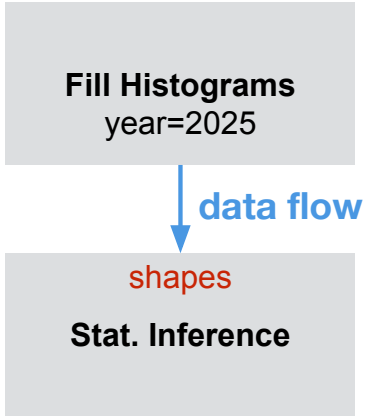
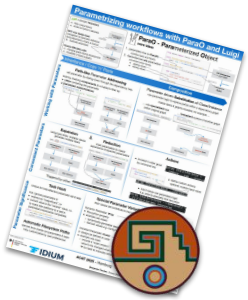



```python
class FillHistograms(Task):
    year = Param[int](default=2025)

class StatInference(Task):
    shapes = Param[FillHistograms]()
```



ParaO: In situ reproducible workflows through user-centric design

Martin Erdmann, Benjamin Fischer, Felix Zinn

PyHEP.dev - Amsterdam
04-09-2026

With funding from the:
Federal Ministry of Research, Technology and Space

RWTH AACHEN UNIVERSITY

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

Immediate Goals

- Eliminate Chores
- Assist Bookkeeping

Induce eachother

“Side Effects”

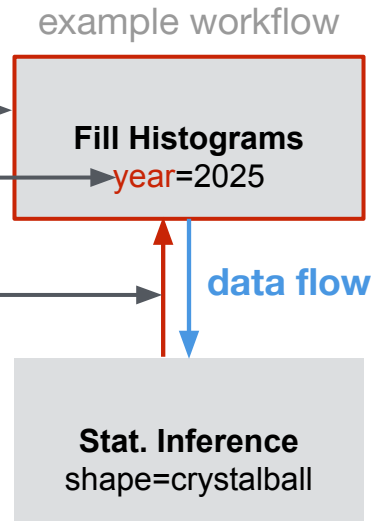
- Reproducibility
- Reusability

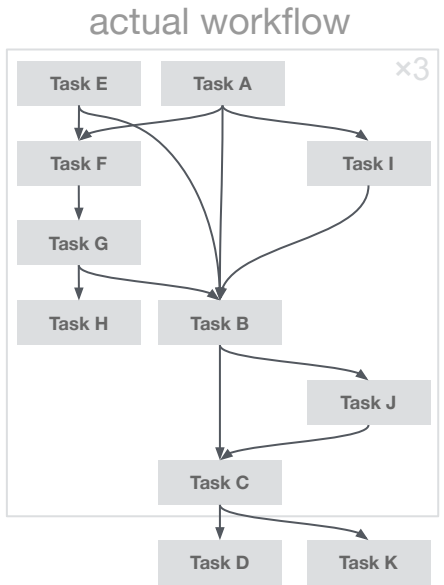
Ingredients

- Task
- Parameter
- Dependency

Primary Hurdles

- Boilerplate
- Workflow Changes





1. **Task(s) – each need**
   - =1 `run` method
     - returns output
   - ≥0 parameter definitions
     - differentiate tasks

```python
class FillHistograms(Task):
    year = Param[int](default=2025)

    def run(self) -> dict[str, Hist]:
        assert self.year > 2020
        return {"signal": ...}
```

2. **Entry point**
   - provide *your* global parameters
     - e.g.: storage directory

```python
if __name__ == "__main__":
    CLI({
        ("output", "dir_base"): "./output",
    }).run()
```

