ArgoCon

EUROPE

ArgoCon
EUROPE

Orchestrating Python Functions Natively with Hera

ArgoCon EU

March 19th, 2024

Speakers

ELLIOT GUNTON

Bloomberg Engineering

Senior Software Engineer

Hera maintainer

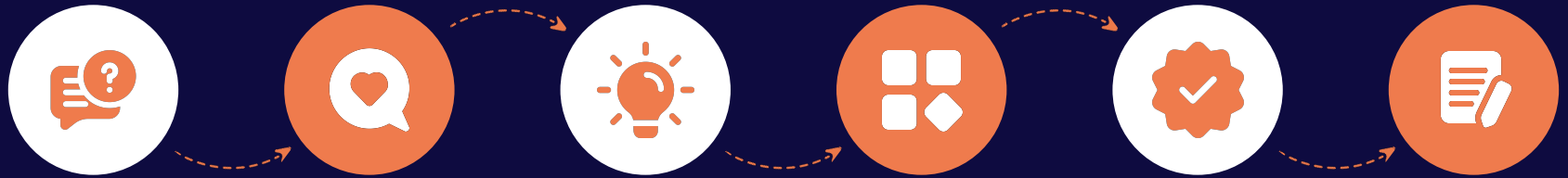
JP ZIVALICH

pipekit

CTO & Founder

Hera evangelist

Outline



Why build on Argo Workflows?

The Argo developer experience

Introducing Hera

Hera features and examples

Case Studies: How Hera helps solve problems

Takeaways

The Whats and Whys

Why build on Argo Workflows?

1. The de facto Kubernetes Workflow Orchestration Standard
2. Allows users to scale more than non-cloud-native tools
3. Kubernetes and container-native
4. Active and growing community
5. Long-term support
6. Vendor-neutral

What problems does Argo Workflows have?

YAML is a barrier to entry

Hard to test

Hard to reuse (outside of `WorkflowTemplates`)

Hard to maintain long `Workflows` and `WorkflowTemplates`

```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:
         ...
```

Why build on Python?

- Mature developer experience
- Extensive libraries
- Preferred scripting language for many
- Code completion features
- Access to testing frameworks
  - Unit testing of individual functions
  - System testing of the whole workflow
- Packaging and versioning

How can we get the best of both?

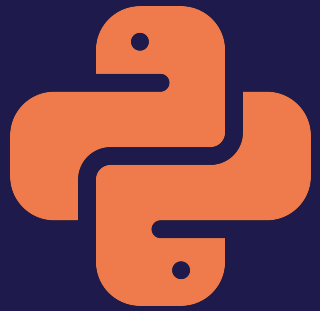
PYTHON

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

ARGO

- The de facto Kubernetes Workflow Orchestration Standard
- Kubernetes and container-native
- Long-term support
- Active and growing community
- Vendor-neutral

Introducing Hera!



The Python SDK for Argo Workflows

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

Lets you interact with Argo Workflows entirely through Python!

Did we mention we love Python?!?

Seamless Developer Experience

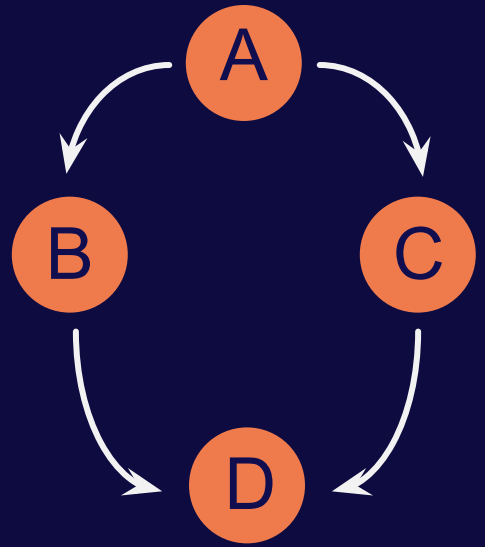
- Business logic in functions
- Workflow orchestration logic of these functions
- Submission of workflows

