="C", arguments={"message": "C"})
        D = echo(name="D", arguments={"message": "D"})
        A >> [B, C] >> D
```

We “call” the echo container, and pass Task parameters

You can also easily describe task dependencies with the >> (right shift) operator

A Quick Side-By-Side

```python
from hera.workflows import DAG, Container, Parameter, Task, Workflow

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    echo = Container(
        name="echo",
        image="alpine:3.7",
        command=["echo", "{{inputs.parameters.message}}"],
        inputs=[Parameter(name="message")],
    )
    with DAG(name="diamond"):
        A = echo(name="A", arguments={"message": "A"})
        B = echo(name="B", arguments={"message": "B"})
        C = echo(name="C", arguments={"message": "C"})
        D = echo(name="D", arguments={"message": "D"})
        A >> [B, C] >> D
```

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
 generateName: dag-diamond-
spec:
 entrypoint: diamond
 templates:
 - name: diamond
   dag:
     tasks:
     - name: A
       template: echo
       arguments:
         parameters: [{name: message, value: A}]
     - depends: A
       name: B
       template: echo
       arguments:
         parameters: [{name: message, value: B}]
     - depends: A
       name: C
       template: echo
       arguments:
         parameters: [{name: message, value: C}]
     - depends: B && C
       name: D
       template: echo
       arguments:
         parameters: [{name: message, value: D}]
 - name: echo
   container:
     command: ["echo", "{{inputs.parameters.message}}"]
     image: alpine:3.7
   inputs:
     parameters:
     - name: message
```

Which do you prefer?

Container-native is cool and all...

But what about Python-native?

Functions *Are* Templates!

The script decorator containerises your function!

```python
from hera.workflows import DAG, Workflow, script

@script(image="python:3.12")
def echo(message):
    print(message)

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    with DAG(name="diamond"):
        A = echo(name="A", arguments={"message": "A"})
        B = echo(name="B", arguments={"message": "B"})
        C = echo(name="C", arguments={"message": "C"})
        D = echo(name="D", arguments={"message": "D"})
        A >> [B, C] >> D
```

Functions *Are* Templates!

Same syntax as containers to create Tasks

```python
from hera.workflows import DAG, Workflow, script

@script(image="python:3.12")
def echo(message):
    print(message)

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    with DAG(name="diamond"):
        A = echo(name="A", arguments={"message": "A"})
        B = echo(name="B", arguments={"message": "B"})
        C = echo(name="C", arguments={"message": "C"})
        D = echo(name="D", arguments={"message": "D"})
        A >> [B, C] >> D
```

Create Your Workflow From Your Favourite IDE/Terminal

Add some info to communicate with your Argo instance – your namespace and a **WorkflowsService**

“Create” your workflow!

```python
from hera.workflows import DAG, Workflow, WorkflowsService

with Workflow(
    generate_name="dag-diamond-",
    entrypoint="diamond",
    namespace="argo",
    workflows_service=WorkflowsService(
        host="https://localhost:2746"),
) as w:
    with DAG(name="diamond"):
        ...
w.create()
```

