How to make direct use of ICON without really touching it?

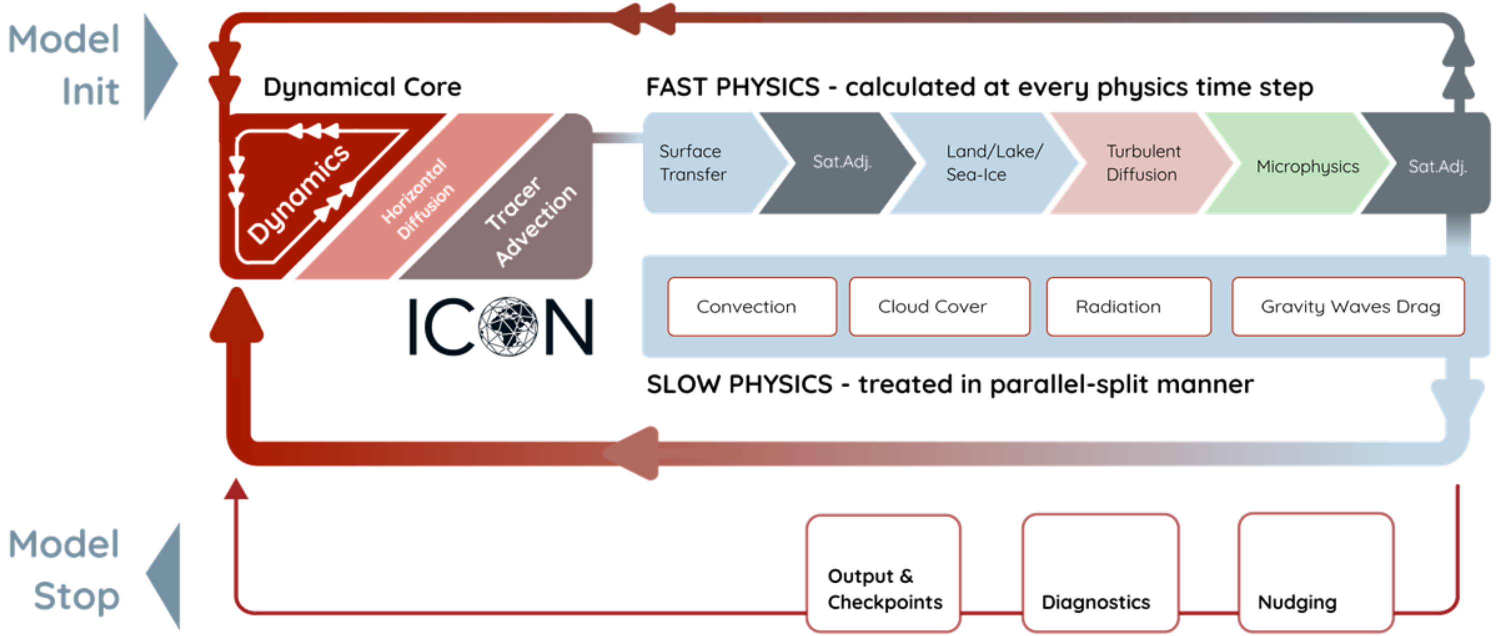
Mahnoosh Haghighatnasab and ComIn Team
DWD, DLR, DKRZ, FZJ | natESM Community Workshop |
18 Feb 2025

ComIn

ICON Community Interface

ICON control flow

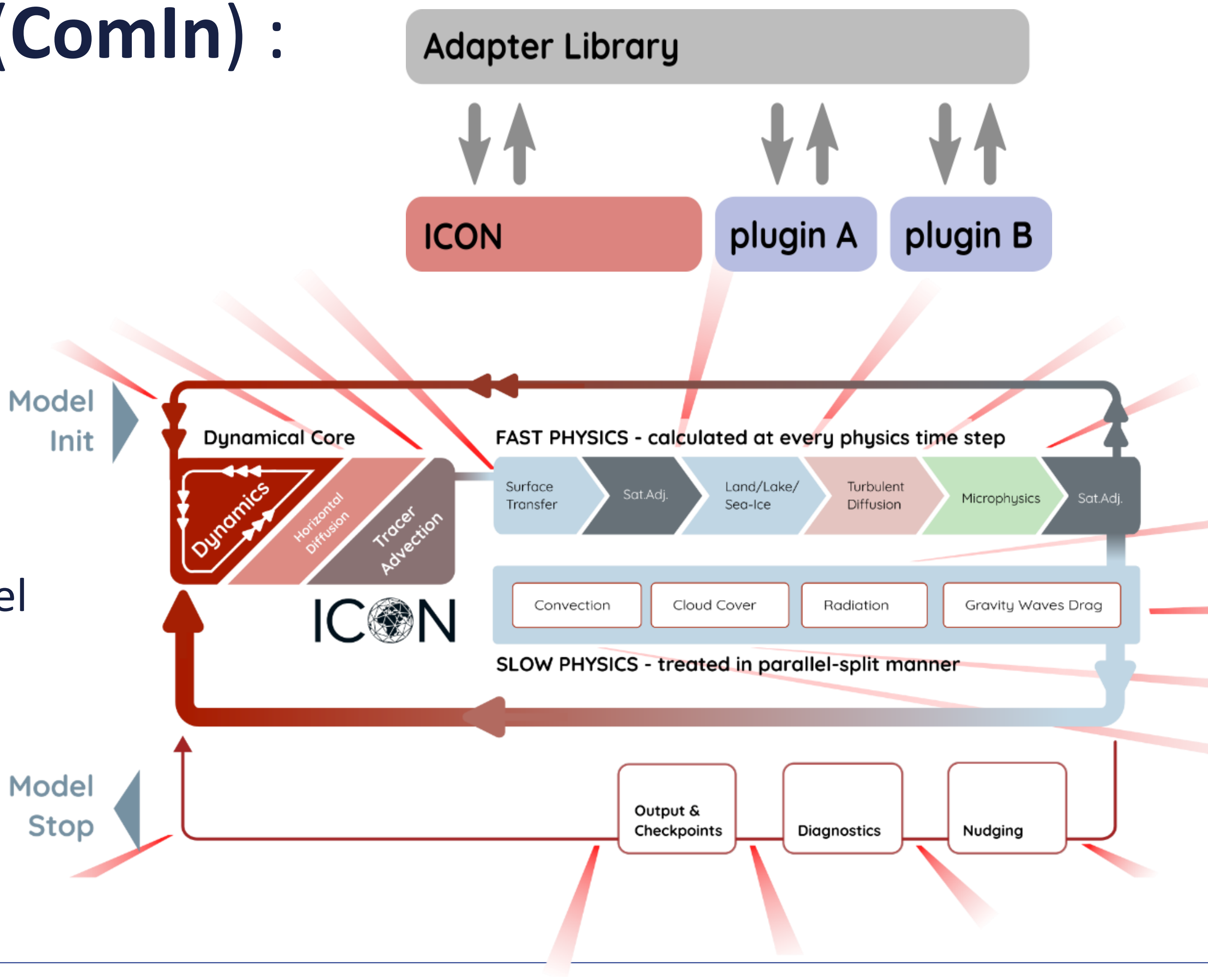
- **Release ICON 2024.10** Fortran and C code: **1,123,740** lines
- Where should my implementations be?
- Going through review process
- Difficult to maintain



Source: F.Prill (DWD)

ICON Community Interface (**ComIn**) :

- Connects plugins to the ICON host model
- Plugin functions are called at pre-defined events (Entry Points)
- Regulates the access and creation of model variables
- Guaranties stable interface for external projects



Source: F.Prill (DWD)

Plugin mechanism for ICON

Behind the scenes: dynamic linking

- ComIn relies on dynamic linking to implement the plugin functionality.
- *Dynamic/shared linking:* operating system loads the necessary shared libraries into memory at runtime

Language interoperability in plugins

- **Shared libraries** for Fortran or C/C++ plugins
- **Python plugins** do not need any compilation process

Comin plugins building blocks

ComIn plugins building blocks

Overview

```
@comin.EP_ATM_WRITE_OUTPUT_BEFORE
def python_diagfct():
    print("diagnostic function called!")
```

append subroutines to a callback register

Overview

```
@comin.EP_ATM_WRITE_OUTPUT_BEFORE
def python_diagfct():
    print("diagnostic function called!")
```

append subroutines to a callback register

```
@comin.EP_SECONDARY_CONSTRUCTOR
def constructor():
    handle = comin.var_get( [entrypoint], (varname, domain_id),
        comin.COMIN_FLAG_WRITE | comin.COMIN_FLAG_SYNC_HALO
    )
    ...
    np.asarray(handle) = some_value
```

read/write access to model variables

Overview

```
@comin.EP_ATM_WRITE_OUTPUT_BEFORE
def python_diagfct():
    print("diagnostic function called!")
```

append subroutines to a callback register

```
@comin.EP_SECONDARY_CONSTRUCTOR
def constructor():
    handle = comin.var_get( [entrypoint], (varname, domain_id),
        comin.COMIN_FLAG_WRITE | comin.COMIN_FLAG_SYNC_HALO
    )
    ...
    np.asarray(handle) = some_value
```

read/write access to model variables

```
comin.var_request_add((varname, domain_id), False)
comin.metadata_set((varname, domain_id), tracer=True)
```

creating additional variables

Overview

```
@comin.EP_ATM_WRITE_OUTPUT_BEFORE
def python_diagfct():
    print("diagnostic function called!")
```

append subroutines to a callback register

```
@comin.EP_SECONDARY_CONSTRUCTOR
def constructor():
    handle = comin.var_get( [entrypoint], (varname, domain_id),
        comin.COMIN_FLAG_WRITE | comin.COMIN_FLAG_SYNC_HALO
    )
    ...
    np.asarray(handle) = some_value
```

read/write access to model variables

```
comin.var_request_add((varname, domain_id), False)
comin.metadata_set((varname, domain_id), tracer=True)
```