3. **Invoke it**

```
$ ./whatever.py FillHistograms --year=2026
```

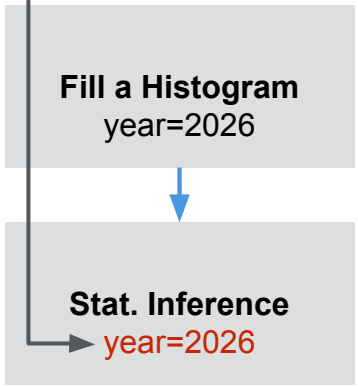
✔ minimal Boilerplate

Dependencies – via Parameters

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

In Luigi

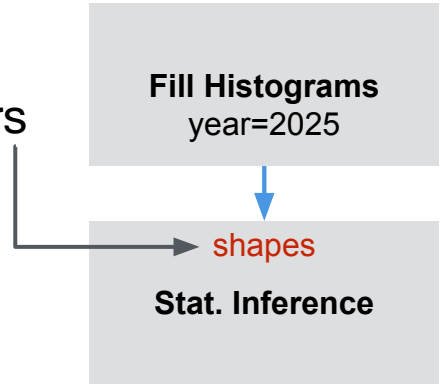
- dependencies via: `requires` method
- parameters need manual redefinition



➜ very cumbersome

With ParaO

- dependencies *are* parameters
  ➜ *composition*
- parameters propagated automatically




```python
class FillHistograms(Task):
    year = Param[int](default=2025)

class StatInference(Task):
    shapes = Param[FillHistograms]()

    def run(self):
        self.shapes.year
        self.year  # also works
```

```
(variable) year: int
```

Difference between ParaO & Task

ParaO: **Para**meterized **O**bject

- basic building block
- similar to a python `dataclass`

➜ generally ephemeral

Task: inherits from ParaO

- step in a workflow
  - defined by `run()` method
- output stored (e.g. to disk)
  - can be loaded

➜ effectively persistent

```python
from parao import ParaO, Task, Param

class ArrayExporter(Task): ...

class Trainer(ParaO):
    optimizer = Param[str]("AdamW")
    learning_rate = Param[float](1e-3)
    warmup_epochs = Param[int](3)
    schedule = Param[str]("cos")

    def train(self, model, data): ...

class DNNTraining(Task):
    input_data = ParaO[ArrayExporter]()
    trainer = ParaO[Trainer]()

    def run(self):
        data = self.input_data.load()
        model = self.make_model()

        self.trainer.train(model, data)
```

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

How to change parameter **down the line** of dependencies?

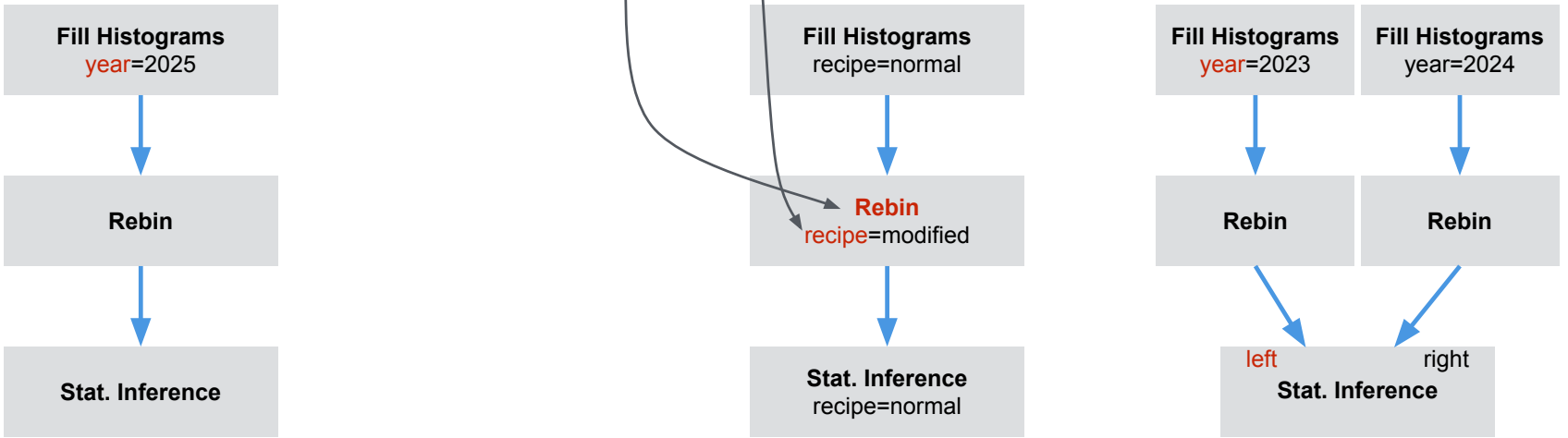
➜ dial down through dependencies **tree**, by **chaining** address components:

class containing the parameter OR parameter name/instance

```
StatInference({"year": 2025})
StatInference(year=2025)
```

```
StatInference({
    (Rebin, "recipe"): "modified",
})
```

```
StatInference({
    ("left", "year"): 2023,
})
```



Parameter driven Substitution

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

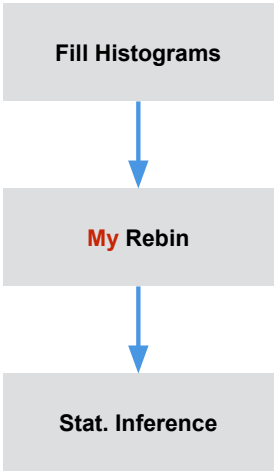
- modify the dependency graph
  - by substitution of dependencies

➜ reusable tasks & entrie graphs

```
class StatInference(ParaO):
    histograms = Param[Rebin]()
```

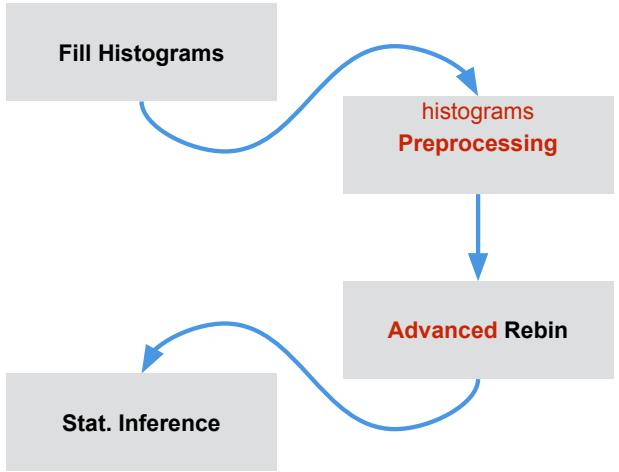
use **specialized** task within original graph

```
StatInference({Rebin: MyRebin})
```



transplant a piece of another graph

```
StatInference({
    Rebin: AdvancedRebin,
    (Preprocessing, "histograms", "__class__"): FillHistograms
})
```



RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

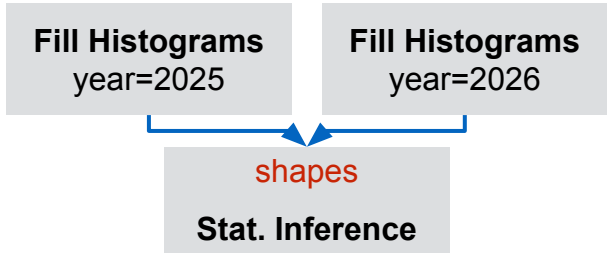
Parameter Expansion

- detect oversupplied value via type hints
  e.g. `list[int]` provided for `int` Parameter
- expands into one Task per entry, then:
  - collected when explicitly allowed
  - run them in sequence

```
$ ./whatever.py FillHistograms --year=2025,2026
```

➜ useful when testing/scanning variants

```
class StatInference(Task):
    shapes = Param[FillHistograms](
        collect=["year"]
    )
```




```
StatInference(year=[2025, 2026])
```

Automatic hash/path: from name and significant parameters (including nested)

