ArgoCon

#argocon

ArgoCon NA 2024

Data Science Workflows Made Easy

Python-Powered Argo for Your Organization

November 12, 2024

About us

pipekit

Elliot Gunton

Senior Software Engineer
Hera maintainer

xaira

Flaviu Vadan

Senior Software Engineer
Hera author/maintainer

About Pipekit

Scale Argo & Kubernetes with Pipekit

- Direct support from 40% of the active Argo Workflows maintainers & Hera maintainers in the world
- Save engineering time and up to 60% on compute costs
- Add 3 Argo maintainers, 7 Argo contributors and 1 Hera maintainer to your team
- Serving startups & Fortune 500 enterprises 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

About Xaira Therapeutics

xaira

Drug design

Experiments

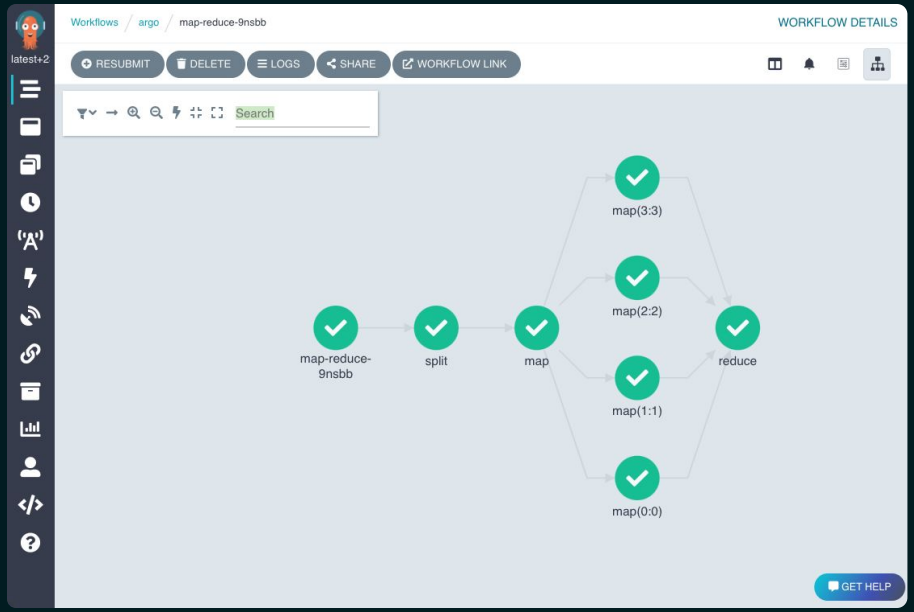
Diseases

New biology

Outline

1. What is Hera?
2. A Python-flavored script template deep dive
3. Local developer experience and hooks
4. Writing workflows with Hera
5. Analysis sample walkthrough

What is Argo Workflows?



The Kubernetes Workflow Orchestration engine

Scalable, Kubernetes and container-native

Vendor neutral with an active community

What is Hera?

Hera is the Python SDK for Argo Workflows:

- Fully featured
- Python experience
- Option for GitOps experience

Hera Internal Platform

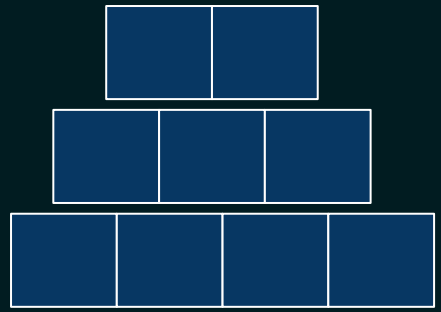
Hera **hooks** allow you to:

- Integrate your platform
- Empower your users
- Scale your team

github.com/pipekit/talk-demos

Users / Workflows

Python as usual, high level workflows, experimentation, and business logic.



Platform / Infrastructure

Authentication, GPU flags, K8s annotations, image defaults, etc.