creating additional variables

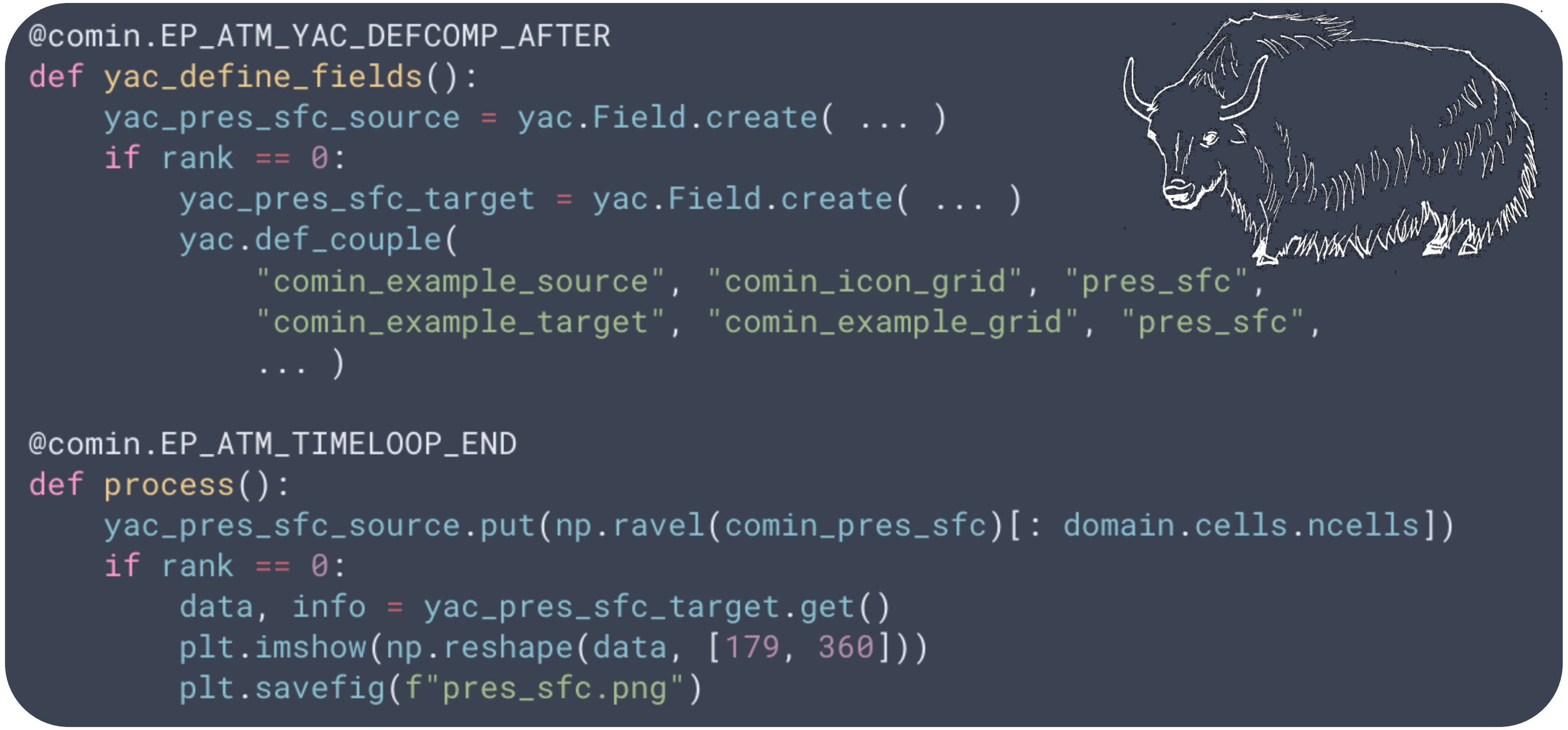
```
domain = comin.descrdata_get_domain(domain_id)
clon = np.asarray(domain.cells.clon)
clat = np.asarray(domain.cells.clat)
```

descriptive data structures

contain information on the ICON setup, the computational grids, and the simulation

Overview

- ComIn is an interface not a coupler
- YAC instances can be used in ComIn Plugins



```
@comin.EP_ATM_YAC_DEFCOMP_AFTER
def yac_define_fields():
    yac_pres_sfc_source = yac.Field.create( ... )
    if rank == 0:
        yac_pres_sfc_target = yac.Field.create( ... )
        yac.def_couple(
            "comin_example_source", "comin_icon_grid", "pres_sfc",
            "comin_example_target", "comin_example_grid", "pres_sfc",
            ... )

@comin.EP_ATM_TIMELOOP_END
def process():
    yac_pres_sfc_source.put(np.ravel(comin_pres_sfc)[: domain.cells.ncells])
    if rank == 0:
        data, info = yac_pres_sfc_target.get()
        plt.imshow(np.reshape(data, [179, 360]))
        plt.savefig(f"pres_sfc.png")
```

Enable the plugin mechanism

- Make sure ICON is configured with `--enable-comin`
- In the Fortran Namelist:

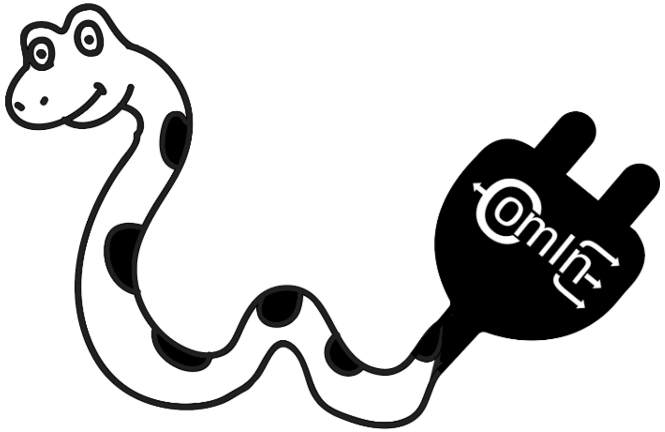
```
&comin_nml
  plugin_list(1)%name           = "comin_plugin"
  plugin_list(1)%plugin_library = "libpython_adapter.so"
  plugin_list(1)%options        = "comin_plugin.py"
/
```

multiple plugins can be added

embedded Python interpreter

filename of Python script passed as an option

Embedded Python, wrapped pointers



- ComIn plugins makes use of an embedded Python interpreter.

- Executes a Python script in the primary constructor

- Field pointers are directly exposed to plugins, wrapped as NumPy arrays
  GPU accelerators: device pointers wrapped by CuPy library

Entry point implementation

- Currently 41 entrypoints
- New entry points can be easily introduced:
  `CALL icon_call_callback(…)`
- **Granularity**: above block loop level, only global variables are exposed
- **ICON calls ComIn, not vice versa!**
  processes in the host model are not switched off, but can only be deactivated using ICON namelist switches

Project history and status

Nov 2022

Project started as a collaboration:
DWD, **DLR-IPA** and **FZJ**

Jan 2023

- **DKRZ** joined the project

Jul 2023

- Design white paper and mock-up completed

Jan 2024

- **ComIn v0.1.0** is part of the first **ICON Open Source Release**

Oct 2024

- **ComIn v0.2.0** released

➢ "ComIn could serve as the potential interface to the wider community"
ICON Dev Meeting, Oct 2024

➢ "ComIn was recognized as a valuable tool by the community."
natESM Newsletter, Issue 5, May 2024

Feb 2025

- Next release ComIn **v0.3.0** (adds data types, C++ refactoring, …)

Jun 2025

- ComIn **v1.0.0**
- Stable Interface

🥳

Nov 2022

Project started as a collaboration:

DWD, **DLR-IPA** and **FZJ**

Jan 2023

- **DKRZ** joined the project

Jul 2023

- Design white paper and mock-up completed

Jan 2024

- **ComIn v0.1.0** is part of the first **ICON Open Source Release**

Oct 2024

- **ComIn v0.2.0** released

- "ComIn could serve as the potential interface to the wider community"
  ICON Dev Meeting, Oct 2024
- "ComIn was recognized as a valuable tool by the community."
  natESM Newsletter, Issue 5, May 2024

Feb 2025

- Next release ComIn **v0.3.0**
(adds data types, C++ refactoring, …)

Jun 2025

- ComIn **v1.0.0**
- Stable Interface

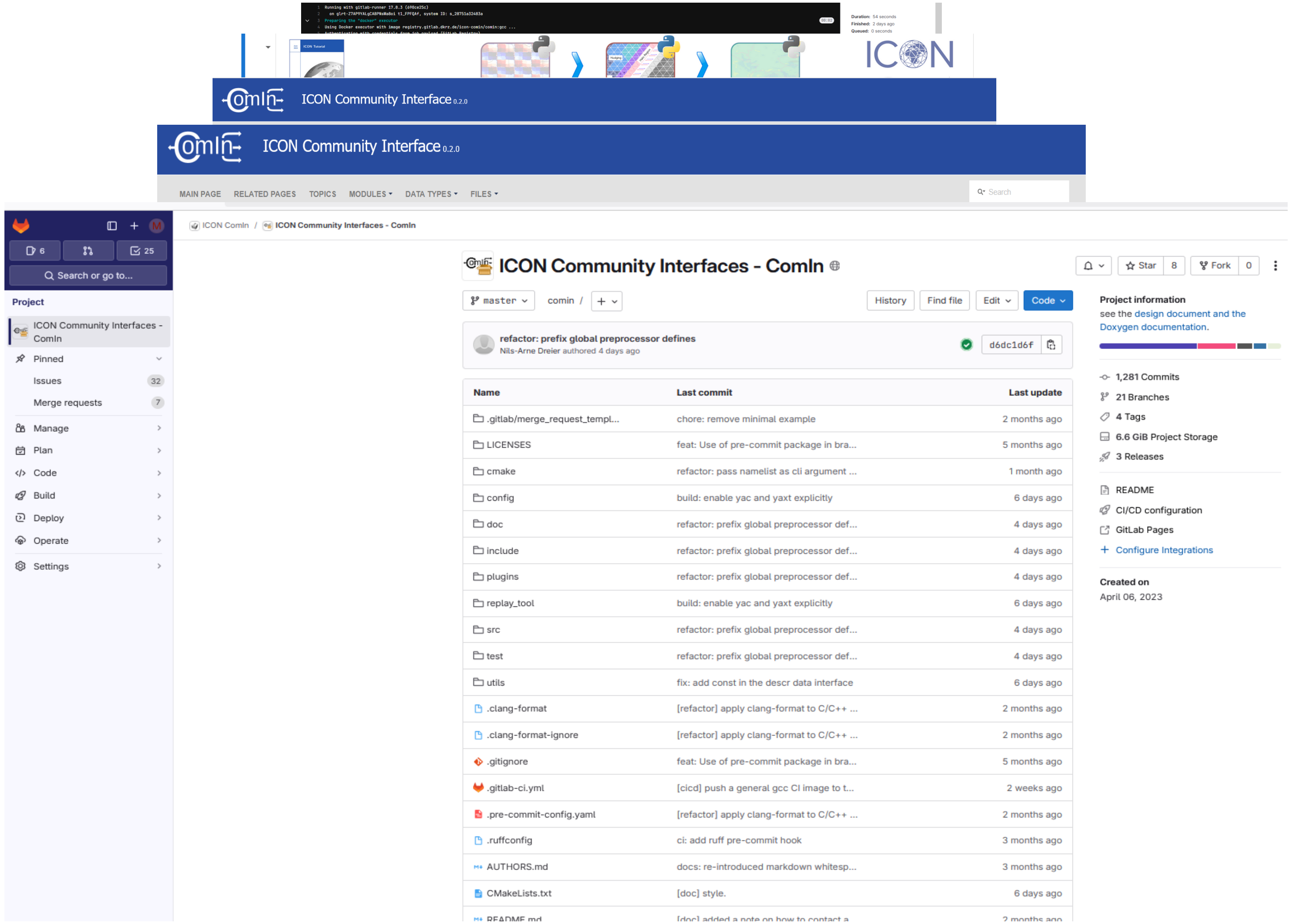
Key Resources

Key Resources

GitLab-Project

https://gitlab.dkrz.de/icon-comin/comin

- Source code and release notes
- Plugin examples
- Tests



ICON Community Interfaces - ComIn

refactor: prefix global preprocessor defines
Nils-Arne Dreier authored 4 days ago