Example: `StatInference(year=2026).run.output.path`

➜ `./output/StatInference/2026/result_2f68….pkl`

… much more

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

in-situ workflows bring

- reproducibility, less chores, …
  ➜ even for early chaotic stages
- requires low friction

ParaO: **Para**meterized **O**bjects

- accommodates changes
- versatile parameter mechanics
- explicitly addresses limitations in predecessor tools i.e. Luigi

Ready to use:

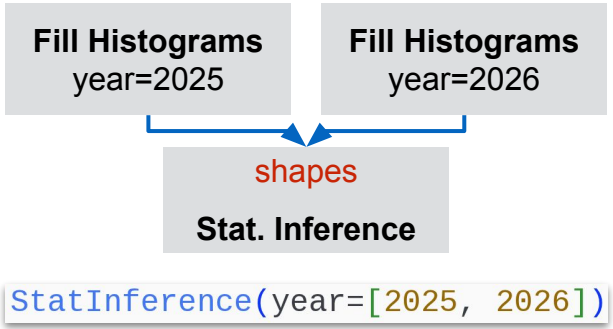
- `pip install parao`
- gitlab.com/bfis/parao
- feedback appreciated

```python
class FillHistograms(Task):
    year = Param[int](default=2025)

class StatInference(Task):
    shapes = Param[FillHistograms]()

    def run(self):
        self.shapes.year
        self.year  # also works
```

(variable) year: int




```python
StatInference(year=[2025, 2026])
```

Backup

What is this all about?

ParaO:

- presented python package: **Para**meterized **O**bjects

In situ

- use with still developing workflows, different focus & requirements

reproducible

- primary benefit of workflows, when codified correctly

workflows

- anything beyond a single step

through user-centric design

- workflow tool effectiveness scales with engagement
  - must minimize friction with user
- based on 1st & 2nd hand experience with Luigi in CMS analyses of ttH(bb), HH(4b), HH(bbWW), VH(bb), …

Outline

1. Workflows
2. Quickstart
3. Dependencies
4. All the Change
5. Task Graphs
6. Parameters
7. At Once

Workflows change – often!