Script Templates Deep Dive

Inline Script Templates

The Python function is dumped into the YAML as source code

```python
from hera.workflows import script

@script(image="python:3.12")
def my_matcher(string: str):
    import re
    MATCH_ME = "argo"
    print(bool(re.match(MATCH_ME, string)))
```

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

      import re
      MATCH_ME = 'argo'
      print(bool(re.match(MATCH_ME, string)))
```

Inline Script Templates

```python
from hera.workflows import script

@script(image="python:3.12")
def my_matcher(string: str):
    import re
    MATCH_ME = "argo"
    print(bool(re.match(MATCH_ME, string)))
```

The source code is run through the `python` command in the Python `image` you specify

Equivalent to running `python my_code.py` in the container

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

      import re
      MATCH_ME = 'argo'
      print(bool(re.match(MATCH_ME, string)))
```

Inline Script Templates

You can use any image that contains your dependencies

```python
from hera.workflows import script

@script(image="python:3.12")
def my_matcher(string: str):
    import re
    MATCH_ME = "argo"
    print(bool(re.match(MATCH_ME, string)))
```

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

      import re
      MATCH_ME = 'argo'
      print(bool(re.match(MATCH_ME, string)))
```

Drawbacks for Inline Script Templates

Good for quick prototyping but comes with large drawbacks:

Imports and helper functions from outside the inline function **cannot be used**

```python
from hera.workflows import script

@script(image="python:3.12")
def my_matcher(string: str):
    import re
    MATCH_ME = "argo"
    print(bool(re.match(MATCH_ME, string)))


from hera.workflows import Artifact

@script(inputs=Artifact(name="in-art", path="/tmp/file"))
def consumer():
    with open("/tmp/file", "r") as f:
        print(f.readlines())
```

Drawbacks for Inline Script Templates

Good for quick prototyping but comes with large drawbacks:

Hard to **unit test**

```python
from hera.workflows import script

@script(image="python:3.12")
def my_matcher(string: str):
    import re
    MATCH_ME = "argo"
    print(bool(re.match(MATCH_ME, string)))
```

```python
from hera.workflows import Artifact

@script(inputs=Artifact(name="in-art", path="/tmp/file"))
def consumer():
    with open("/tmp/file", "r") as f:
        print(f.readlines())
```

Drawbacks for Inline Script Templates

Good for quick prototyping but comes with large drawbacks:

```python
from hera.workflows import script

@script(image="python:3.12")
def my_matcher(string: str):
    import re
    MATCH_ME = "argo"
    print(bool(re.match(MATCH_ME, string)))
```

Artifacts are **harder to use** →

```python
from hera.workflows import Artifact

@script(inputs=Artifact(name="in-art", path="/tmp/file"))
def consumer():
    import json
    with open("/tmp/file", "r") as f:
        my_artifact = json.loads(f.read())
```

They must be manually loaded and dumped, and **custom data types** are hard to use →

Runner Script Templates

Runner Script Templates

Native Python - no more self-contained functions!

```python
from hera.workflows import script
import re
MATCH_ME = "argo"

@script(image="argocon-hera:v1", constructor="runner")
def my_matcher(string: str) -> bool:
    return bool(re.match(MATCH_ME, string))
```

```yaml
- name: my-matcher
  inputs:
    parameters:
    - name: string
  script:
    image: argocon-hera:v1
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_module.my_file:my_matcher
    source: '{{inputs.parameters}}'
```

Runner Script Templates

Functions can be easily tested!

```
assert my_matcher("argo")
```

```python
from hera.workflows import script
import re
MATCH_ME = "argo"

@script(image="argocon-hera:v1", constructor="runner")
def my_matcher(string: str) -> bool:
    return bool(re.match(MATCH_ME, string))
```

```yaml
- name: my-matcher
  inputs:
    parameters:
    - name: string
  script:
    image: argocon-hera:v1
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_module.my_file:my_matcher
    source: '{{inputs.parameters}}'
```

Runner Script Templates

Enable using the `constructor` kwarg

```python
from hera.workflows import script
import re
MATCH_ME = "argo"

@script(image="argocon-hera:v1", constructor="runner")
def my_matcher(string: str) -> bool:
    return bool(re.match(MATCH_ME, string))
```

```yaml
- name: my-matcher
  inputs:
    parameters:
    - name: string
  script:
    image: argocon-hera:v1
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_module.my_file:my_matcher
    source: '{{inputs.parameters}}'
```

Runner Script Templates

The Hera Runner is the Python command module entrypoint, which runs your function in the container →

