PYCMOR

Empowering scientists in CMORization

Pavan Siligam, *January 23, 2026*

Download PDF

What is CMORization?

The process of converting climate model output data into a standardized format, in accordance with rules set by the Climate Model Intercomparison Projects (CMIP) and CF Conventions, ensuring consistency in variable-names, units, metadata, and file structure for easier analysis and comparison across different models.

For `pycmor` it means to provide compliance for the following

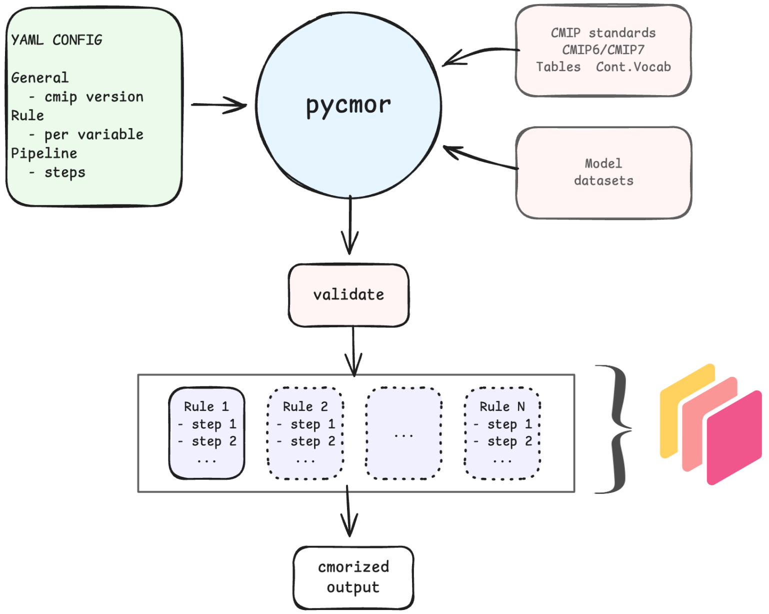
CF Conventions

- Ensure metadata is set
  - coordinate attributes
  - variable attributes
  - global attributes

CMIP6/CMIP7 specification + Controlled Vocabulary

- Ensure file structure
  - file(s) naming
  - directory structure
- match variable units
- match temporal resolution
- match dimensions
- bounds (time_bnds, lat_bnds, …)

pycmor workflow



yaml configuration

Entry point

```yaml
general:
  name: "AWI-ESM-1-1-lr PI Control"
  description: "CMOR configuration for AWIESM 1.1 LR"
  maintainer: "pgierz"
  email: "pgierz@awi.de"
  cmor_version: "CMIP7"
pycmor:
  # parallel: True
  warn_on_no_rule: False
  use_flox: True
  dask_cluster: "slurm"
  dask_cluster_scaling_mode: fixed
  fixed_jobs: 1
  # minimum_jobs: 8
  # maximum_jobs: 30
  # You can add your own path to the dimensionless mapping t
  # If nothing is specified here, it will use the built-in o
inherit:
  # Common attributes shared by all rules
  activity_id: CMIP
  institution_id: AWI
  source_id: AWI-CM-1-1-HR
```

section	used for
`general`	cmor_version, CV_Dir, CMIP_Table_Dir, …
`pycmor`	application settings
`inherit`	common attributes shared by all rules
`rules`	**per variable settings**: cmor_variable, compound_name, model_variable, data_path, …
`pipelines`	steps to be executed for each variable (uses **Default pipeline** if not provided)
`jobqueue`	SLURM specific settings for launching workers

Pipeline Steps

Default Steps in Pipeline

PyCMOR provides these standard processing steps.

```
STEPS = (
    "pycmor.core.gather_inputs.load_mfdataset",
    "pycmor.std_lib.generic.get_variable",
    "pycmor.std_lib.add_vertical_bounds",
    "pycmor.std_lib.timeaverage.timeavg",
    "pycmor.std_lib.units.handle_unit_conversion",
    "pycmor.std_lib.attributes.set_global",
    "pycmor.std_lib.attributes.set_variable",
    "pycmor.std_lib.attributes.set_coordinates",
    "pycmor.std_lib.dimensions.map_dimensions",
    "pycmor.core.caching.manual_checkpoint",
    "pycmor.std_lib.generic.trigger_compute",
    "pycmor.std_lib.generic.show_data",
    "pycmor.std_lib.files.save_dataset",
)
```