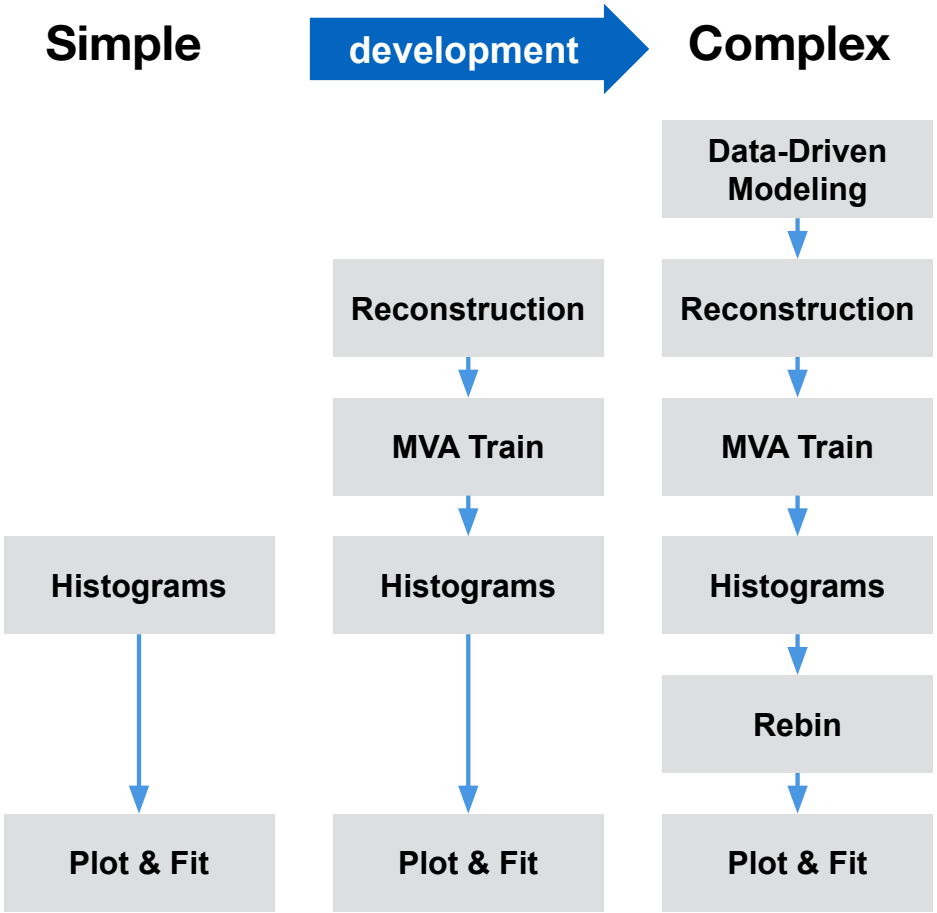
Research is asking *hard* questions

➜ methods & needs change

➜ workflows change a lot

Typical scenarios

- add/remove
  - (intermediate) Tasks
  - Parameters
- change
  - Parameter defaults
  - Task implementation
- test many variants
- …

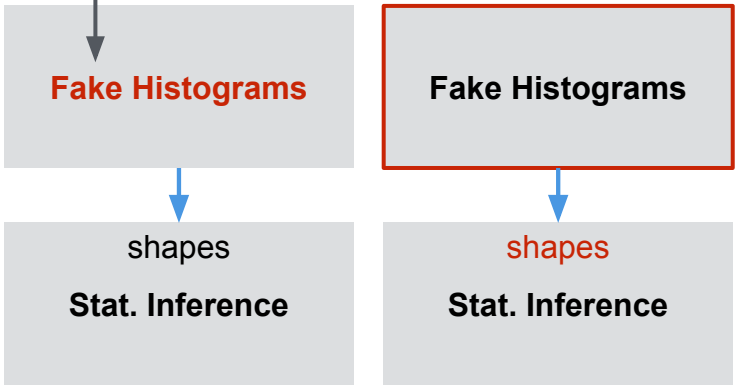


Supporting Taskgraph Changes

Task Substitution

- alter effective taskgraph
- leverage parameter addressing
- at class level *OR* inject instance

```
StatInference({
    FillHistograms: FakeHistograms,
    # or
    "shapes": FakeHistograms(),
})
```



Runtime Parameter Propagation

- covers taskgraph changes, i.e.
  - substitution of Tasks, including:
    - subgraph injection
  - Parameter values, including:
    - runtime calculated
    - complex composites
- affects:
  - “inherited” Parameters
  - Task uniqueness (i.e. a hash)
    - ➜ automatic output storage paths

e.g. `StatInference(year=2026).run.output.path`

`./output/StatInference/2026/result_2f68….pkl`

Parameter Neutral Value

when adding parameter late during development,
it encodes previously hardcoded value

➜ retains access to previous results

```
class FillHistograms(Task):
    num_bins = Param[int](
        neutral=20,
        default=100,
    )
```

Arbitrary Auxiliary Information

- stored on parameter definition
- programmatic access

➜ useful for e.g. hyperparameter scans

```
class TrainDNN(Task):
    learning_rate = Param[float](
        default=1e-3,
        sampler="log_uniform",
        range=(1e-6, 0.1),
    )
```

Const Parameters

- derive uniqueness from hard-coded value

➜ useful when changing implementation

```
class FillHistograms(Task):
    implementation = Const(
        "prod_v3_final2_really"
    )
```