```python
from hera.workflows import script
import re
MATCH_ME = "argo"

@script(image="argocon-hera:v1", constructor="runner")
def my_matcher(string: str) -> bool:
    return bool(re.match(MATCH_ME, string))
```

```yaml
- name: my-matcher
  inputs:
    parameters:
    - name: string
  script:
    image: argocon-hera:v1
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_module.my_file:my_matcher
    source: '{{inputs.parameters}}'
```

Runner Script Templates

```python
from hera.workflows import script
import re
MATCH_ME = "argo"

@script(image="argocon-hera:v1", constructor="runner")
def my_matcher(string: str) -> bool:
    return bool(re.match(MATCH_ME, string))
```

The `image` must be built from dependencies and your code

```
docker build . -t argocon-hera:v1
```

```yaml
- name: my-matcher
  inputs:
    parameters:
    - name: string
  script:
    image: argocon-hera:v1
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_module.my_file:my_matcher
    source: '{{inputs.parameters}}'
```

Runner Script Templates

Key takeaway

Using Runner scripts makes Python on Argo intuitive!

```python
from hera.workflows import script
import re
MATCH_ME = "argo"

@script(image="argocon-hera:v1", constructor="runner")
def my_matcher(string: str) -> bool:
    return bool(re.match(MATCH_ME, string))
```

```yaml
- name: my-matcher
  inputs:
    parameters:
    - name: string
  script:
    image: argocon-hera:v1
    command:
    - python
    args:
    - -m
    - hera.workflows.runner
    - -e
    - my_module.my_file:my_matcher
    source: '{{inputs.parameters}}'
```

Script Annotations

Metadata stored in **Annotated** is read by Hera and used to build the workflow

```python
@script(constructor="runner")
def output_dict(
    a_number: Annotated[
        int,
        Parameter(name="number-param"),
    ],
) -> Annotated[
    Dict[str, int],
    Parameter(name="dict-param"),
]:
    return {"your-value": a_number}
```

Hera's Automatic Deserialization

The Hera Runner can automatically deserialize JSON inputs to arbitrary Python objects...

```python
from pydantic import BaseModel

class MyObject(BaseModel):
    a_dict: dict
    a_str: str = "a default string"

@script(constructor="runner")
def artifact_loader(
    my_object: Annotated[
        MyObject,
        Artifact(
            name="my-obj-artifact",
            loader=ArtifactLoader.json,
        ),
    ],
) -> Annotated[int, Artifact(name="dict-value")]:
    return my_object.a_dict["key"]
```

Hera's Automatic Deserialization

...And automatically serialize arbitrary objects as template outputs

→

```python
from pydantic import BaseModel

class MyObject(BaseModel):
    a_dict: dict
    a_str: str = "a default string"

@script(constructor="runner")
def artifact_loader(
    my_object: Annotated[
        MyObject,
        Artifact(
            name="my-obj-artifact",
            loader=ArtifactLoader.json,
        ),
    ],
) -> Annotated[int, Artifact(name="dict-value")]:
    return my_object.a_dict["key"]
```

Script Templates Recap

1. Inline templates are good for prototyping and trying out Hera

2. Runner templates are better for everything else

3. Read point 2 again

Writing Workflow Logic

Context Manager Syntax

Context manager syntax aligns with YAML syntax

Helping users grasp concepts from both YAML and Python docs!

```python
with Workflow(
    generate_name="model-training-pipeline-",
    entrypoint="run-training",
) as w:
    with DAG(name="run-training"):
        datasets_task = load_and_split_dataset()
        scaling_task = feature_scaling(
            arguments=[
                datasets_task.get_artifact("X_train"),
                datasets_task.get_artifact("X_test"),
            ],
        )
        model_training_task = model_training(
            arguments=[
                scaling_task.get_artifact("X_train"),
                datasets_task.get_artifact("y_train"),
            ],
        )
        (
            datasets_task
            >> scaling_task
            >> model_training_task
        )
```

Syntactic Sugar for DAG construction

How can we avoid Task/Script boilerplate?