- Walkthrough assumes zero knowledge
- User guides for more complex features

Enough talk, let's code!

```
pip install hera
```

From the Argo YAML experience…

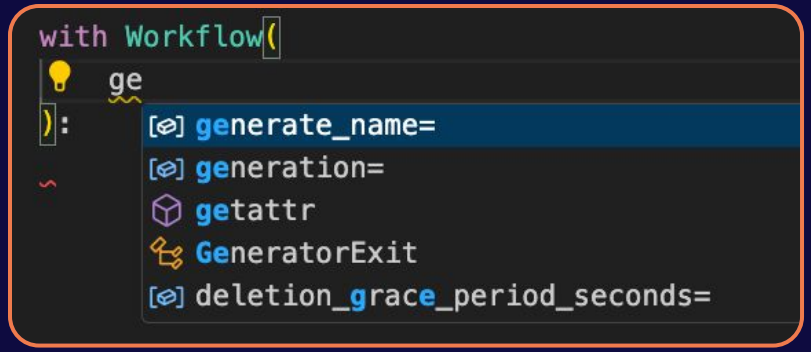



```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
 generateName: dag-diamond-
spec:
 entrypoint: diamond
 templates:
 - name: diamond
   dag:
     tasks:
     - name: A
       template: echo
       arguments:
         parameters: [{name: message, value: A}]
     - depends: A
       name: 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!

ArgoCon EUROPE

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




```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, DAGs, Steps and more, 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 like this Container template are automatically detected and added to the Workflow's templates

```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!

While these Tasks are added to the DAG

```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

We can simply “call” the echo container template!

You can easily describe task dependencies with the `rshift` operator!

```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
```

Container templates?
I want more Python!

Functions *are* templates!

Use your functions as script templates with the script decorator!

Call them in your DAGs and Steps!

```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 straight from your favourite IDE!

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

Call `create` on 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()
```

A script template caveat…

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

Script templates expect the code dumped into the `source` field of the YAML

In Hera, we call this an “inline” script because the code is “inline” in your YAML

```python
from hera.workflows import script

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

A script template caveat…

So, our function is type-less and practically untestable!

```python
from hera.workflows import script


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

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

```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)
```

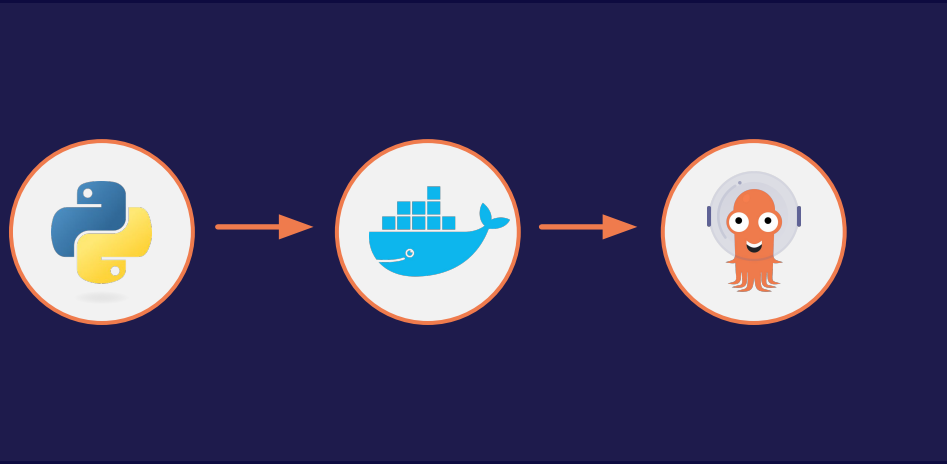
What if we could do better?