Doing many things – in one step

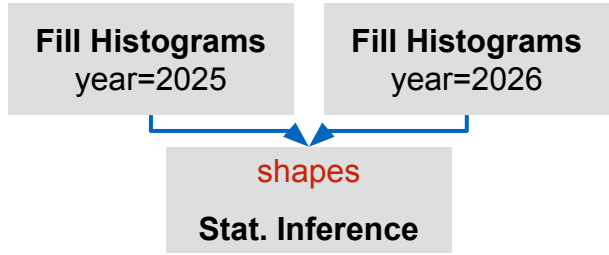
Parameter Expansion

- detect oversupplied value via type hints
  e.g. `list[int]` provided for `int` Parameter
- expands into one Task per entry, then:
  - collected when explicitly allowed
  - run them in sequence

```
$ ./whatever.py FillHistograms --year=2025,2026
```

➜ useful when testing/scanning variants

```
class StatInference(Task):
    shapes = Param[FillHistograms](
        collect=["year"]
    )
```




```
StatInference(year=[2025, 2026])
```

CLI Chaining: initiate multiple action at once – processed in order

- Example: *remove* previous results for two Tasks and *run* the latter

```
$ ./whatever.py Reconstruction --remove=0 FillHistograms --remove=0 --run
```

➜ useful when debugging, especially with multiple involved Tasks

Luigi's strength: **Parameters**

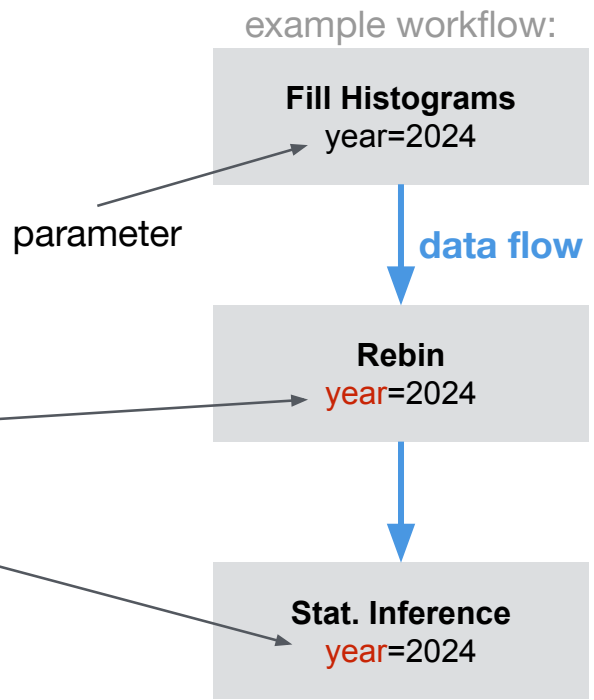
- fairly unique feature
- similar to Python's dataclass

BUT: core design issue

- manual propagation of parameter definitions along dependency chains
  - ➜ duplicate parameters

Causes difficulties with:

- changing graph (during development)
- reusability of tasks & graphs



RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

ParaO

Parameterized **o**bject

1. parameters live on **ParaO**s
   - workflow tasks are ParaOs
2. dependencies **are** parameters
   - access like values

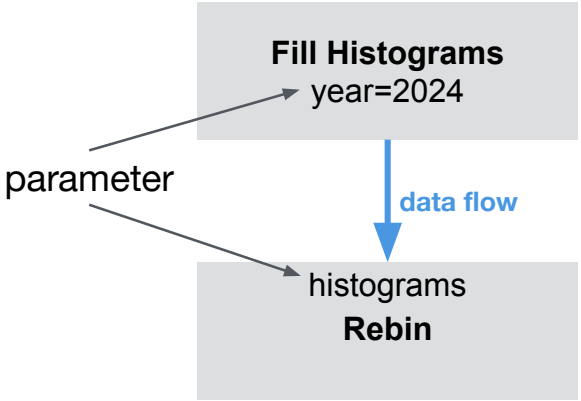
```python
from parao import ParaO, Parameter

class FillHistograms(ParaO):
    year: int = Parameter(default=2024)

class Rebin(ParaO):
    histograms: FillHistograms = Parameter()

    def run(self):
        self.histograms.year  # explicit
        self.year  # short-hand (automatic)

Rebin(year=2025)
```



set & get values with convenience

dependencies expressed via
inheritance ➜ composition

Parameter driven Substitution: Details

Substitute by class

- subject must be subclass of “key”
  - also also of non-ParaO keys
- replacement can match later keys

```
StatInference({Rebin: MyRebin})
```