Name	Last commit	Last update
.gitlab/merge_request_templ...	chore: remove minimal example	2 months ago
LICENSES	feat: Use of pre-commit package in bra...	5 months ago
cmake	refactor: pass namelist as cli argument ...	1 month ago
config	build: enable yac and yaxt explicitly	6 days ago
doc	refactor: prefix global preprocessor def...	4 days ago
include	refactor: prefix global preprocessor def...	4 days ago
plugins	refactor: prefix global preprocessor def...	4 days ago
replay_tool	build: enable yac and yaxt explicitly	6 days ago
src	refactor: prefix global preprocessor def...	4 days ago
test	refactor: prefix global preprocessor def...	4 days ago
utils	fix: add const in the descr data interface	6 days ago
.clang-format	[refactor] apply clang-format to C/C++ ...	2 months ago
.clang-format-ignore	[refactor] apply clang-format to C/C++ ...	2 months ago
.gitignore	feat: Use of pre-commit package in bra...	5 months ago
.gitlab-ci.yml	[cicd] push a general gcc CI image to t...	2 weeks ago
.pre-commit-config.yaml	[refactor] apply clang-format to C/C++ ...	2 months ago
.ruffconfig	ci: add ruff pre-commit hook	3 months ago
AUTHORS.md	docs: re-introduced markdown whitesp...	3 months ago
CMakeLists.txt	[doc] style.	6 days ago
README.md	[doc] added a note on how to contact a	2 months ago

Project information
see the design document and the Doxygen documentation.

1,281 Commits
21 Branches
4 Tags
6.6 GiB Project Storage
3 Releases

README
CI/CD configuration
GitLab Pages
Configure Integrations

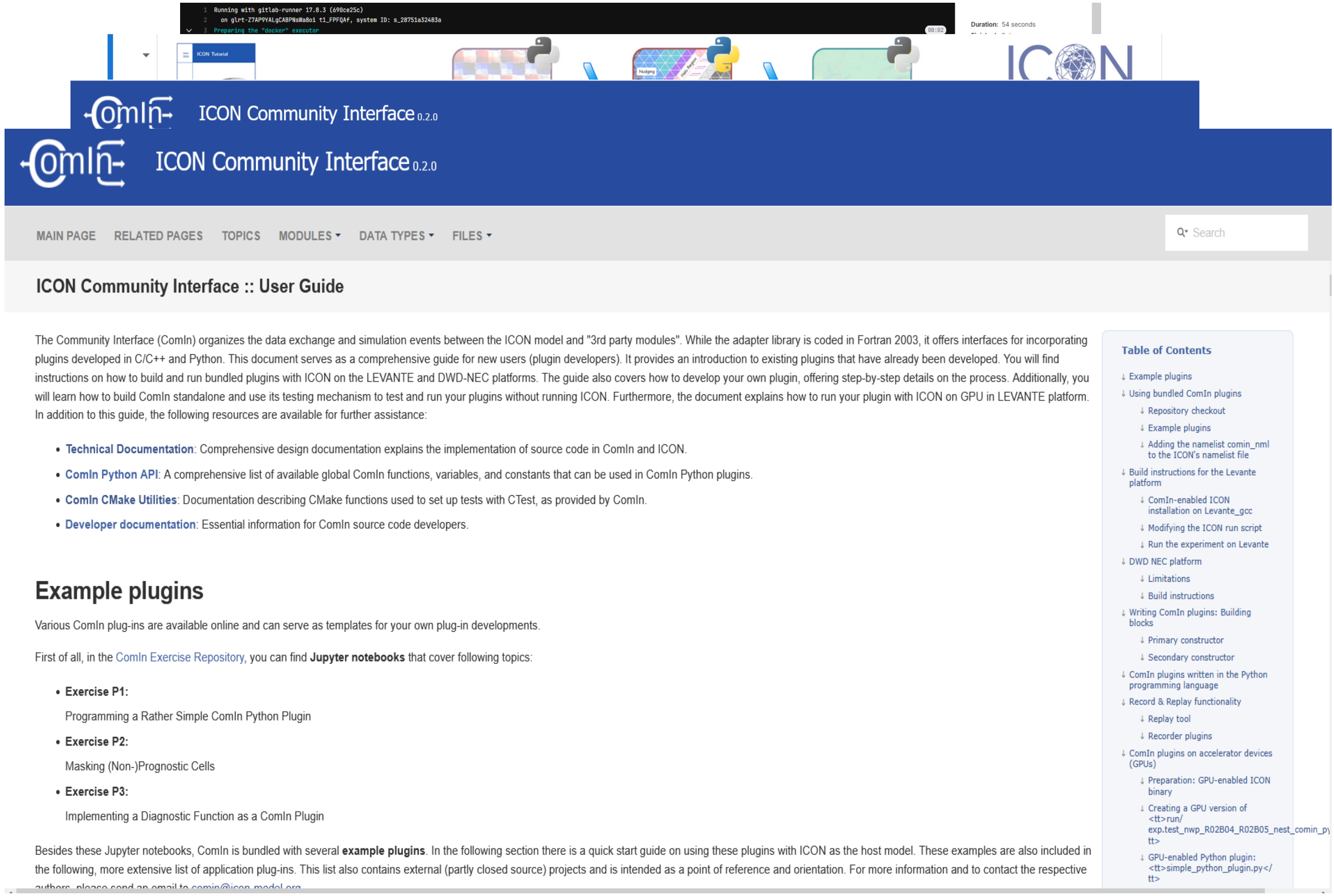
Created on
April 06, 2023

Key Resources

Extensive documentation

https://icon-comin.gitlab-pages.dkrz.de/comin/

- User guide
- Design document
- Developer document
- Doxygen page



ICON Community Interface :: User Guide

The Community Interface (ComIn) organizes the data exchange and simulation events between the ICON model and "3rd party modules". While the adapter library is coded in Fortran 2003, it offers interfaces for incorporating plugins developed in C/C++ and Python. This document serves as a comprehensive guide for new users (plugin developers). It provides an introduction to existing plugins that have already been developed. You will find instructions on how to build and run bundled plugins with ICON on the LEVANTE and DWD-NEC platforms. The guide also covers how to develop your own plugin, offering step-by-step details on the process. Additionally, you will learn how to build ComIn standalone and use its testing mechanism to test and run your plugins without running ICON. Furthermore, the document explains how to run your plugin with ICON on GPU in LEVANTE platform. In addition to this guide, the following resources are available for further assistance:

- Technical Documentation: Comprehensive design documentation explains the implementation of source code in ComIn and ICON.
- ComIn Python API: A comprehensive list of available global ComIn functions, variables, and constants that can be used in ComIn Python plugins.
- ComIn CMake Utilities: Documentation describing CMake functions used to set up tests with CTest, as provided by ComIn.
- Developer documentation: Essential information for ComIn source code developers.

Example plugins

Various ComIn plug-ins are available online and can serve as templates for your own plug-in developments.

First of all, in the ComIn Exercise Repository, you can find **Jupyter notebooks** that cover following topics:

- **Exercise P1:**
  Programming a Rather Simple ComIn Python Plugin
- **Exercise P2:**
  Masking (Non-)Prognostic Cells
- **Exercise P3:**
  Implementing a Diagnostic Function as a ComIn Plugin

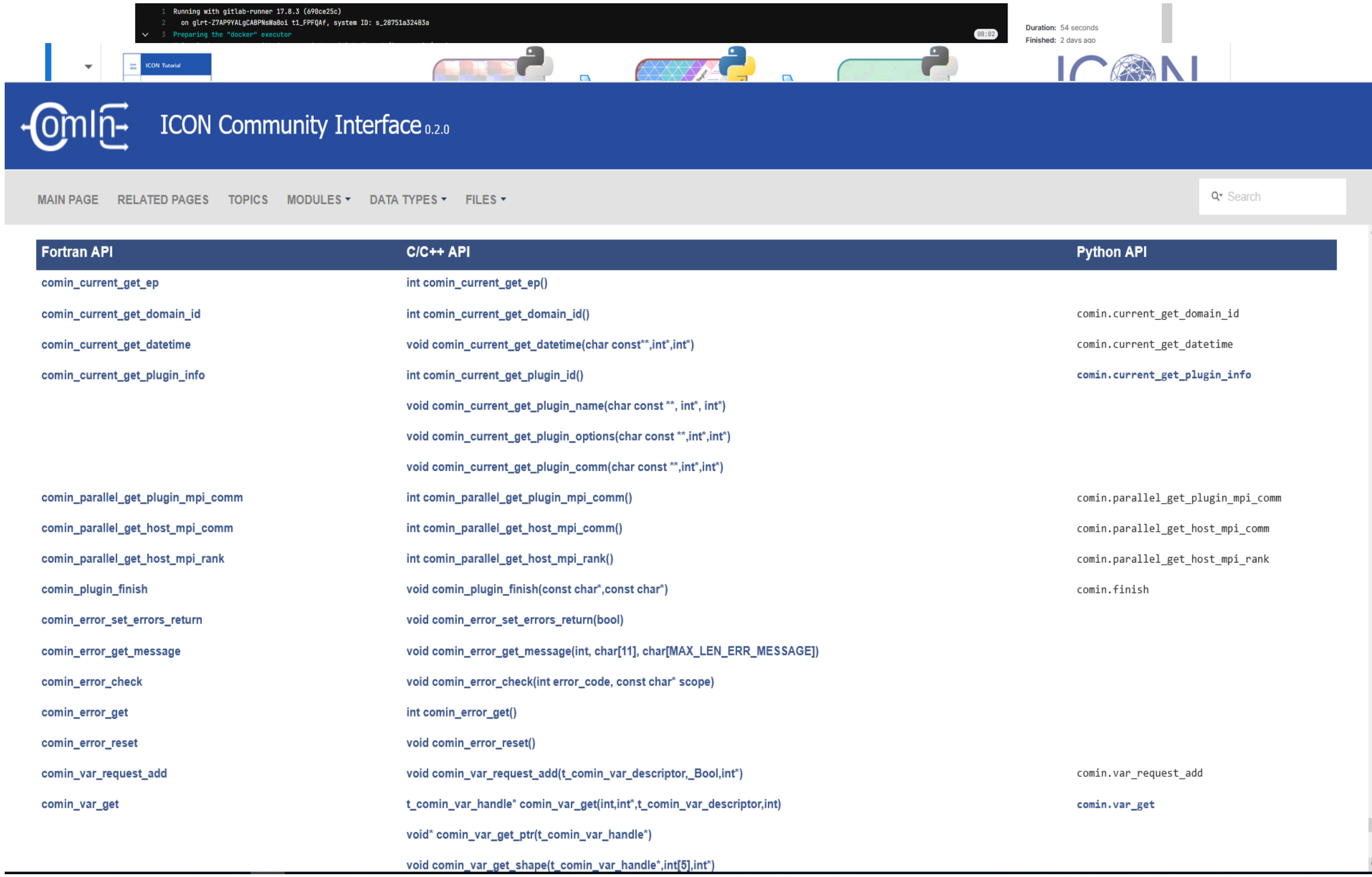
Besides these Jupyter notebooks, ComIn is bundled with several **example plugins**. In the following section there is a quick start guide on using these plugins with ICON as the host model. These examples are also included in the following, more extensive list of application plug-ins. This list also contains external (partly closed source) projects and is intended as a point of reference and orientation. For more information and to contact the respective authors, please send an email to comin@icon-model.org