```python
def model_training():
    ...
train_task = Task(
    name="model_training",
    template=Script(
        name="model-training",
        source=model_training,
    ),
    arguments=[...],
)
```

Syntactic Sugar for DAG construction

```python
def model_training():
    ...
train_task = Task(
    name="model_training",
    template=Script(
        name="model-training",
        source=model_training,
    ),
    arguments=[...],
)

# is equivalent to...
@script()
def model_training():
    ...

with DAG(name="run-training"):
    train_task = model_training(arguments=[...])
```

Call the function to do all this for you! 

Syntactic Sugar for DAG construction

Easily specify `Task` dependencies →

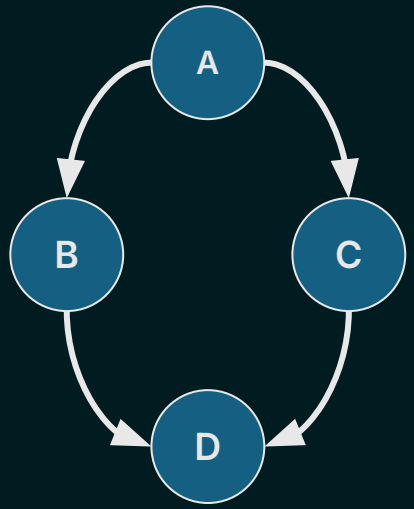
```
datasets_task >> scaling_task >> model_training_task
```

Syntactic Sugar for DAG construction

Declare multiple dependencies at once!

```
datasets_task >> scaling_task >> model_training_task

A >> [B, C] >> D
```



Looping Looping Looping

```python
@script()
def generate():
    import json
    import sys
    json.dump([i for i in range(10)], sys.stdout)


@script()
def consume(value: int):
    print(f"Received value: {value}!")
```

```python
with Workflow(
    generate_name="dynamic-fanout-",
    entrypoint="d",
) as w:
    with DAG(name="d"):
        generator_task = generate()
        consumer_task = consume(
            with_param=generator_task.result
        )
        generator_task >> consumer_task
```

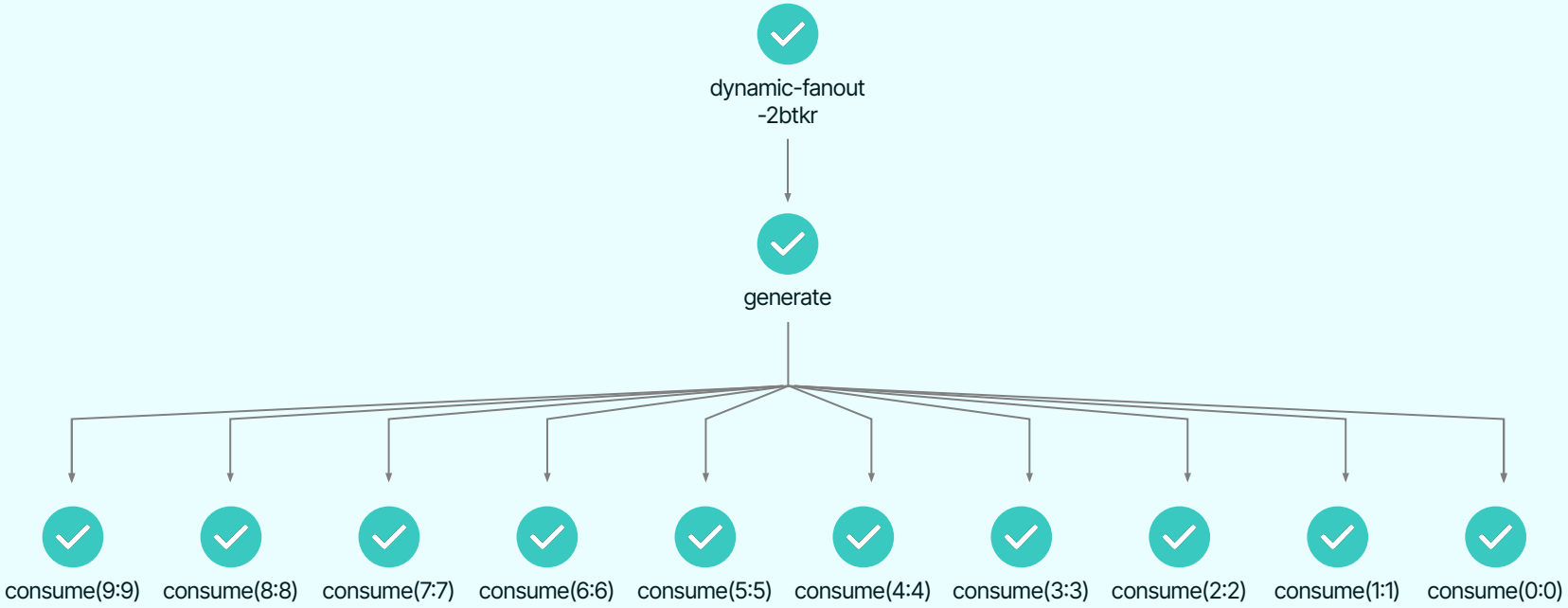
Looping Looping Looping

