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 starts with an **overview of publications** associated to ComIn.
- 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** on the 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.

Table of Contents

ComIn publications

If you have published a paper using or describing ComIn, please expand the list yourself in a gitlab issue or contact comin@icon-model.org for the publication to be added.

Publications directly on ComIn

Please cite this work if you are publishing results using ComIn.

2025

Hartung, K., Kern, B., Dreier, N.-A., Geisbüsch, J., Haghighatnasab, M., Jöckel, P., Kerkweg, A., Loch, W. J., Prill, F., and Rieger, D.: ICON ComIn – the ICON Community Interface (ComIn version 0.1.0, with ICON version 2024.01-01), Geosci. Model Dev., 18, 1001–1015, https://doi.org/10.5194/gmd-18-1001-2025, 2025.

Publications with plugins using ComIn

This section does not contain any publications yet. Be the first to contribute here!

Overview of ComIn 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.

Name	Programming Language	Type	Author	Year	Description	URL	Status	Co ver
simple_fortran	Fortran	Tutorial	ICON-Comin	2024	None	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
calc_water_column	Fortran	Tutorial	ICON-Comin	2024	Simple diagnostic application, calculating the liquid water path, the ice water path, and the total water column.	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
yaxt_fortran	Fortran	Tutorial	ICON-Comin	2024	Example plugin to demonstrate the usage of the YAXT communication library.	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
simple_c	C/C++	Tutorial	ICON-Comin	2024	None	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
yac_input	C/C++	Tutorial	ICON-Comin	2024	Plugin written in C which encapsulates a YAC coupling inside a ComIn plugin.	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
yaxt_c	C/C++	Tutorial	ICON-Comin	2024	Plugin written in C to demonstrate using the YAXT communication library.	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
simple_python_plugin	Python	Tutorial	ICON-Comin	2024	None	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
point_source	Python	Tutorial	ICON-Comin	2024	Test plugin requesting a tracer that participates in ICON's turbulence and convection scheme. This plugin also shows how a grid point search in Python using SciPy can be done.	https://gitlab.dkrz.de/icon-comin/comin/-/tree/master/plugins	active	0.2
Hands-On Exercises	Python, Jupyter Notebooks	Tutorial	ICON-Comin	2024	Example plugins including the use of matplotlib, mpi4py	https://gitlab.dkrz.de/icon-comin/comin-training-exercises	active	0.1
MESSy Adapter	Fortran	Framework Adapter	DLR-IPA	2024	Adapter for the Modular Earth Submodel System (MESSy)		active	0.2
Prometheus Exporter	C/C++	Visualization	DWD	2023	Online monitoring within Grafana/Prometheus		discontinued	pre 0.1
Catalyst Adapter	C/C++	Visualization	DKRZ	2024	In situ visualization with Paraview/Catalyst	https://gitlab.dkrz.de/icon-comin/catalyst_adapter	active	
Solar Eclipse	Python	prognostic (prototype)	DWD	tbd.	considering solar eclipse in the insolation		to be developed	
calc_pot_etp	Python	diagnostic	CLM / UDAG project	tbd.	Calculation of potential evapotranspiration		to be developed	

cell tracking	Fortran	diagnostic	CLM / UDAG project	tbd.	cell tracking based on 5min precipitation output		to be developed	
proimpact 5 min prec	Python	diagnostic	ProImpact / Felipe Navarrete	2025	5min precipitation diagnostic	https://github.com/felipehnn/proimpact_comin	active	0.3
WindWaker	C++	Physics	DWD	2025	Model wakes of offshore wind farms	https://gitlab.dkrz.de/b381588/windwaker	active	0.3

Finally, the ComIn repository also contains a dummy cmake project in `test/downstream_project` which might serve as a starting point for the cmake build system.

Using bundled ComIn plugins

Repository checkout

In order to use the plugins with ICON, the first step involves building the ICON and plugins. The instruction of building and using the plugins is explained here for the *Levante_gcc* and the *DWD_nec* platforms:

Clone the ICON repository, which is publicly available under the BSD-3-C license:

```
module load git
git clone git@gitlab.dkrz.de:icon/icon-model.git

cd icon
git submodule update --init
```

Example plugins

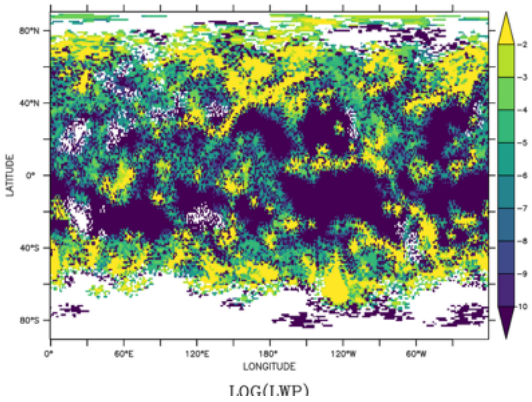
After cloning ICON, the example plugins can be found in `externals/comin/plugins`. In this folder there are different examples written in Fortran, C and Python:

▶ **simple_fortran_plugin.F90**
Simple ComIn plugin written in the **Fortran** programming language

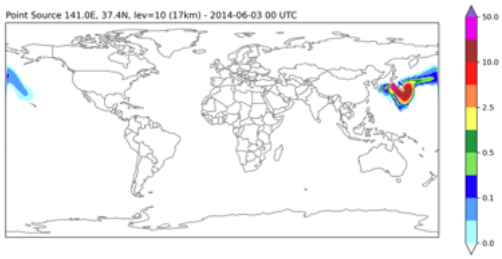
▶ **simple_c_plugin.c**
Simple ComIn plugin written in the **C** programming language

▶ **simple_python_plugin.py**
Simple ComIn plugin written in **Python**, using the ComIn Python adapter

▶ **calc_water_column_plugin.F90**
Add variables to ICON including metadata
Access prognostic fields of ICON, e.g. humidity tracers, including conditional usage of graupel
Access metrics data (height of half levels)



▶ **point_source.py** (Python)
Request a tracer that participates in ICON's turbulence and convection schemes.
add point source emissions to this tracer, update the tracer with tendencies received from ICON's turbulence and convection schemes.
This plugin also shows how a grid point search in Python using SciPy can be done.