Table of Contents

Key Resources

Fortran, C/C++ and Python API

- Equivalent interfaces for plugins:
    - Python
    - Fortran
    - C/C++.



```
Running with gitlab-runner 17.8.3 (690ce25c)
  on glrt-Z7AP9YALgCABPNsWa8oi t1_FPFQAf, system ID: s_28751a32483a
Preparing the "docker" executor
```

Duration: 54 seconds
Finished: 2 days ago

ICON Community Interface 0.2.0

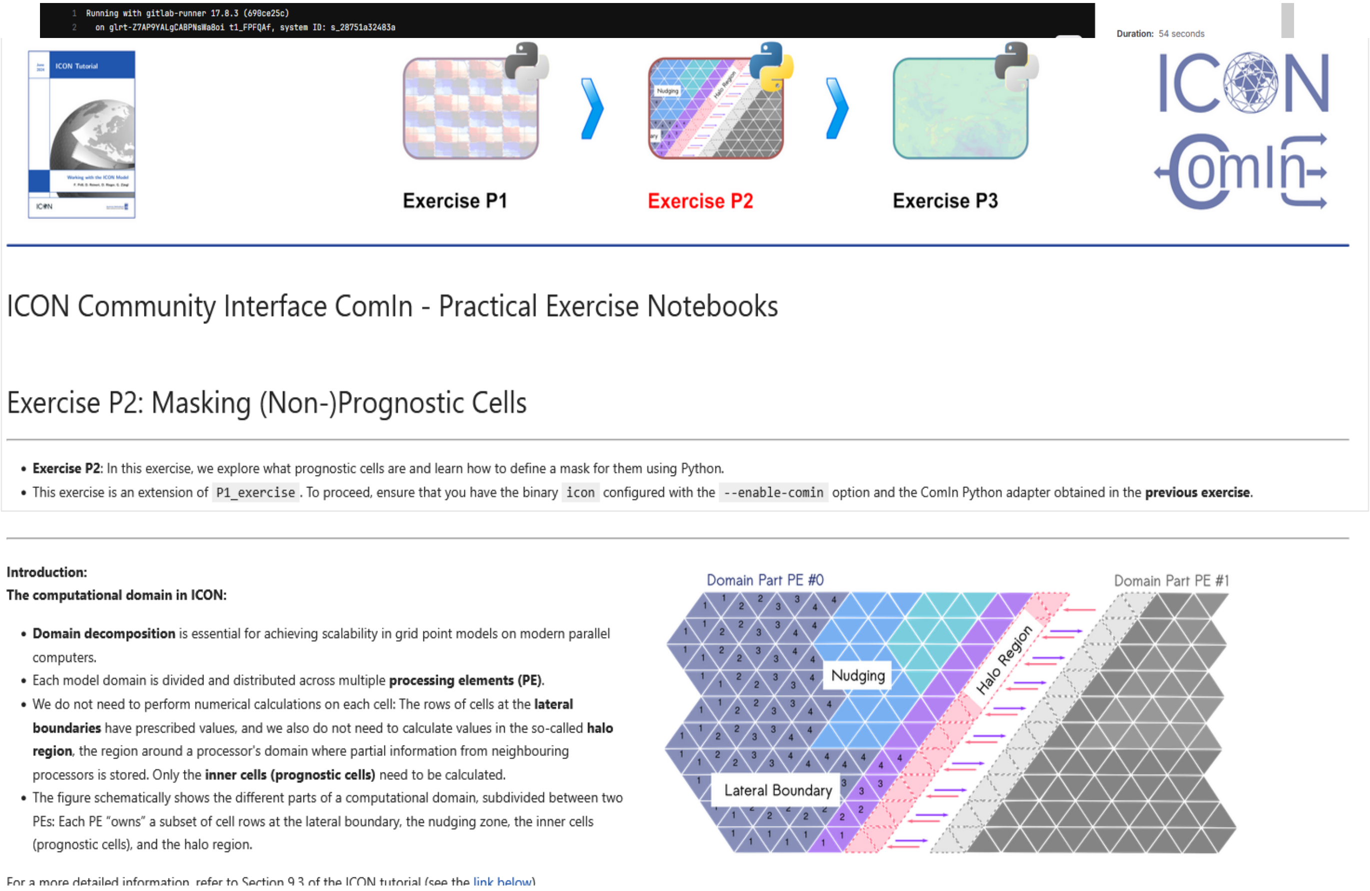
Fortran API	C/C++ API	Python API
comin_current_get_ep	int comin_current_get_ep()	
comin_current_get_domain_id	int comin_current_get_domain_id()	comin.current_get_domain_id
comin_current_get_datetime	void comin_current_get_datetime(char const**,int*,int*)	comin.current_get_datetime
comin_current_get_plugin_info	int comin_current_get_plugin_id()	comin.current_get_plugin_info
	void comin_current_get_plugin_name(char const **, int*, int*)	
	void comin_current_get_plugin_options(char const **,int*,int*)	
	void comin_current_get_plugin_comm(char const **,int*,int*)	
comin_parallel_get_plugin_mpi_comm	int comin_parallel_get_plugin_mpi_comm()	comin.parallel_get_plugin_mpi_comm
comin_parallel_get_host_mpi_comm	int comin_parallel_get_host_mpi_comm()	comin.parallel_get_host_mpi_comm
comin_parallel_get_host_mpi_rank	int comin_parallel_get_host_mpi_rank()	comin.parallel_get_host_mpi_rank
comin_plugin_finish	void comin_plugin_finish(const char*,const char*)	comin.finish
comin_error_set_errors_return	void comin_error_set_errors_return(bool)	
comin_error_get_message	void comin_error_get_message(int, char[11], char[MAX_LEN_ERR_MESSAGE])	
comin_error_check	void comin_error_check(int error_code, const char* scope)	
comin_error_get	int comin_error_get()	
comin_error_reset	void comin_error_reset()	
comin_var_request_add	void comin_var_request_add(t_comin_var_descriptor,_Bool,int*)	comin.var_request_add
comin_var_get	t_comin_var_handle* comin_var_get(int,int*,t_comin_var_descriptor,int)	comin.var_get
	void* comin_var_get_ptr(t_comin_var_handle*)	
	void comin_var_get_shape(t_comin_var_handle*,int[5],int*)	

Key Resources

ComIn Exercise Notebooks

https://gitlab.dkrz.de/icon-comin/comin-training-exercises

- Prepared for Levante platform



```
Running with gitlab-runner 17.8.3 (690ce25c)
  on glrt-Z7AP9YALgCABPNsWa8oi t1_FPFQAf, system ID: s_28751a32483a
```

Duration: 54 seconds

ICON Community Interface ComIn - Practical Exercise Notebooks

Exercise P2: Masking (Non-)Prognostic Cells

- **Exercise P2**: In this exercise, we explore what prognostic cells are and learn how to define a mask for them using Python.
- This exercise is an extension of `P1_exercise`. To proceed, ensure that you have the binary `icon` configured with the `--enable-comin` option and the ComIn Python adapter obtained in the **previous exercise**.

Introduction:
The computational domain in ICON:

- **Domain decomposition** is essential for achieving scalability in grid point models on modern parallel computers.
- Each model domain is divided and distributed across multiple **processing elements (PE)**.
- We do not need to perform numerical calculations on each cell: The rows of cells at the **lateral boundaries** have prescribed values, and we also do not need to calculate values in the so-called **halo region**, the region around a processor's domain where partial information from neighbouring processors is stored. Only the **inner cells (prognostic cells)** need to be calculated.
- The figure schematically shows the different parts of a computational domain, subdivided between two PEs: Each PE "owns" a subset of cell rows at the lateral boundary, the nudging zone, the inner cells (prognostic cells), and the halo region.

For a more detailed information, refer to Section 9.3 of the ICON tutorial (see the link below)

Key Resources

Tests, "record & replay" tool, CICD

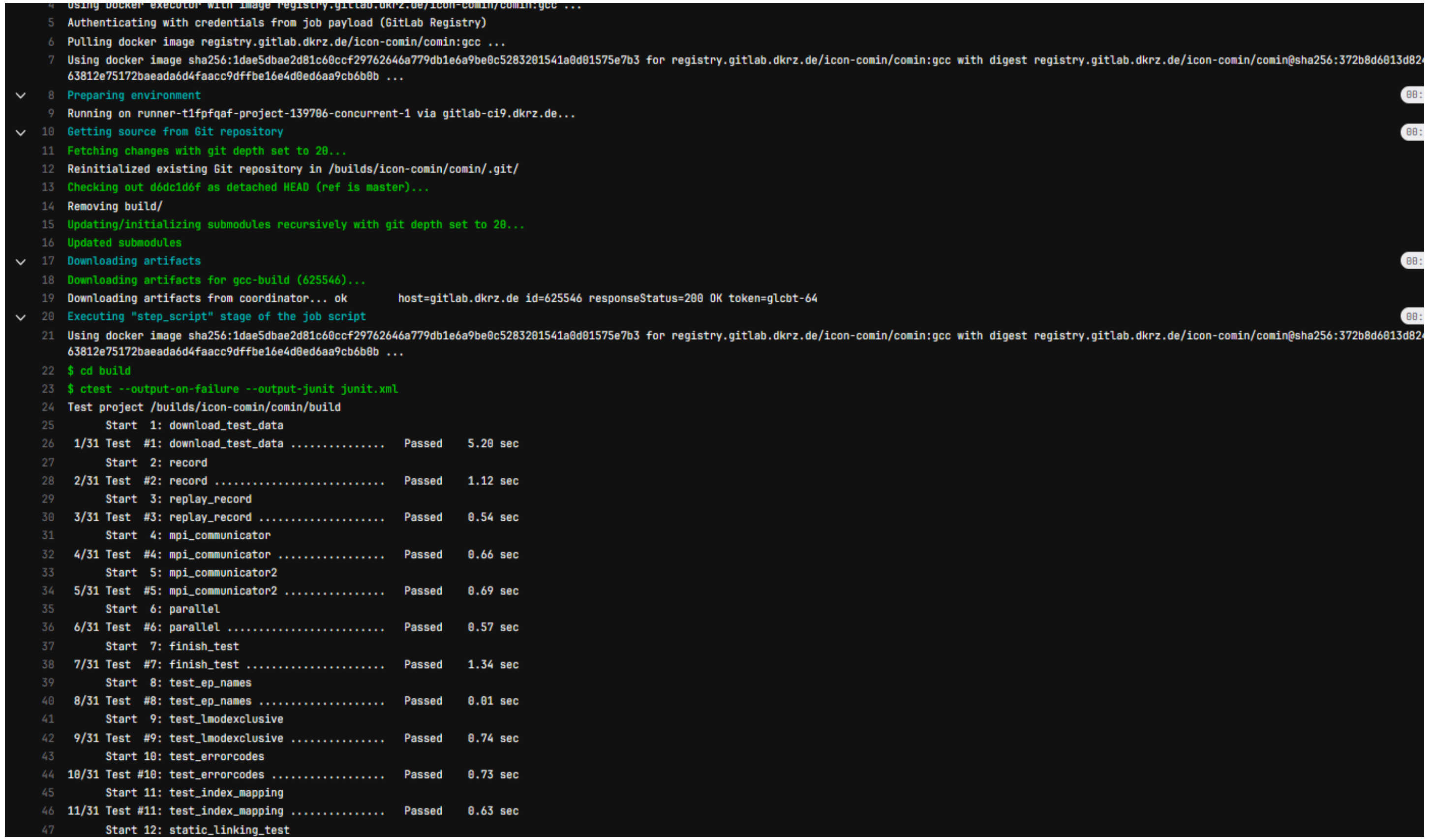
Gitlab pipeline

- ComIn is shipped together with ~30 tests (`ctest` testing framework)