```python
@script()
def generate():
    import json
    import sys
    json.dump([i for i in range(10)],
sys.stdout)


@script()
def consume(value: int):
    print(f"Received value: {value}!")
```

```python
with Workflow(
    generate_name="dynamic-fanout-",
    entrypoint="d",
) as w:
    with DAG(name="d"):
        generator_task = generate()
        consumer_task = consume(
            with_param=generator_task.result
        )
        generator_task >> consumer_task
```

Looping Looping Looping



Developing Workflows

Runner Scripts Local(ish) Development Lifecycle

Write code and test locally!

Runner Scripts Local(ish) Development Lifecycle

Write code and test locally!

Set script template image name using Hera's config, write workflow

```
global_config.image = "my-image:v1"
```

Runner Scripts Local(ish) Development Lifecycle

Write code and test locally!

Set script template image name using Hera's config, write workflow

```
global_config.image = "my-image:v1"
```

Build an image from your code

```
docker build . -t my-image:v1
```

Push to your Argo-linked image repo

```
docker push my-org/my-image:v1
```

Runner Scripts Local(ish) Development Lifecycle

Write code and test locally!

Set script template image name using Hera's config, write workflow

```
global_config.image = "my-image:v1"
```

Build an image from your code

```
docker build . -t my-image:v1
```

Push to your Argo-linked image repo

```
docker push my-org/my-image:v1
```

Create Workflow on cluster to test the workflow itself

```
argo submit my-workflow.yaml
```

Or create it through a Hera Python script!

How can we automate the development lifecycle?

Continuous Integration for Workflows

Build your new image in the Pull Request CI

```
[.github/workflows/cicd.yaml]
steps:
# GitHub Workflow setup steps...
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: v1.1-rc${{ github.event.number }}
- name: Build and run workflow
  run: IMAGE_NAME=argocon-hera:v1.1-rc${{ github.event.number }}
       python -m my_module.run_workflow
```

```
[src/my_module/run_workflow.py]
global_config.image = os.environ.get("IMAGE_NAME", "argocon-hera:v1")
with Workflow(generate_name="my-workflow-") as w:
    # Build and run the workflow
```

Continuous Integration for Workflows

Set an environment variable during your PR CI steps and run a test `Workflow` to see it in action

```
[.github/workflows/cicd.yaml]
steps:
# GitHub Workflow setup steps...
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: v1.1-rc${{ github.event.number }}
- name: Build and run workflow
  run: IMAGE_NAME=argocon-hera:v1.1-rc${{ github.event.number }}
       python -m my_module.run_workflow

[src/my_module/run_workflow.py]
global_config.image = os.environ.get("IMAGE_NAME", "argocon-hera:v1")
with Workflow(generate_name="my-workflow-") as w:
    # Build and run the workflow
```

Continuous Integration for Workflows

We set `global_config.image` using the environment variable

```
[.github/workflows/cicd.yaml]
steps:
# GitHub Workflow setup steps...
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: v1.1-rc${{ github.event.number }}
- name: Build and run workflow
  run: IMAGE_NAME=argocon-hera:v1.1-rc${{ github.event.number }}
       python -m my_module.run_workflow
```