See It Running…

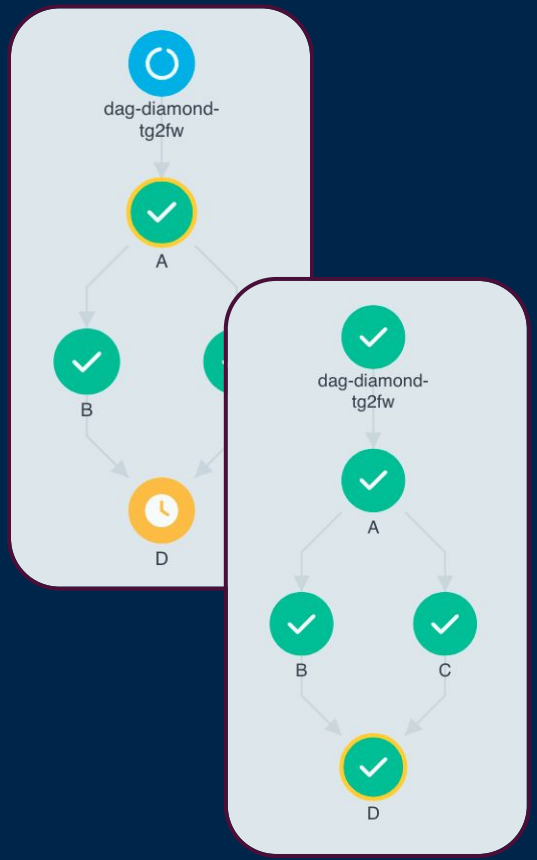
```python
from hera.workflows import DAG, Workflow, WorkflowsService
import hera.workflows.models as m

HOST = "https://localhost:2746"
with Workflow(
    generate_name="dag-diamond-",
    entrypoint="diamond",
    namespace="argo",
    workflows_service=WorkflowsService(
        host=HOST),
) as w:
    with DAG(name="diamond"):
        ...
submitted_w = cast(m.Workflow, w.create())
name = submitted_w.metadata.name
namespace = submitted_w.metadata.namespace
print(f"Submitted {name}")
print(f"Open {HOST}/workflows/{namespace}/{name}")
```

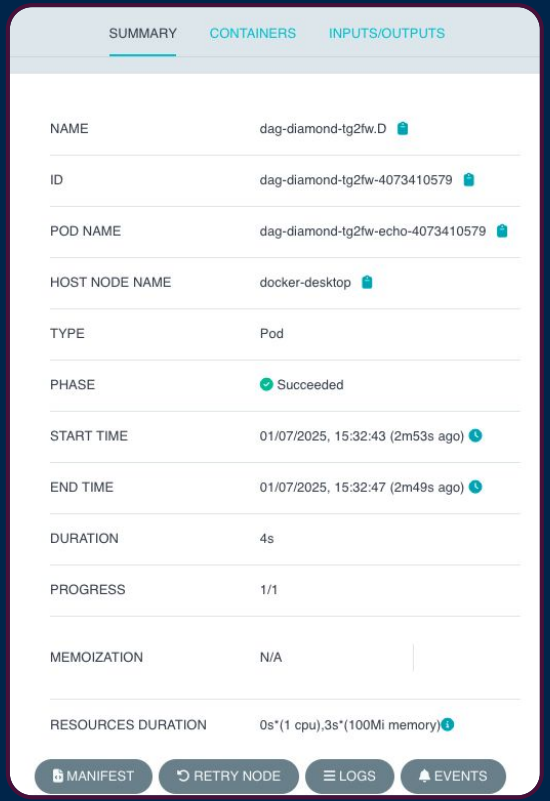
Inspect the created Workflow

Log from running locally:

```
Submitted dag-diamond-tg2fw
Open https://localhost:2746/workflows/argo/dag-diamond-tg2fw
```

On The Argo UI!





Where you can see

- live progress
- DAG task details in the sidebar
- per-container logs

This is an “Inline” Script Template

Hera “compiles” your workflow definition to YAML for Argo to understand it

Inline script templates dump the function body into the `source` field of the YAML

```python
from hera.workflows import script

@script(image="python:3.12")
def echo(message: str):
    print(message)
```

```yaml
- name: echo
  inputs:
    parameters:
    - name: message
  script:
    command:
    - python
    image: python:3.12
    source: |-
      import json
      try: message = json.loads(r'''{{inputs.parameters.message}}''')
      except: message = r'''{{inputs.parameters.message}}'''

      print(message)
```

Spot the Problem(s) with this Code…

Where did the type go?!

```python
from hera.workflows import script

@script(image="python:3.12")
def echo(message: str):
    print(message)  # plus, returns are not allowed!
```

```yaml
- name: echo
  inputs:
    parameters:
    - name: message
  script:
    command:
    - python
    image: python:3.12
    source: |-
      import json
      try: message = json.loads(r'''{{inputs.parameters.message}}''')
      except: message = r'''{{inputs.parameters.message}}'''

      print(message)
```

Hera helps by adding **json.loads** for your parameters, but we're at the mercy of whatever value the user gives us!