- Makes use of previously recorded ICON datasets ("**record & replay**" tool)

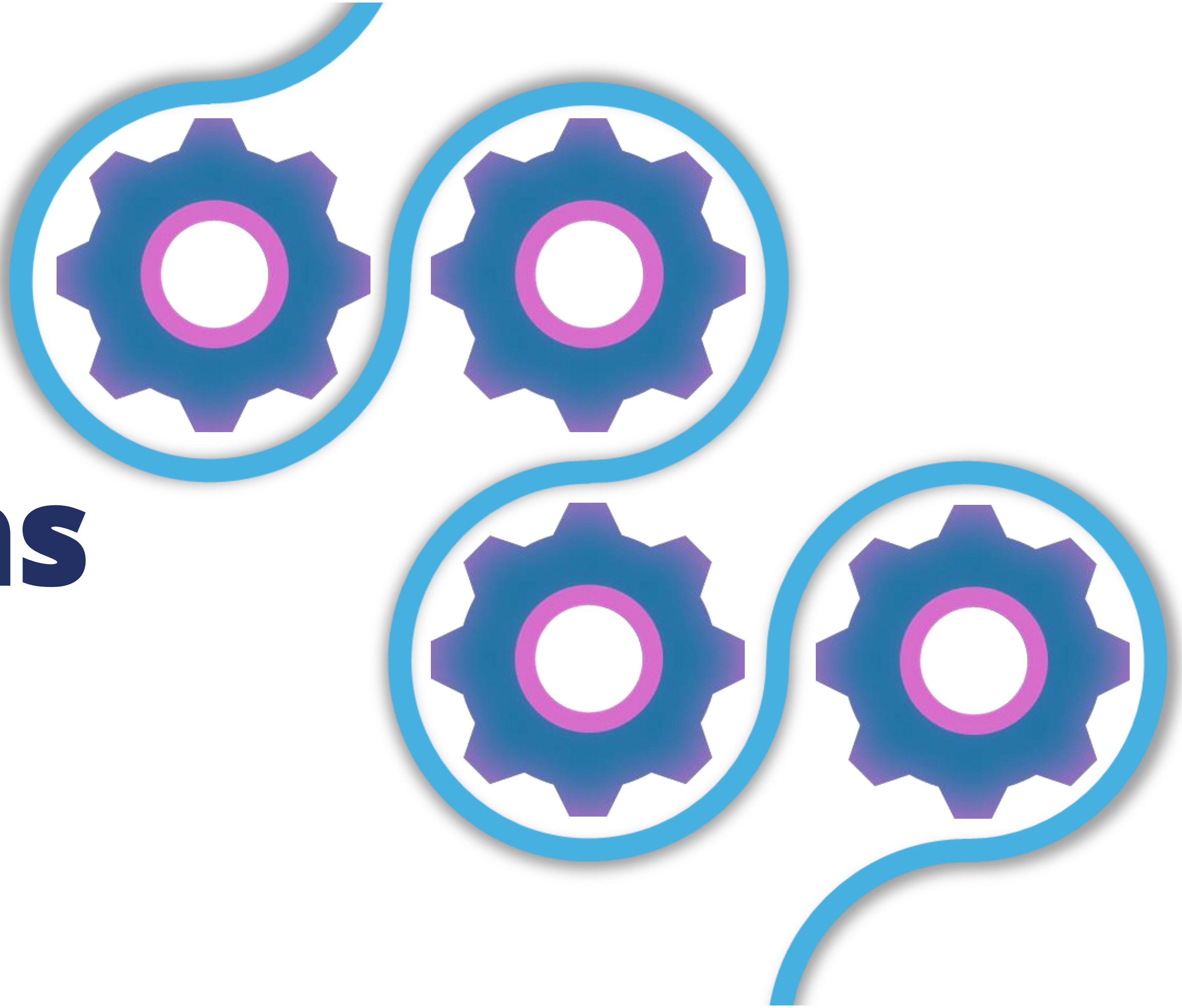
- **record & replay** is very helpful for developing plugins

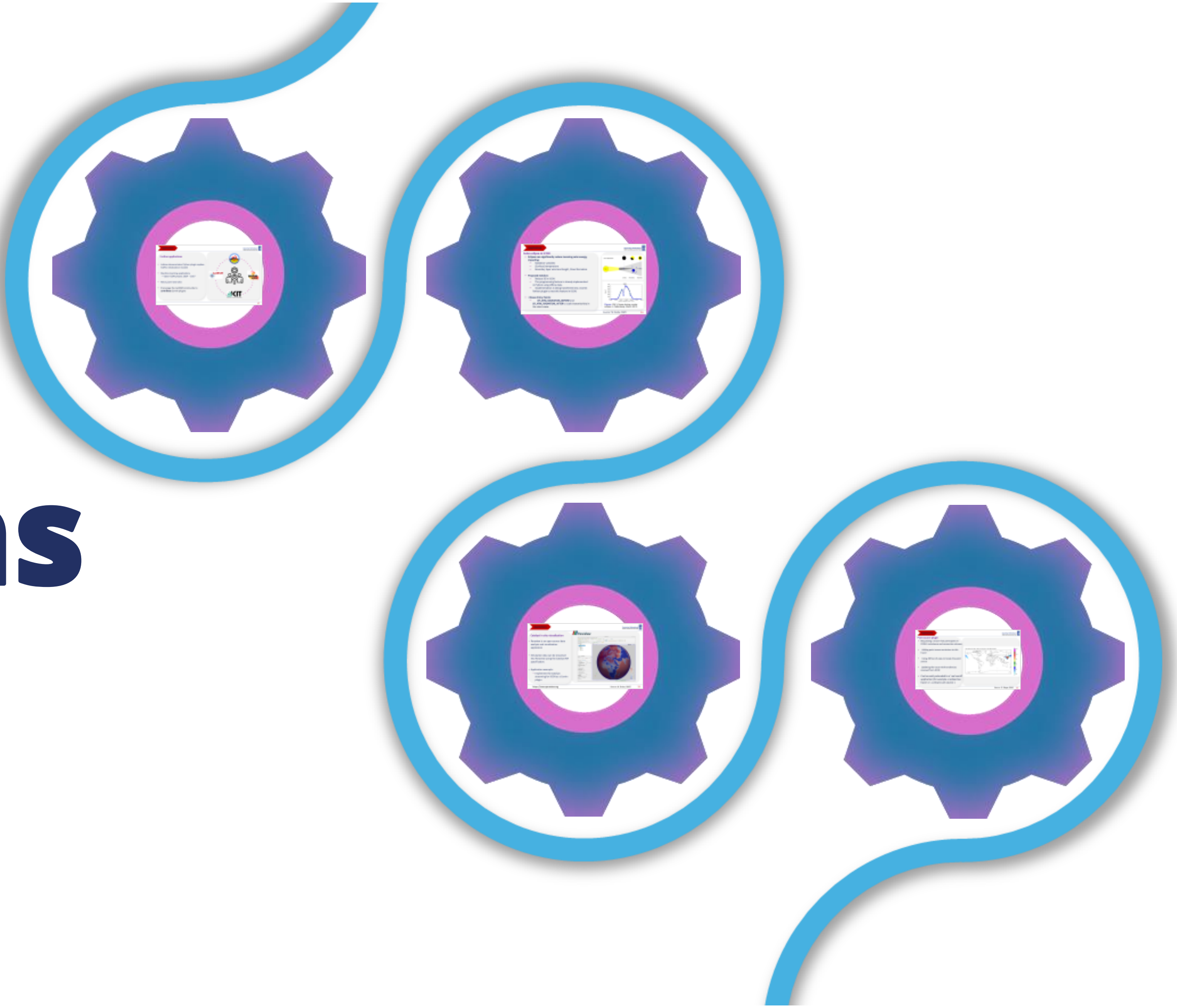
- Gitlab CICD pipeline:
  - automated builds and tests
  - source code formatting
  - compilation of documentation



```
Using docker executor with image registry.gitlab.dkrz.de/icon-comin/comin:gcc ...
Authenticating with credentials from job payload (GitLab Registry)
Pulling docker image registry.gitlab.dkrz.de/icon-comin/comin:gcc ...
Using docker image sha256:1dae5dbae2d81c60ccf29762646a779db1e6a9be0c5283201541a0d01575e7b3 for registry.gitlab.dkrz.de/icon-comin/comin:gcc with digest registry.gitlab.dkrz.de/icon-comin/comin@sha256:372b8d6013d82
63812e75172baeada6d4faacc9dffbe16e4d0ed6aa9cb6b0b ...
Preparing environment
Running on runner-t1fpfqaf-project-139706-concurrent-1 via gitlab-ci9.dkrz.de...
Getting source from Git repository
Fetching changes with git depth set to 20...
Reinitialized existing Git repository in /builds/icon-comin/comin/.git/
Checking out d6dc1d6f as detached HEAD (ref is master)...
Removing build/
Updating/initializing submodules recursively with git depth set to 20...
Updated submodules
Downloading artifacts
Downloading artifacts for gcc-build (625546)...
Downloading artifacts from coordinator... ok        host=gitlab.dkrz.de id=625546 responseStatus=200 OK token=glcbt-64
Executing "step_script" stage of the job script
Using docker image sha256:1dae5dbae2d81c60ccf29762646a779db1e6a9be0c5283201541a0d01575e7b3 for registry.gitlab.dkrz.de/icon-comin/comin:gcc with digest registry.gitlab.dkrz.de/icon-comin/comin@sha256:372b8d6013d82
63812e75172baeada6d4faacc9dffbe16e4d0ed6aa9cb6b0b ...
$ cd build
$ ctest --output-on-failure --output-junit junit.xml
Test project /builds/icon-comin/comin/build
      Start  1: download_test_data
 1/31 Test  #1: download_test_data ...............   Passed    5.20 sec
      Start  2: record
 2/31 Test  #2: record ...........................   Passed    1.12 sec
      Start  3: replay_record
 3/31 Test  #3: replay_record ....................   Passed    0.54 sec
      Start  4: mpi_communicator
 4/31 Test  #4: mpi_communicator .................   Passed    0.66 sec
      Start  5: mpi_communicator2
 5/31 Test  #5: mpi_communicator2 ................   Passed    0.69 sec
      Start  6: parallel
 6/31 Test  #6: parallel .........................   Passed    0.57 sec
      Start  7: finish_test
 7/31 Test  #7: finish_test ......................   Passed    1.34 sec
      Start  8: test_ep_names
 8/31 Test  #8: test_ep_names ....................   Passed    0.01 sec
      Start  9: test_lmodexclusive
 9/31 Test  #9: test_lmodexclusive ...............   Passed    0.74 sec
      Start 10: test_errorcodes
10/31 Test #10: test_errorcodes ..................   Passed    0.73 sec
      Start 11: test_index_mapping
11/31 Test #11: test_index_mapping ...............   Passed    0.63 sec
      Start 12: static_linking_test
```