```
[src/my_module/run_workflow.py]
global_config.image = os.environ.get("IMAGE_NAME", "argocon-hera:v1")
with Workflow(generate_name="my-workflow-") as w:
    # Build and run the workflow
```

Continuous Deployment and Versioning for WorkflowTemplates

Bump the Python version in release steps ⟶

```
[.github/workflows/release.yaml]
steps:
# GitHub Workflow setup steps...
- name: Bump version number
  run: poetry version ${{ github.event.release.tag_name }}
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: ${{ github.event.release.tag_name }}
- name: Build and release workflow
  run: python -m my_module.release_workflow_template
[src/my_module/release_workflow_template.py]
import my_package
global_config.image = f"argocon-hera:v{my_package.__version__}"
with WorkflowTemplate(name=f"my-template-v{version}") as w:
    # Build and create WorkflowTemplate on cluster
```

Continuous Deployment and Versioning for WorkflowTemplates

Build your new release image with the new version

```
[.github/workflows/release.yaml]
steps:
# GitHub Workflow setup steps...
- name: Bump version number
  run: poetry version ${{ github.event.release.tag_name }}
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: ${{ github.event.release.tag_name }}
- name: Build and release workflow
  run: python -m my_module.release_workflow_template
[src/my_module/release_workflow_template.py]
import my_package
global_config.image = f"argocon-hera:v{my_package.__version__}"
with WorkflowTemplate(name=f"my-template-v{version}") as w:
    # Build and create WorkflowTemplate on cluster
```

Continuous Deployment and Versioning for WorkflowTemplates

Run the release script to release a new versioned WorkflowTemplate using your new code image!

```
[.github/workflows/release.yaml]
steps:
# GitHub Workflow setup steps...
- name: Bump version number
  run: poetry version ${{ github.event.release.tag_name }}
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: ${{ github.event.release.tag_name }}
- name: Build and release workflow
  run: python -m my_module.release_workflow_template
[src/my_module/release_workflow_template.py]
import my_package
global_config.image = f"argocon-hera:v{my_package.__version__}"
with WorkflowTemplate(name=f"my-template-v{version}") as w:
    # Build and create WorkflowTemplate on cluster
```

Continuous Deployment and Versioning for WorkflowTemplates

Set `global_config.image` to new package version in release script

→

```
[.github/workflows/release.yaml]
steps:
# GitHub Workflow setup steps...
- name: Bump version number
  run: poetry version ${{ github.event.release.tag_name }}
- name: Build image
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: ${{ github.event.release.tag_name }}
- name: Build and release workflow
  run: python -m my_module.release_workflow_template
[src/my_module/release_workflow_template.py]
import my_package
global_config.image = f"argocon-hera:v{my_package.__version__}"
with WorkflowTemplate (name=f"my-template-v{version}") as w:
    # Build and create WorkflowTemplate on cluster
```

Class defaults and Prebuild hooks

Shorthand image default →

```
global_config.image = f"argocon-hera:v{my_package.__version__}"

global_config.set_class_defaults(
    Script,
    constructor=RunnerScriptConstructor()
)

@global_config.register_pre_build_hook
def set_script_resources(script_template: Script) -> Script:
   if is_gpu_script(script_template):
       script_template.resources = Resources(
           cpu_request=10,
           gpus=4,
           memory_request="50Gi",
           ephemeral_request="50Gi",
       )
   return script_template
```

Class defaults and Prebuild hooks

Set class defaults so you can build off a base for any Hera class!