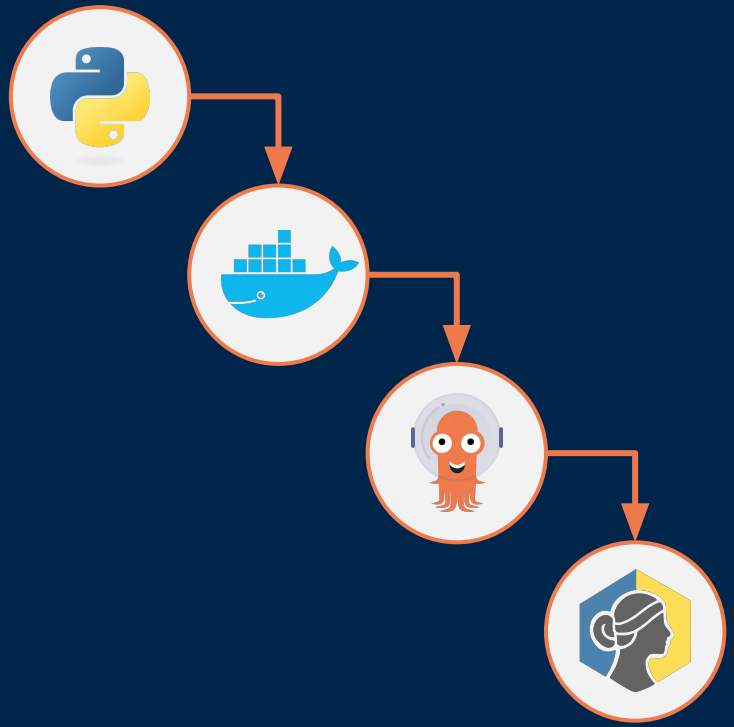
Runner-na-na-na-na-na-na na-na-na-na-na-na-na-na Batman!

The Hera Script Runner

The Hera Script Runner

Hera's Script Runner lets you easily run native Python functions on Argo Workflows

1. First, build a Docker/OCI image with your code and dependencies
2. Then, Argo pulls your image and Hera runs your code



Type-safe Functions

The Hera runner deserializes and type checks the inputs at runtime

With the Hera runner, we can also return values straight from the function

```python
from hera.workflows import script

@script(
    constructor="runner",
    image="my-built-python-image",
)
def calculate_area_of_rectangle(
    length: float, width: float
) -> float:
    return length * width
```

Testing Locally

```python
from hera.workflows import script

@script(
    constructor="runner",
    image="my-built-python-image",
)
def calculate_area_of_rectangle(
    length: float, width: float
) -> float:
    return length * width
```

You can test your script template like normal Python code

```python
def test_calculate_area_of_rectangle():
   assert calculate_area_of_rectangle(2.0, 3.0) == 6.0
```

Not possible in YAML!

How to Set Up the Hera Runner

Just add two values to the script decorator!

(Remember you'll need to build and push the image later)

```python
from hera.workflows import script

@script(
    constructor="runner",
    image="my-built-python-image",
)
def calculate_area_of_rectangle(
    length: float, width: float
) -> float:
    return length * width
```

```yaml
- name: calculate-area-of-rectangle
  inputs:
    parameters:
    - name: length
    - name: width
  script:
    image: my-built-python-image
    source: '{{inputs.parameters}}'
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_package.workflow:calculate_area_of_rectangle
```

It's PY-dantic, not PE-dantic!

```
pip install pydantic
```

Pydantic-powered functions

Type hints during development

Type validation at runtime

Pydantic BaseModel template Inputs and Outputs (i.e. custom structured IO)

Automagic JSON (de)serialisation

Using Pydantic

Let's replace the function inputs with a new "Rectangle" class

```python
from hera.workflows import script

@script(
    constructor="runner",
    image="my-built-python-image",
)
def calculate_area_of_rectangle(
    length: float, width: float
) -> float:
    return length * width
```

Using Pydantic

Create a BaseModel subclass (with its own area function)

```
class Rectangle(BaseModel):
    length: float
    width: float

    def area(self) -> float:
        return self.length * self.width
```

Using Pydantic

```python
class Rectangle(BaseModel):
    length: float
    width: float

    def area(self) -> float:
        return self.length * self.width
```

Use the new class as a function input argument

```python
@script(constructor="runner", image="my-built-python-image")
def calculate_area_of_rectangle(
    rectangle: Rectangle
) -> float:
    return rectangle.area()
```

And test your *script template*!

```python
def test_calculate_area_of_rectangle():
    r = Rectangle(length=2.0, width=3.0)
    assert calculate_area_of_rectangle(r) == 6.0
```

Need Custom Output Serialisation? We've Got That Too!