The Hera Script Runner

ArgoCon EUROPE

Hera's Script Runner runs on an image built using your code

- Users can build a Buildpacks/Docker image with their code and dependencies
- Argo then pulls that image to run the code as written



The YAML that Hera exports uses some magic to make this happen!

The command is still Python, but the source is now the input parameters which are passed to the `hera.workflows.runner` module

The Hera Script Runner

ArgoCon EUROPE

Hera's Script Runner runs on an image built using your code

- Users can build a Buildpacks/Docker image with their code and dependencies
- Argo then pulls that image to run the code as written

The YAML that Hera exports uses some magic to make this happen!

The command is still Python, but the source is now the input parameters which are passed to the `hera.workflows.runner` module

```yaml
script:
  image: my-built-python-image
  command:
  - python
  args:
  - -m
  - hera.workflows.runner
  - -e
  - argocon_tutorial:calculate_area_of_rectangle
  source: '{{inputs.parameters}}'
```

Type-safe functions!

We first need to specify the `constructor` as “runner”

And we need our template to run on an `image` that we’re going to build from this code

```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
```

Type-safe functions!

Parameters are still inferred from the function signature and exported to YAML

The Hera runner deserializes and type checks the inputs at runtime!

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

```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
```

Type-safe functions!

ArgoCon EUROPE

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

Which means…

```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
```

Type-safe functions – testing!

ArgoCon EUROPE

```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
```

We can test script template functions with normal Python tests!

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

Type-safe functions – testing!

ArgoCon EUROPE

However, only JSON types are allowed in the inputs…

```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

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

What if we could do...
even better?

Hera Pydantic

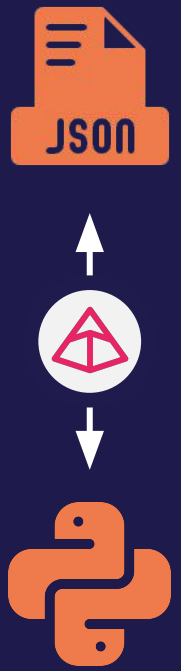
```
pip install pydantic
```

Pydantic-powered functions!

The Hera Script Runner integrates with Pydantic for type validation, and allows Pydantic classes in your template inputs and outputs

- You get Python type hints and auto-completion during template development
- The Script Runner gets your Pydantic objects for you by automatically deserializing JSON string inputs

Ensuring type safety for input objects at runtime!



Pydantic-powered functions!

ArgoCon EUROPE

Let’s create a Pydantic BaseModel class

And we can give it a function

```python
from pydantic import BaseModel

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

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

Pydantic-powered functions!

ArgoCon EUROPE

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

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

Pass Pydantic types directly into (and out of) functions!

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

Unit test your template logic just like any other function!

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

I like testing things!

Good news, everyone!
Workflows are testable!

We can launch a workflow,
waiting until completion…

```
with Workflow(generate_name="hello-world-", entrypoint="steps") as w:
    with Steps(name="steps"):
        echo_to_param(arguments={"message": "Hello world!"})

def test_create_workflow():
    model_workflow = w.create(wait=True)
    assert model_workflow.status and
        model_workflow.status.phase == "Succeeded"

    echo_node = next(filter(
            lambda n: n.display_name == "echo-to-param",
            model_workflow.status.nodes.values(),
    ))

    assert echo_node.outputs.parameters[0].value == "Hello world!"
```

Good news, everyone!
Workflows are testable!