Applications

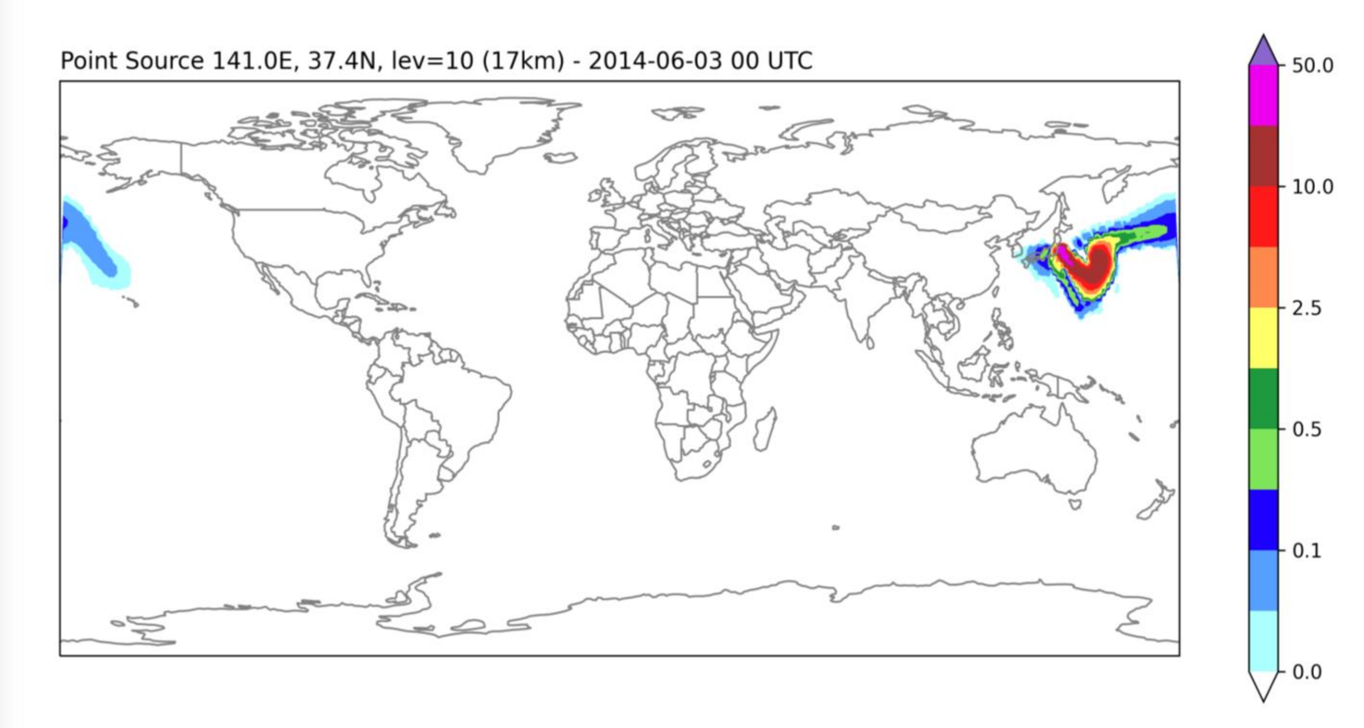




Applications

Point source plugin

- Requesting a tracer that participates in ICON's turbulence and convection scheme
- Adding point source emissions to this tracer
- Using *KDTree* of *scipy* to locate the point source
- Updating the tracer with tendencies received from ICON
- Can be easily extended to a 'real world' application (for example a radioactive tracer or a volcanic ash source)

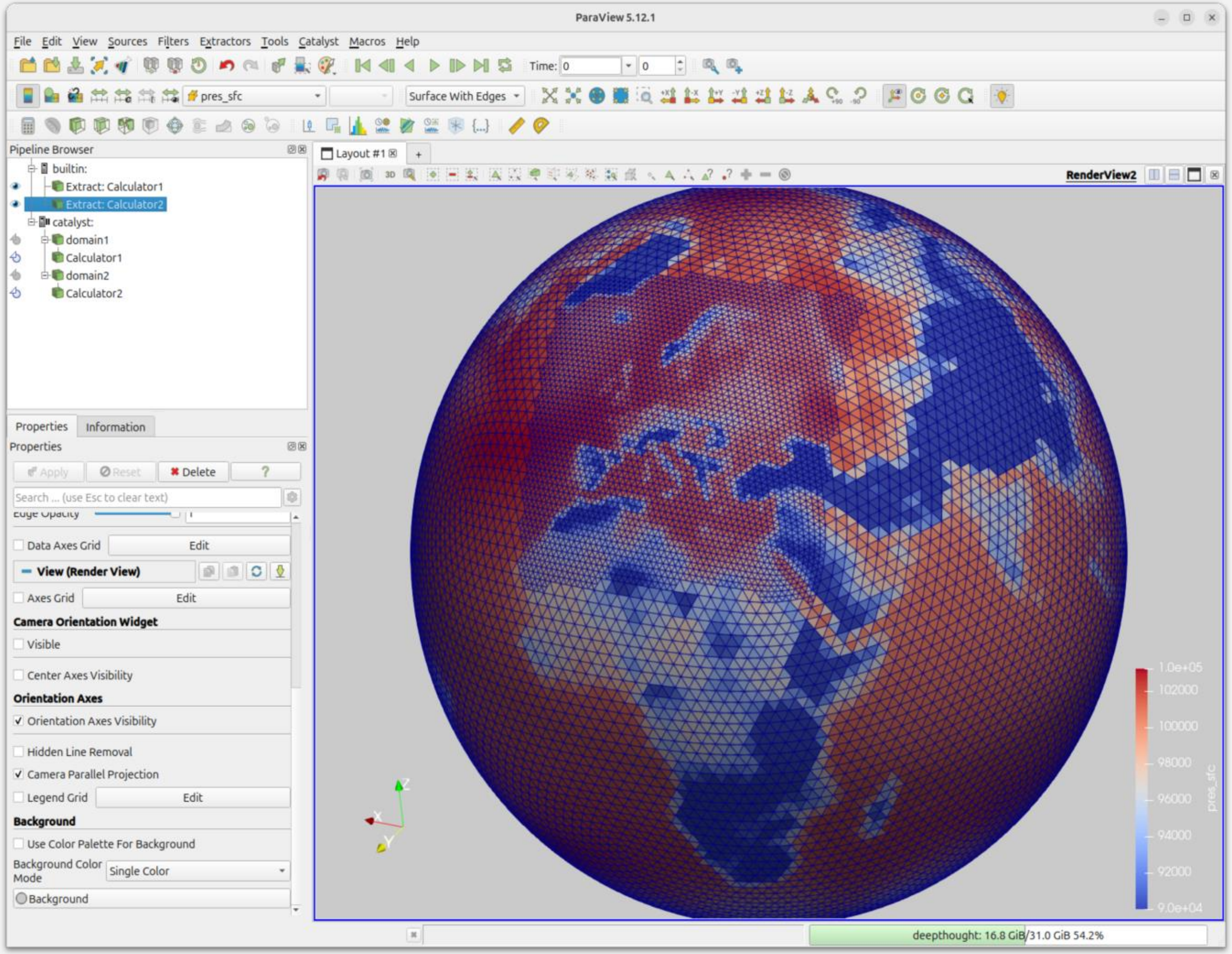


Source : D. Rieger, DWD

Catalyst in situ visualization

- Paraview is an open-source data analysis and visualization application
- Simulation data can be streamed into Paraview using the Catalyst API specification.
- Application example:
  - Implement the Catalyst streaming for ICON as a ComIn plugin.

ParaView



☐ Data Axes Grid

View (Render View)

☐ Axes Grid

Camera Orientation Widget

☐ Visible

☐ Center Axes Visibility

Orientation Axes

☑ Orientation Axes Visibility

☐ Hidden Line Removal

☑ Camera Parallel Projection

☐ Legend Grid

Background

☐ Use Color Palette For Background

Background Color Mode: Single Color

○ Background

deepthought: 16.8 GiB/31.0 GiB 54.2%

https://www.paraview.org

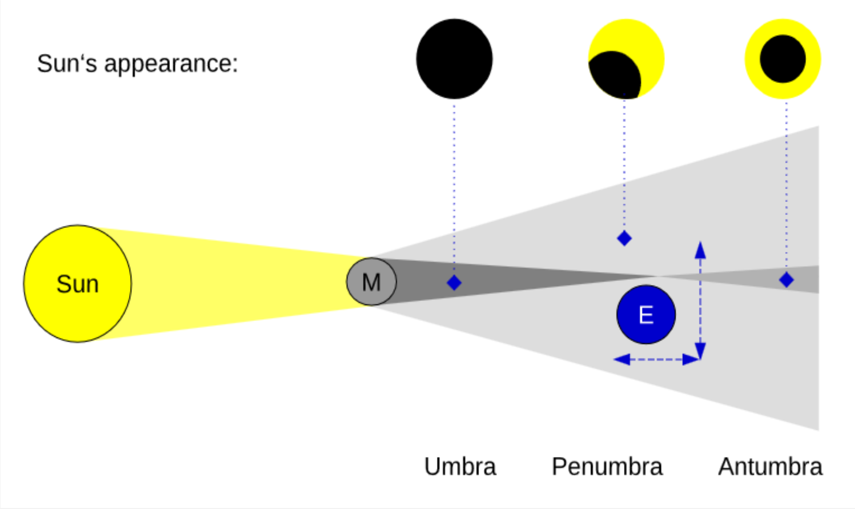
Source: N. Dreier, DKRZ

Solar eclipse in ICON

- **Eclipses can significantly reduce incoming solar energy, impacting**:
  - Radiation variables
  - (Surface) temperature
  - Boundary layer wind and height, Cloud formation

- **Proposed Solution:**
  - Reduce S0 in ICON
  - The programming feature is already implemented in Python using offline data.
  - Implementation is being transferred into a ComIn Python plugin to test this feature in ICON.