Substitute at Parameter

- affects specified parameter
- can substitute either:
  - **type**: class used
    - show here
  - **value**: explicit instance
    - just like any other parameter

```
class Preprocessing(ParaO):
    histograms = Param[MakeHistos]()

StatInference({
    Rebin: AdvancedRebin,
    (Preprocessing, "histograms", "__class__"): FillHistograms
})
```

Advanced Parameters: Expansion

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

Use case: User wants to run workflow for several parameter values, at once.

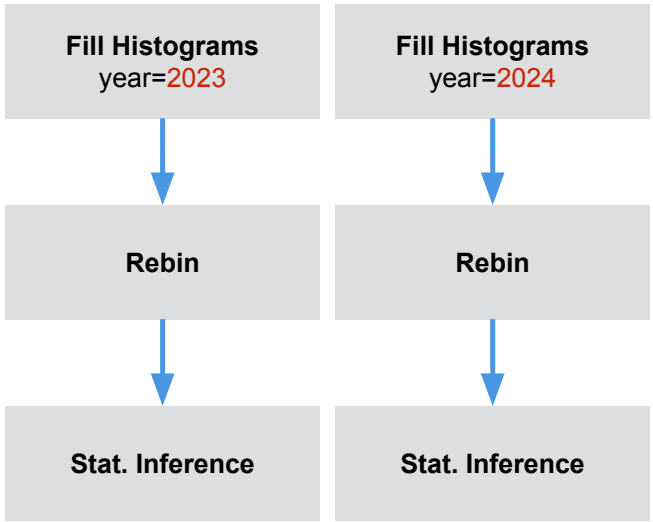
➜ generate them on the fly!

```
year: int = Parameter(default=2024)
```

+

```
parao StatInference --year=2023,2024
```

=



Ingredients:

1. parameter with defined type
2. provide that parameter
   list of values of desired type

Consequence:

- command line: executes them all
- in code: special `Expansion` exception
  - handle manually, or with *Reduction*

Advanced Parameters: Reduction

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

Use case: Some Tasks can handle multiple (similar inputs) at once.

```
year: int = Parameter(default=2024)
```

+

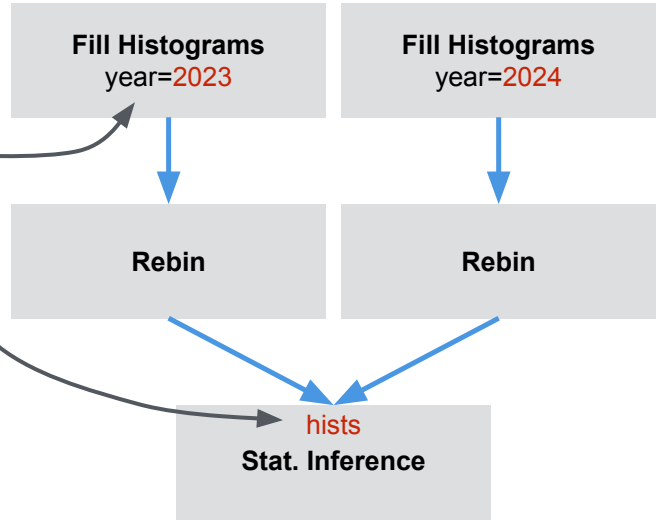
```
parao StatInference --year=2023,2024
```

=

➜ mark Parameter

- to collect Expansions
- can be filtered/limited

```
hists: Rebin = Parameter(collect={"year"})
```