Inspect the Workflow's status…

```python
with Workflow(generate_name="hello-world-", entrypoint="steps") as w:
    with Steps(name="steps"):
        echo_to_param(arguments={"message": "Hello world!"})

def test_create_workflow():
    model_workflow = w.create(wait=True)
    assert model_workflow.status and
        model_workflow.status.phase == "Succeeded"

    echo_node = next(filter(
            lambda n: n.display_name == "echo-to-param",
            model_workflow.status.nodes.values(),
    ))

    assert echo_node.outputs.parameters[0].value == "Hello world!"
```

Good news, everyone!
Workflows are testable!


```
with Workflow(generate_name="hello-world-", entrypoint="steps") as w:
    with Steps(name="steps"):
        echo_to_param(arguments={"message": "Hello world!"})

def test_create_workflow():
    model_workflow = w.create(wait=True)
    assert model_workflow.status and
        model_workflow.status.phase == "Succeeded"

    echo_node = next(filter(
            lambda n: n.display_name == "echo-to-param",
            model_workflow.status.nodes.values(),
    ))

    assert echo_node.outputs.parameters[0].value == "Hello world!"
```

Get individual nodes…

And check their outputs!

Hera’s future goals

- Bring automatic Python versioning to WorkflowTemplates
- Leverage Python dependency management to manage WorkflowTemplate dependencies

How is Hera used?

Bloomberg

ArgoCon EUROPE

Problem

Building and testing complex ML model remediation pipelines

- Argo Workflows has all the features we need for orchestrating these pipelines
- It is provided as a service at Bloomberg, but was not adopted for ML use cases
  - Difficult to write and test Workflows with YAML
  - YAML workflows had Python code dumped into script templates

Solution

Provide a Python experience for Argo Workflows

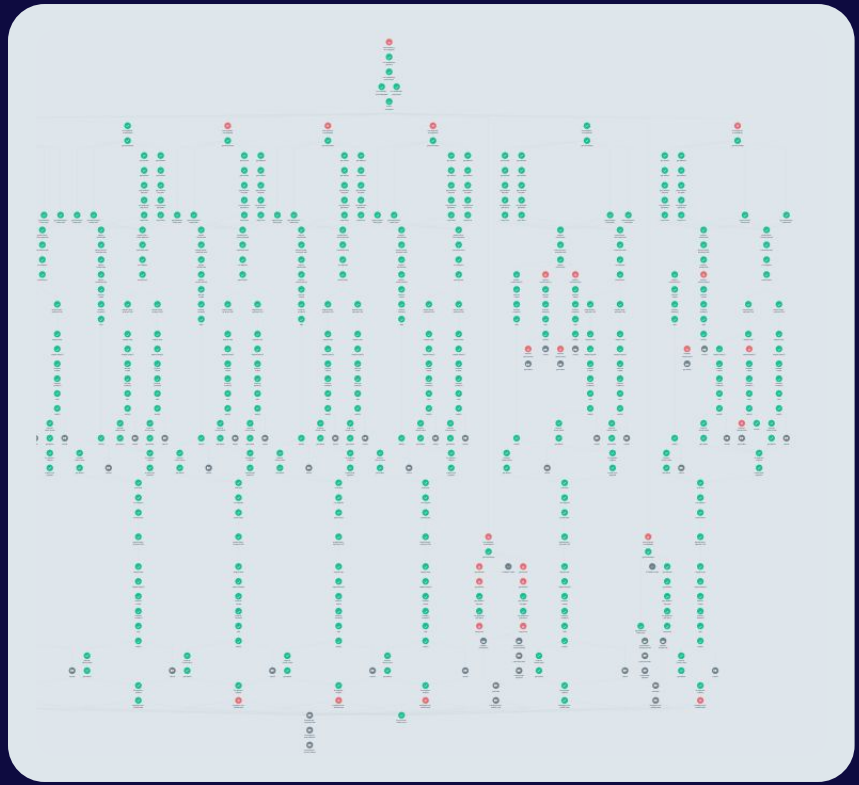
- Work on and release Hera V5
- Provide supporting tools
  - Cookiecutter to provide a boilerplate Python package
  - Testing framework
  - Documentation generator for Workflow Templates
- Workshops and support channels

TechAtBloomberg.com

Bloomberg

Engineering

Bloomberg



Hera has enabled our 300+ AI engineers to use Argo Workflows in production

TechAtBloomberg.com

Bloomberg

Engineering

Pipekit

PROBLEM

managing Argo at scale

- Monitoring/observability for Workflows
- Manage Argo/Hera at scale and across multiple clusters