- **Chosen Entry Points:**
  - ***EP_ATM_RADIATION_BEFORE*** and ***EP_ATM_RADIATION_AFTER*** to scale transmissivity in the short wave.



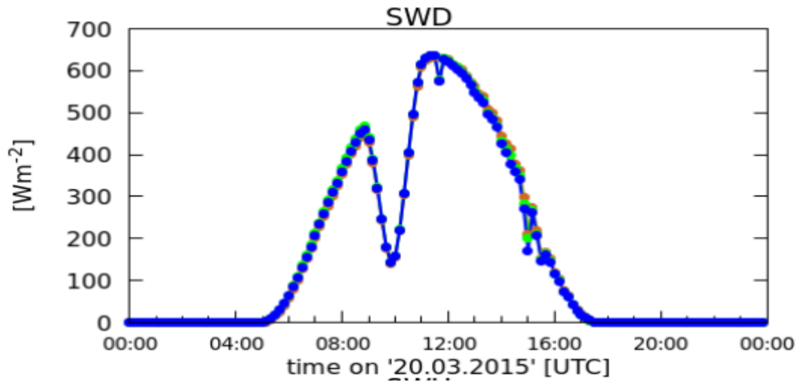


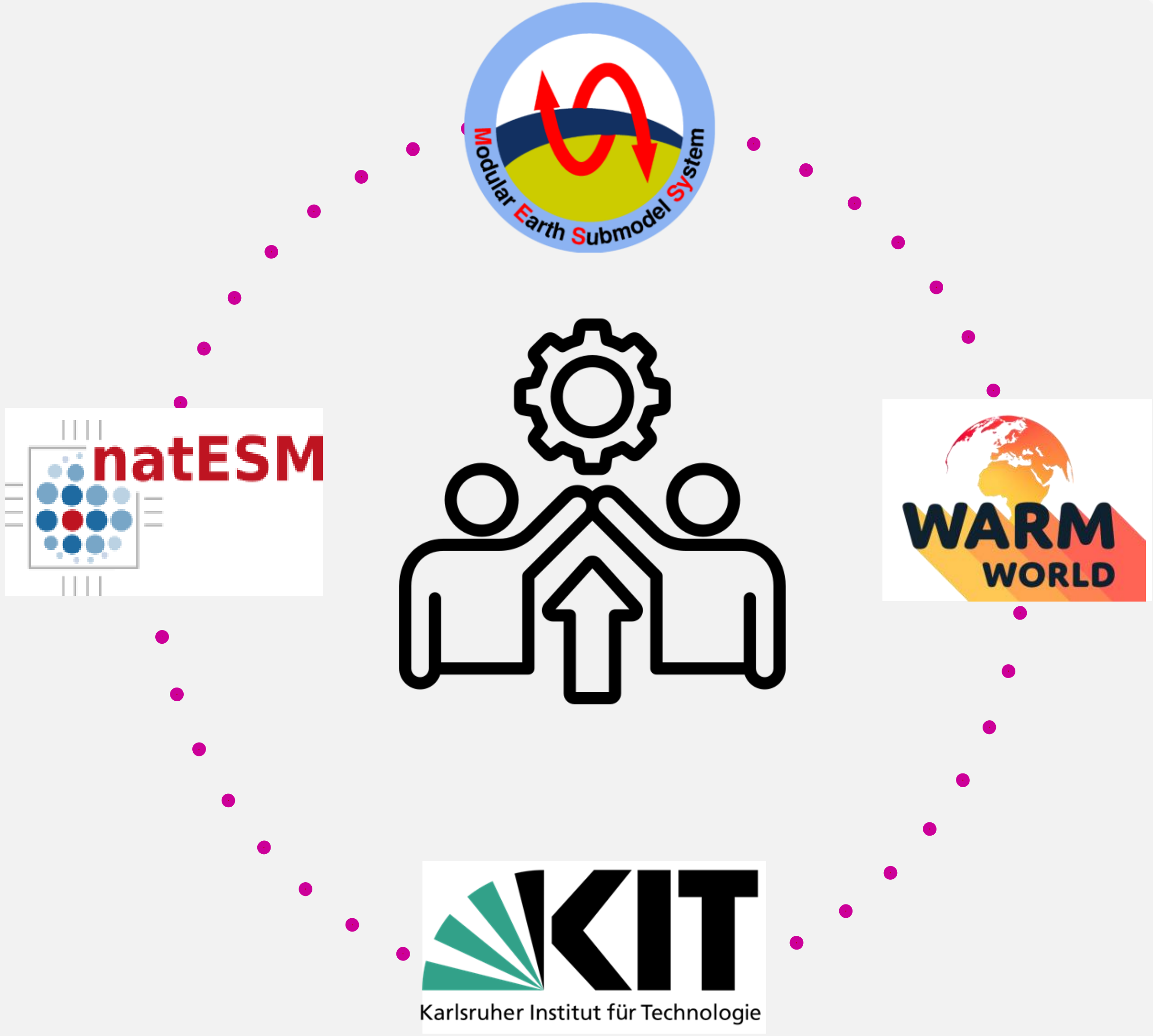
Figure: SW ↓ fluxes during a solar eclipse in Falkenberg, March 2015

Source : M. Burba, DWD

Applications

Further applications

- Initicon demonstrator Python plugin replace ICON’s initialization module
- Machine learning applications
  → Open ICON project 2024 – 2027
- Messy (see next talk)
- Encourage the natESM community to **contribute** ComIn plugins



Team Members

Deutscher Wetterdienst (DWD)

Florian Prill

Mahnoosh
Haghighatnasab

Daniel Rieger

Deutsches Zentrum für Luft- und Raumfahrt (DLR)

Bastian Kern

Kerstin Hartung

Patrick Jöckel

Deutsches Klimarechenzentrum (DKRZ)

Nils-Arne
Dreier

Lakshmi Aparna
Devulapalli

Wilton Jaciel
Loch

Forschungszentrum Jülich (FZJ)

Astrid Kerkweg

Thank You

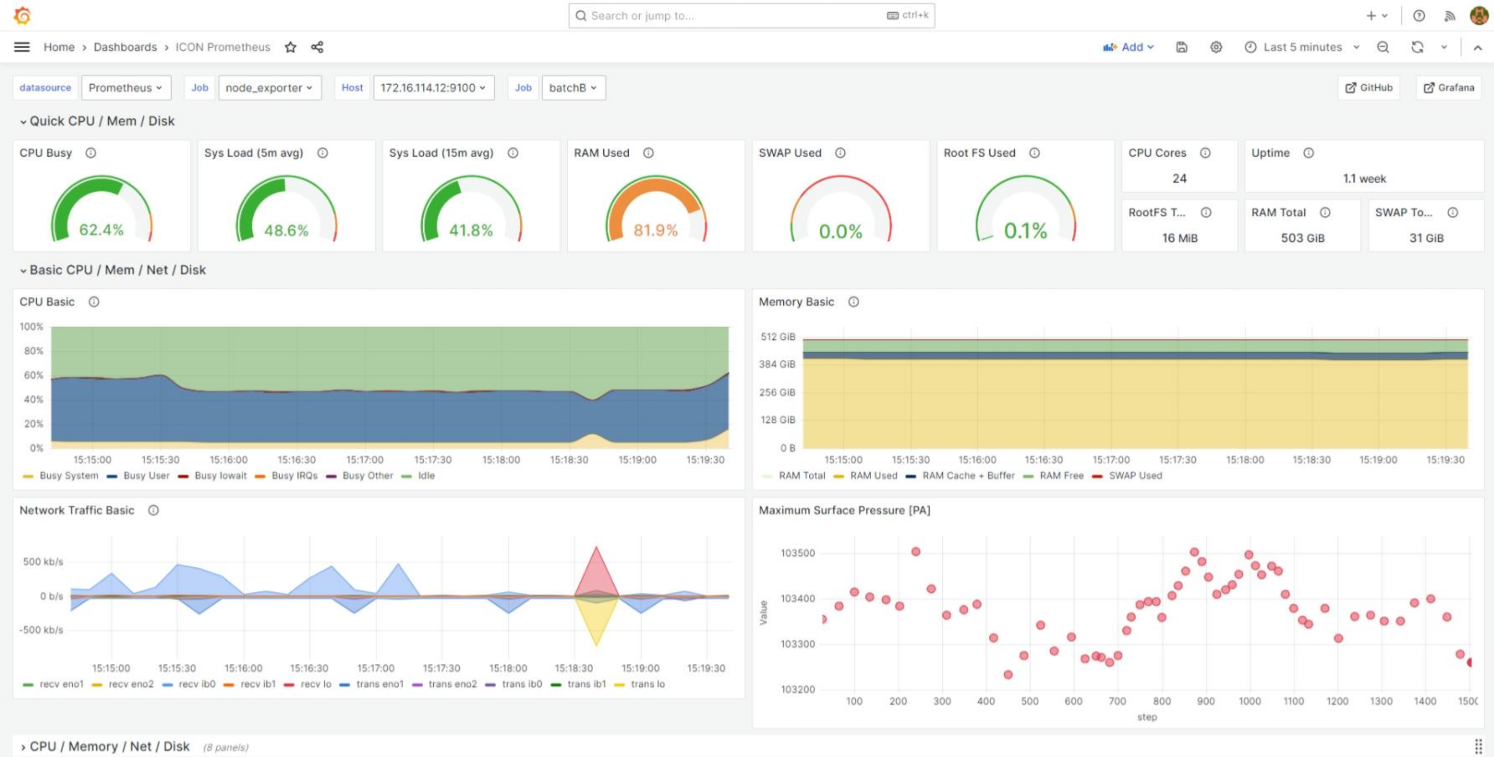
Mahnoosh Haghighatnasab
Deutscher Wetterdienst
E-mail:
mahnoosh.haghighatnasab@dwd.de