- also works with programmatically triggered Expansion, e.g. via:

```
StatInference(year=[2023, 2024])
```

Advanced Parameters: Actions

Use case: Perform auxiliary actions with particular task (& their dependencies).

```
parao StatInference \
--remove --run --status
```

- Implementation: `Action`
  - special parameter subclass
  - decorate a function with it
- further specialized variants
  - `ValueAction`: pass parameter value to the called function
  - `RecursiveAction`: parameter value controls recursion
- useful for bookkeeping, e.g.: print status, remove output, …

```python
class StatInference(Para0):
    @ValueAction
    def iterate(self, n: int = 1):
        ...
```

```
parao StatInference --iterate=5
```

```
parao StatInference --remove=all \
--FillHistograms.remove=no
```

RWTH AACHEN UNIVERSITY
Benjamin Fischer, 04-09-2026

Task Hash

- unique for effective parameters
  - including dependencies!
- ignores parameters if:
  - marked as “insignificant”
  - current value is the “neutral” value, i.e.
    - backwards compatibility when adding parameters during development

```
class FillHistograms(ParaO):
    use_dask: bool = Parameter(insignificant=True)
    n_bins = Parameter(default=100, neutral=20)
```

Automatic Filesystem Paths

- unique through task hash
- can include parameters explicitly
  - when marked, or even heuristically
- fully customizable by user preferences

```
class Trainer(ParaO):
    optimizer = Param[str]("AdamW", pos=0)
    learning_rate = Param[float](1e-3)
    warmup_epochs = Param[int](3, neutral=0)
    schedule = Param[str]("cos", pos=2)
```

Special Parameter Variants

Dynamic Parameter: **Prop**

- like python property
- example use cases:
  - dynamically generated dependencies
  - derived values e.g. config file hash

```python
class StatInference(ParaO):
    use_leptons: bool = Parameter()

    @Prop
    def histograms(self):
        if self.use_leptons:
            return [
                Rebin(channel=ch)
                for ch in ["e", "mu"]
            ]
        else:
            return Rebin(channel="jets")
```

Constant Parameter: **Const**

- always as defined in code
- example use case:
  - encode a “code version”:
    value to be changed when code changed
    ➜ no need to manually remove old output!

```python
class Rebin(ParaO):
    code_version = Const(1)
```

Both still contribute to the Task Hash!

Summary

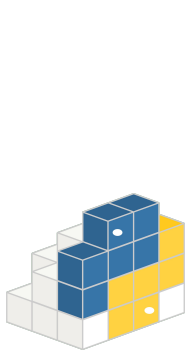
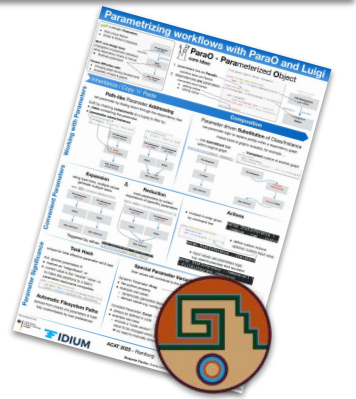
- Core Principle:
  - ParaO: Parameterized Object
  - dependencies via parameter
- Parameters …
  - … have path-like addressing
  - … can substitute, expand & reduce
  - … specializations: Action, Const, Prop
  - … enable: task hash & automatic paths
- Still in early development:
  - `pip install parao`
  - gitlab.com/bfis/parao
  - feedback appreciated!

```python
from parao import ParaO, Parameter

class FillHistograms(ParaO):
    year: int = Parameter(default=2024)

class Rebin(ParaO):
    histograms: FillHistograms = Parameter()

    def run(self):
        self.histograms.year  # explicit
        self.year  # short-hand (automatic)

Rebin(year=2025)
```

```
parao StatInference --remove=all \
--FillHistograms.remove=no
```