SOLUTION

tools and support for managing Argo at scale

- Providing easy integrations with common tools
- `pipekit-sdk` - builds on Hera to allow users to submit to multiple clusters, interact with workflows, and see logs from a single Jupyter Notebook or any Python interpreter

LEARNINGS

- Need to meet both the platform team and data scientists where they're at
- Platform teams need to use normal K8s primitives and declarations they're used to
- Data scientists want to use languages and concepts they're used to

ACCURE Battery Intelligence

ArgoCon EUROPE

Problem

YAML and slow Docker builds blocking productivity

- Difficult to set resources and limits for Python developers
- Unaware of what parameters they can play with
- Had to rebuild the Docker image every time code was changed

Solution

Adopt Hera to improve developer experience

- Hera helps with discoverability of Workflows parameters
- Using Hera's "inline" scripts allows developers to avoid rebuilding the image

Energy Company

ArgoCon EUROPE

Problem

Markets moving too fast for slow iteration speeds

- Data scientists are familiar with Python and not YAML
- Required help from MLOps team to run workflows
  - MLOps team blocked from details by RBAC policies

Solution

Adopt Hera to allow the data science team to be self serve

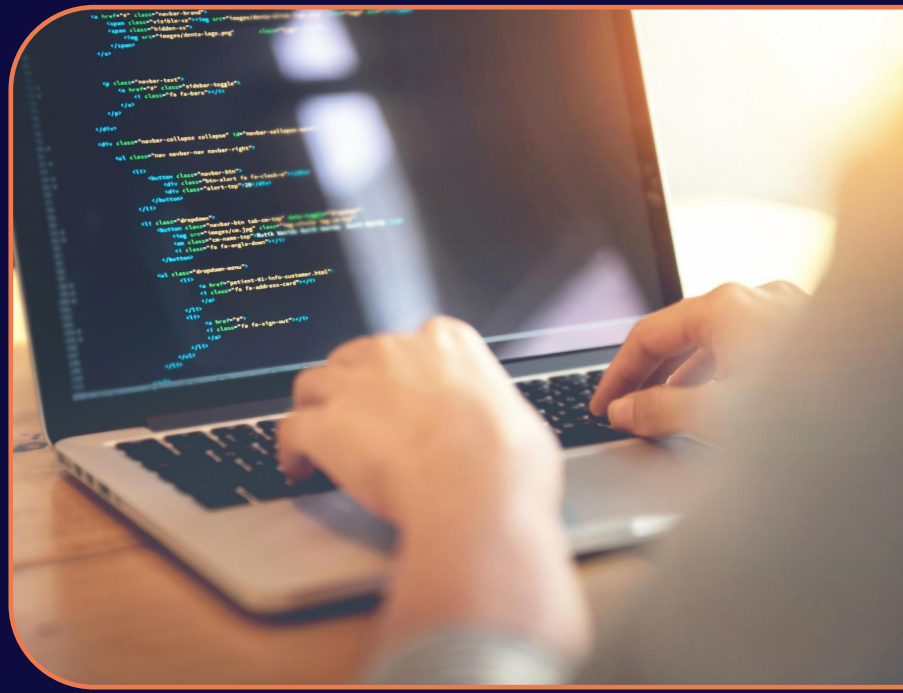
- All-in-one solution for data scientists to write training code, orchestration logic, and submission to the Argo Workflows clusters

Key Takeaways

- Hera makes Argo Workflows easy
- Argo does the heavy lifting of orchestrating your Python functions
- Hera is used in production, operating at scale

TL;DR:

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



Thank you & Questions

Repository
github.com/argoproj-labs/hera

Docs
hera.readthedocs.io

#hera-argo-sdk channel on slack.cncf.io

Working Group meetings coming soon!