Contact: comin@icon-model.org

Applications

Telemetry

- Prometheus + Grafana: popular monitoring system and time series DB [1]
- "exporters" supply frontend with data (via HTTPS)
- e.g. Prometheus node_exporter: exporter for machine metrics
- Proof-of-concept: connect Grafana to ICON with a ComIn exporter plugin
- Flexible monitoring dashboard offered by Grafana



https://grafana.com , https://prometheus.io/

Source: C. Eser (DWD-TI), FP

Outline

Overview

- Introduction about ComIn
- Project history and status
- Documentation
- Tests

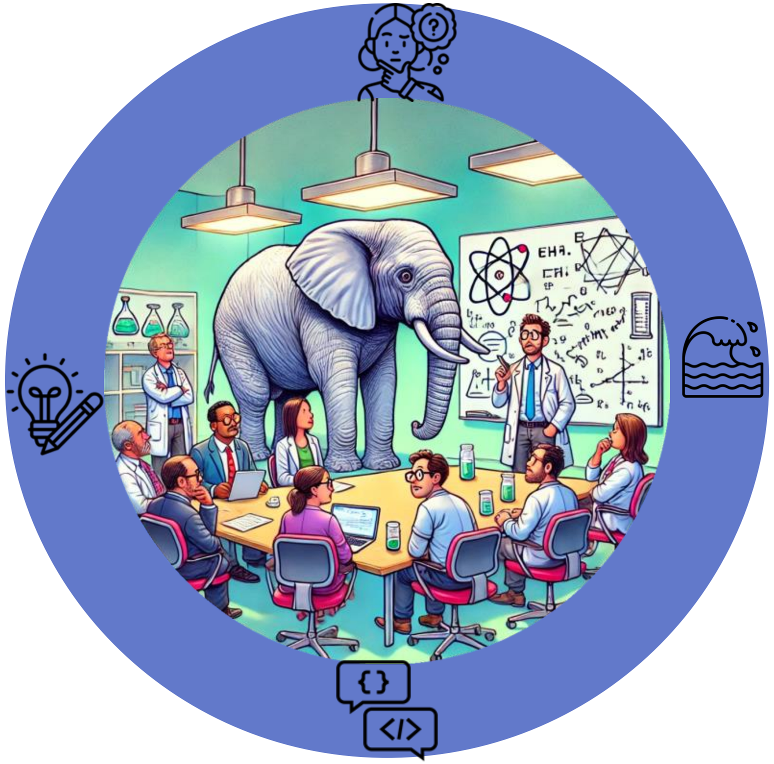
Applications

- Point source
- Catalyst in situ visualization
- Solar eclipse
- Further application

Discussion

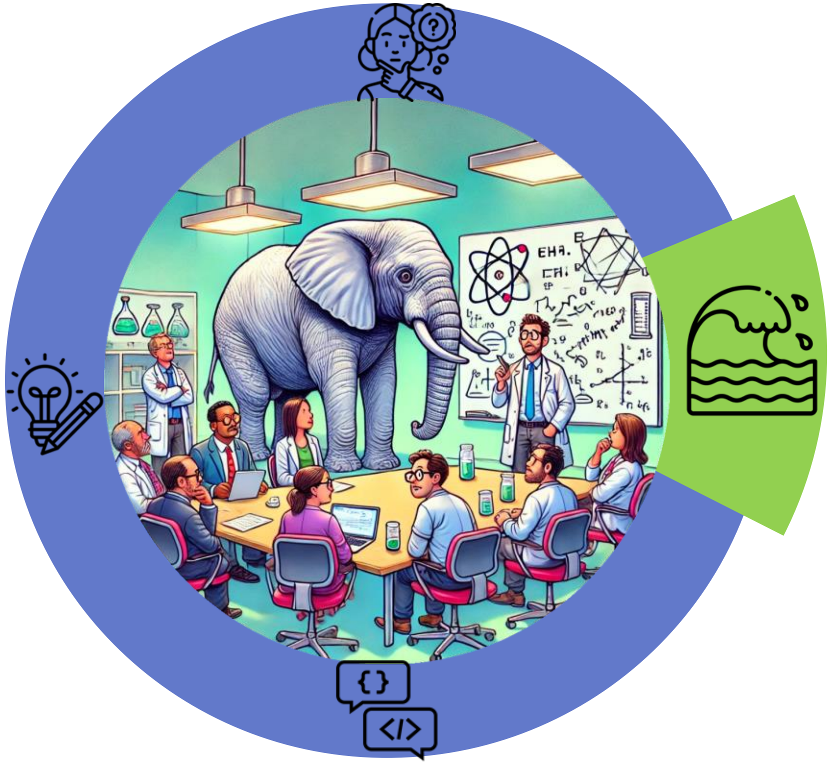
- Important points to discuss

Discussion



Expanding ComIn in other ICON Components

- **Current support limitations:**
  - Only atmosphere component
- **ICON Ocean as an example:**
  - Potential for broader application of ComIn
- **Design and demand uncertainty:**
  - Developers face uncertainty
  - Unclear demand from community
  - Will ComIn remain a tool only for the atmosphere component?
- **Challenges in moving beyond the atmosphere component:**
  - Human resources
  - Technical :
    - Descriptive data structure would lose its simplicity
    - Granularity

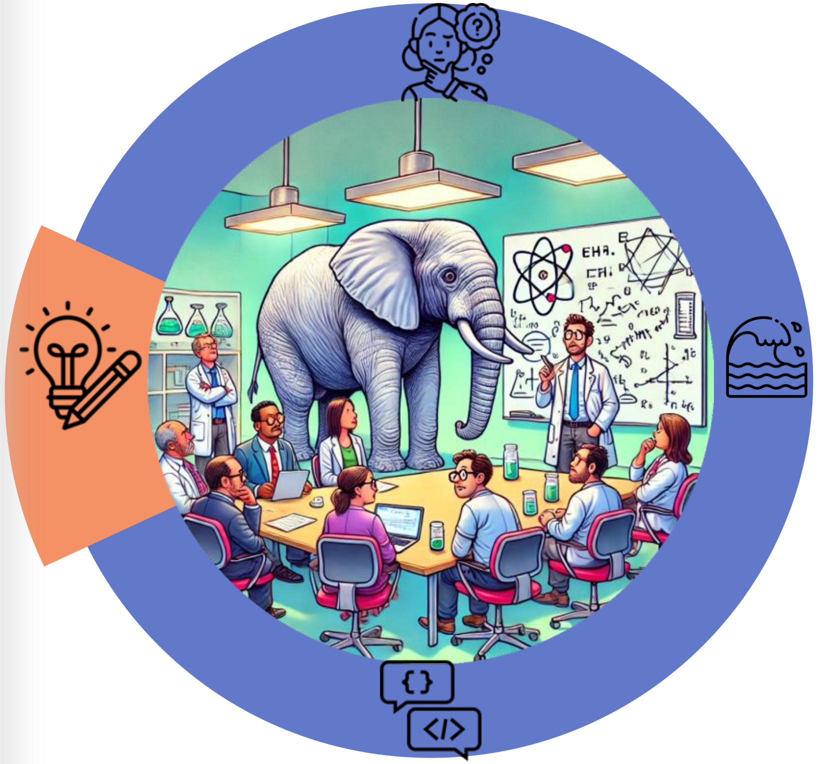


ComIn's role in the upcoming ICON rewrite

- **Ongoing rewrites:**
  - ICON was undergoing a rewrite in Python as part of the Exclaim project.
  - But a more recent rewrite in C++ is also planned.
- **Impact on ComIn:**
  - It is unclear how ComIn fits into this transition?
  - Whether it will be supported in the new C++ version?

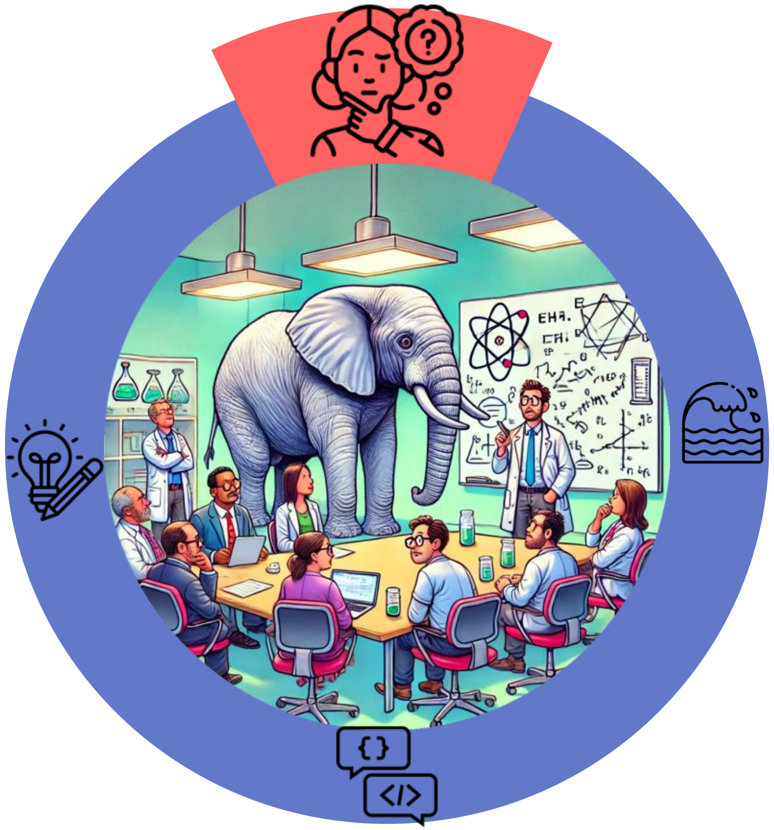
Do design decisions in ComIn restrict its broader application?

- **The limitations introduced by design decisions include:**
  - Granularity above the block loop level
  - Only global ICON variables are exposed to ComIn
  - Inability to switch ICON processes on or off via a ComIn plugin

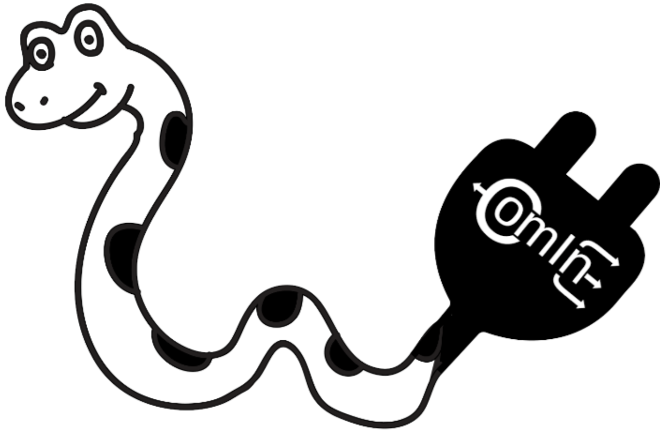


Is ComIn the right choice for your project?

- **YAC vs. ComIn:**
  - YAC already has a Python interface and can couple with ICON
  - Why ComIn is necessary?
- **Choosing the right tool:**
  - Based on their requirements, projects should consider whether they need to use **YAC**, **ComIn** or **bothe of them**?
- **When YAC might be sufficient:**
  - YAC is not a replacement for ComIn but some projects can be sufficiently supported by YAC



Embedded Python, wrapped pointers



- ComIn plugins makes use of an embedded Python interpreter.
- Executes a Python script in the primary constructor
- Field pointers are directly exposed to plugins, wrapped as NumPy arrays
  GPU accelerators: device pointers wrapped by CuPy library

```
TYPE :: t_comin_var_ptr
    TYPE(t_comin_var_descriptor) :: descriptor

    REAL(wp), POINTER :: ptr(:,:,:,:,:) => NULL()
    TYPE(c_ptr)       :: device_ptr = c_null_ptr
END TYPE t_comin_var_ptr
```