Adding Custom Steps

1. Create Custom Function

```
# custom_step.py
def custom_step(data, rule):
    # Your custom processing logic
    return data
```

2. Define Complete Pipeline

Important: When adding custom steps, you must specify the **entire pipeline** including both default and custom steps:

```
pipelines:
  - name: my_custom_pipeline
    steps:
      - "pycmor.core.gather_inputs.load_mfdataset"
      - "pycmor.std_lib.generic.get_variable"
      # ... other default steps ...
      - "script://path/to/custom_step.py:custom_step"
      - "pycmor.std_lib.files.save_dataset"
```

3. Key Points

- Custom steps require full pipeline specification
- Mix default steps with your custom functions
- Use `script://path/to/file:function_name` syntax

Coordinate Attributes

Make your coordinates CF-compliant automatically

Eaxmple: no attribute information

```python
import xarray as xr
import numpy as np

# Dataset with coordinates
ds = xr.Dataset(
    { 'tas': (['time', 'lat', 'lon'],
             np.random.rand(10, 90, 180)),
    },
    coords={
    'time': np.arange(10),
    'lat': np.linspace(-89.5, 89.5, 90),
    'lon': np.linspace(0.5, 359.5, 180),
})

print(ds.lat.attrs)
{}
```

`pycmor` provides `set_coordinate_attributes`

In a processing pipeline (`yaml` config):

```yaml
pipelines:
  steps:
    - "load_mfdataset"
    # ... other steps
    - "set_coordinate_attributes" # Add this step
    # ... other steps
```

In scripts:

```python
from pycmor.std_lib.coordinate_attributes import set_

# Now coordinates have CF-compliant metadata
ds = set_coordinate_attributes(ds, rule)
ds['lat'].attrs
{
  'standard_name': 'latitude',
  'units': 'degrees_north',
  'axis': 'Y',
}
```

Units Handling in PyCMOR

🔬 Chemical units (element-aware)

Automatic conversion for complex chemical units

Example: `mmolC → kg`

Previously (manual):

```
mmolC → molC → gC → kgC
   ÷1e3   ×12.0107   ÷1e3
```

Now (automatic):

```
# Live conversion log
2025-03-13 09:06:37 | INFO  | Converting: mmolC/m2/d → kg m-2 s-1
2025-03-13 09:06:37 | DEBUG | Chemical element detected: Carbon
2025-03-13 09:06:37 | DEBUG | Registering: molC = 12.0107 * g
```

✔ No manual molecular-weight handling
✔ CMIP-compliant physical units

📝 Unit configuration

Wrong units in source data?

→ Define correct units in `model_unit` field in `yaml` file

Dimensionless units?

→ Map them explicitly

```
# Conversion-only mappings
# Output always uses cmor_units
so:
  "0.001": g/kg

intpp:        # Primary production by phytoplankton
  "mol m-2 s-1": "molC m-2 s-1"
```

Frequency Analysis & Processing

PyCMOR analyzes **source data frequency** and processes it to meet **CMIP table requirements**.

Time averaging (CMIP-aware):

- Driven by CMIP table frequency string
- **Methods**
  - Instantaneous → `resample( … ).first()`
  - Mean → `resample( … ).mean()`
  - Climatology → `groupby( … ).mean("time")`

Key principles:

- ✅ **Downsampling supported** (no artificial data creation)
  - `Daily → Monthly`, `6-hourly → Daily`, `3-hourly → 6-hourly`

```
>>> # Month start dates with day offsets
>>> month_start_day_offset = np.array([
...     "2020-01-01",
...     "2020-02-01",
...     "2020-03-04",  # ← 3-day offset
...     "2020-04-01",
...     "2020-05-01",
... ]).astype("datetime64[s]")
...
>>> xarray.infer_freq(month_start_day_offset)
None
>>> infer_frequency(month_start_day_offset, return_metadata=True)
Inferred Frequency : M
Median Δ (days)    : 30.50
Regular Spacing    : ✅
Status             : valid   # ← below threshold limit
>>>
>>> infer_frequency(month_start_day_offset, return_metadata=True, strict=True)
Inferred Frequency : M
Regular Spacing    : ❌
Strict Mode        : ✅
```