```python
global_config.image = f"argocon-hera:v{my_package.__version__}"

global_config.set_class_defaults(
    Script,
    constructor=RunnerScriptConstructor()
)

@global_config.register_pre_build_hook
def set_script_resources(script_template: Script) -> Script:
   if is_gpu_script(script_template):
       script_template.resources = Resources(
           cpu_request=10,
           gpus=4,
           memory_request="50Gi",
           ephemeral_request="50Gi",
       )
   return script_template
```

Class defaults and Prebuild hooks

Or use prebuild hooks for more flexibility, without adding boilerplate everywhere!

→

```
global_config.image = f"argocon-hera:v{my_package.__version__}"

global_config.set_class_defaults(
    Script,
    constructor=RunnerScriptConstructor()
)

@global_config.register_pre_build_hook
def set_script_resources(script_template: Script) -> Script:
   if is_gpu_script(script_template):
       script_template.resources = Resources(
           cpu_request=10,
           gpus=4,
           memory_request="50Gi",
           ephemeral_request="50Gi",
       )
   return script_template
```

Data Science Hera Demo

Key Takeaways

01.

Hera can supercharge the Python experience on Argo Workflows

02.

Use the Hera Runner to unlock the full power of Hera

03.

Automate developer setup and CICD

TL;DR:

Use Hera to easily scale your Data Science workloads on Argo Workflows

Connect with us!

GitHub

argoproj-labs/hera

Docs

hera.readthedocs.io

Slack

slack.cncf.io

#hera-argo-sdk

Free stuff!

GitHub repo of previous talks and demos

github.com/pipekit/talk-demos

Free Workflow Metrics for Argo Workflows

pipekit.io/metrics-signup

Chat with us!

Argo/Infrastructure Help & Advice:

- Find us at Booth T33
- Book a free call! 👉

Hera Key Features

01.

Write Python

Business logic contained in native Python functions - no special Workflow logic polluting your functions, making them testable in isolation.

02.

Integrations with Pydantic

Hera automatically deserializes Parameters and Artifacts and performs type validation.

03.

Sensible syntax

No more YAML. Code completion, actual CI checks, etc.

Optimizing Onboarding for Your Organization

Cookiecutter to create new repositories including Workflow boilerplate and `make` targets

QR code here

Optimizing Onboarding for Your Organization

Cookiecutter to create new repositories including Workflow boilerplate and `make` targets

Use an in-house Hera wrapper package to set defaults for your organization

→

[my_org package]

```
WorkflowsService(
    host="https://my-org.argo-workflows.com"
)
```

[user code]

```
from my_org.workflows import (
    WorkflowsService,
)
```

QR code here

Decorator Syntax

Experimental decorators inspired by FastAPI to create DAG and Steps as functions

Making DAGs and Steps locally runnable!

```python
w = Workflow(generate_name="model-training-pipeline-")


@w.set_entrypoint
@w.dag()
def run_training() -> TrainingDAGOutput:
    datasets = load_and_split_dataset()
    scaling = feature_scaling(
        FeatureScalingInput(
            X_train=datasets.X_train,
            X_test=datasets.X_test,
        )
    )
    train_model = model_training_pydantic(
        ModelTrainingInput(
            X_train=scaling.X_train,
            y_train=datasets.y_train,
        )
    )

    return TrainingDAGOutput(model=train_model.model)
```

Decorator Syntax

Hera infers the Task dependencies from Parameter/Artifact passing and builds the DAG for you!

```python
w = Workflow(generate_name="model-training-pipeline-")


@w.set_entrypoint
@w.dag()
def run_training() -> TrainingDAGOutput:
    datasets = load_and_split_dataset()
    scaling = feature_scaling(
        FeatureScalingInput(
            X_train=datasets.X_train,
            X_test=datasets.X_test,
        )
    )
    train_model = model_training_pydantic(
        ModelTrainingInput(
            X_train=scaling.X_train,
            y_train=datasets.y_train,
        )
    )

    return TrainingDAGOutput(model=train_model.model)
```

Decorator Syntax

Functions must take special Hera Pydantic inputs and outputs

Passing parameters and artifacts are treated exactly the same!

```python
w = Workflow(generate_name="model-training-pipeline-")


@w.set_entrypoint
@w.dag()
def run_training() -> TrainingDAGOutput:
    datasets = load_and_split_dataset()
    scaling = feature_scaling(
        FeatureScalingInput(
            X_train=datasets.X_train,
            X_test=datasets.X_test,
        )
    )
    train_model = model_training_pydantic(
        ModelTrainingInput(
            X_train=scaling.X_train,
            y_train=datasets.y_train,
        )
    )

    return TrainingDAGOutput(model=train_model.model)
```