Dump any kind of binary format using user-defined functions passed into annotations

```
@script(constructor="runner")
def create_dataframe_parquet() -> Annotated[
    DataFrame,
    Artifact(
        name="dataset",
        dumpb=lambda df: df.to_parquet(),
        archive=NoneArchiveStrategy(),
    ),
]:
    data = {
        "age": [23, 19, 43, 65, 72],
        "height": [1.63, 1.82, 1.77, 1.59, 1.61],
    }
    return DataFrame(data)
```

What Else Can Hera Do?

- Test Workflows end-to-end
  - Check execution path and outputs
- Set class defaults and pre-build hooks
  - Helps platform teams set up wrapper packages to provide common configurations for Hera and Argo Workflows
- Easy Workflow versioning
  - Piggyback off Python versioning to enable versioned Workflows and WorkflowTemplates

`python -m demo`

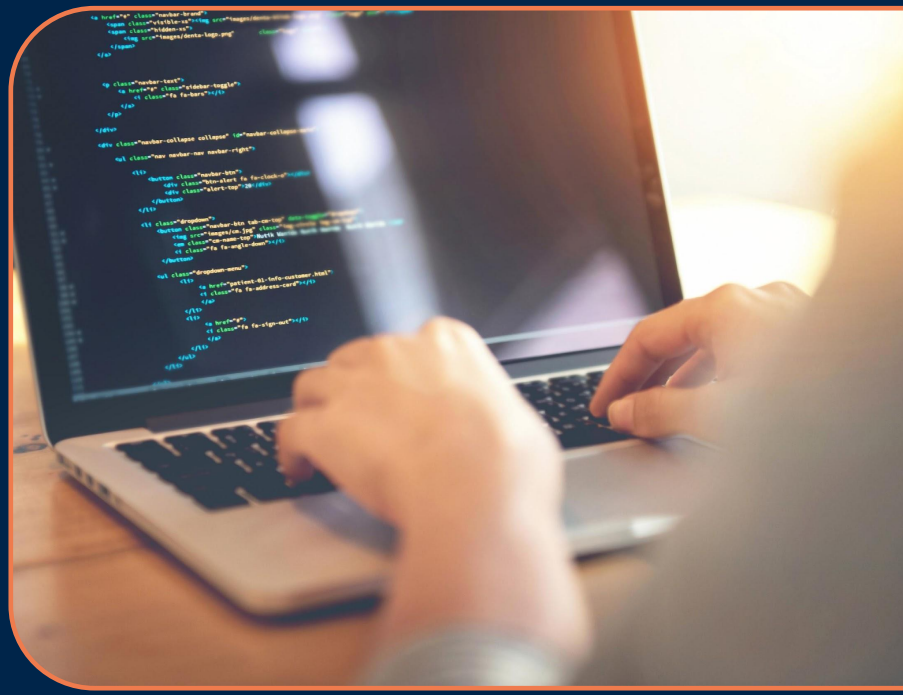
Repo: github.com/elliotgunton/hera-example-project

Key Takeaways

- Don’t let YAML be a barrier to Kubernetes adoption
- Hera makes Workflow Orchestration on Kubernetes accessible
- Hera supercharges your Pythonic Argo Workflows experience

TL;DR:

Argo Workflows and Hera can give you the best cloud-native workflow orchestration experience



Connect with us!

GitHub

argoproj-labs/hera

Docs

hera.readthedocs.io

Slack

slack.cncf.io

#hera-argo-sdk

Further reading

GitHub repo of Pipekit's previous talks and demos

github.com/pipekit/talk-demos

Questions?