Adding the namelist comin_nml to the ICON's namelist file

To enable the plugins, one or more entries are added to ICON's namelist `comin_nml` (which is part of the namelist file "atmo_namelist"). Specify the plugin information in the order you want to use them. The different items of `plugin_list` are

- `name`: the name of the plugin.
- `plugin_library`: the shared library file associated with the plugin.
- `primary_constructor` (optional): name of the primary constructor. It must be specified if it is **not** `comin_main`.
- `options` (optional): offers the possibility to pass a character string (e.g. a python script filename) to the plugin.
- `comm` (optional): denotes the name of the MPI communicator that is created for this particular plugin. This is useful when exchanging data with other running processes. The parameter `comm` can be left as an empty string if the application does not require a communicator for this plugin.

  Note: The first two components are mandatory to set.

```
&comin_nml
plugin_list(1)%name           = "simple_fortran_plugin"
plugin_list(1)%plugin_library = "libsimple_fortran_plugin.so"
```

Build instructions for the Levante platform

ComIn-enabled ICON installation on Levante_gcc

- **Load the necessary Python packages** To make use of the Python adapter a proper Python installation must be available, in particular if the `mpi4py` package should be used by the plugins. We recommend to use the Python installation contained in Levante's `spack` install tree. However, the package `py-mpi4py` is not contained there, therefore we created a custom `spack` install tree which can be used by placing the following file in your home directory:

```
mkdir ~/.spack
cat <<EOF >> ~/.spack/upstreams.yaml
upstreams:
  community_spack:
    install_tree: /work/k20200/k202160/community-spack/install
  system_installs:
    install_tree: /sw/spack-levante
EOF
```

  Load the package with

```
spack load py-mpi4py
```

  For running the **point_source.py** example the `scipy` package is required which can also be found in the community-spack and can be loaded with

```
spack load py-scipy
```

- **Build ICON with ComIn.** Out-of-source build is recommended.

```
cd icon
mkdir build && cd build
../config/dkrz/levante.gcc --enable-comin --enable-bundled-python=comin
make -j6
```

The `--enable-bundled-python=comin` configuration option is used to build the ComIn Python adapter.

- **Build the plugins.** Through the building process, a shared library of the plugin is generated, allowing for dynamic loading while running ICON:

```
(cd externals/comin/build && cmake -DCOMIN_ENABLE_EXAMPLES=ON .)
(cd externals/comin/build && make)
```

Modifying the ICON run script

It is recommended to adjust your run script template before configuring ICON. This modified version of your template will be copied to the `build/run` subdirectory then.

Modifying your experiment's template involves two parts. First, the `&comin_nml` namelist needs to be added to the namelist "atmo_namelist" (see above).

In addition, you need to set the path of your plugin's shared library in the `LD_LIBRARY_PATH`. To accomplish this, there are two options available.

- You can add it directly in to your run script:

```
export LD_LIBRARY_PATH="${path_to_plugin}:$LD_LIBRARY_PATH"
```

- Alternatively you can use the auxiliary function `add_comin_setup` in the run script which does the same automatically. To use this function your `basedir` variable must be set to the `build` directory of your ICON installation.

```
add_comin_setup "/externals/comin/build/plugins/simple_fortran"
```

Run the experiment on Levante

The following commands copy the modified version of your template to the `build/run` subdirectory and launch the batch job

```
cd build
./make_runscripts --all
cd run
sbatch name_of_your_runscript.run
```

An alternative option is to run your experiment on interactive nodes. Allocate resources for a suitable (cheap!) cluster queue on `Levante` and wait for the interactive job to start:

```
salloc -p interactive -A <account> --time=04:00:00 --tasks-per-node=128 --nodes=1 --exclusive
```

Then run the test interactively (remember to make your `$BASEDIR` known to the new shell: `export BASEDIR= ...`):

```
cd $BASEDIR/icon/build/run
./name_of_your_runscript.run
```

DWD NEC platform

Limitations

Note that the support for ComIn plugins written in the Python programming language is limited to the x86 NEC Vector Hosts. Native support for task on the NEC Vector Engines is currently under investigation.

Build instructions

- Create a build directory.

```
mkdir build && cd build
mkdir VH VE
```

- Build ICON on vector host.

```
cd VH
../../config/dwd/rcl.VH.gcc --enable-comin
make -j6
```

- Build plugins on vector host.

```
cd externals/comin/build
module purge
module load apps sx/default gcc/11.2.0 mpi/3.5.0 libfyaml/0.8-VH-gnu unsupported cmake/3.26.4
sed -i 's/-static//g' CMakeCache.txt
cmake -DCMAKE_C_COMPILER=mpincc -DCMAKE_C_FLAGS='-vh' -DCMAKE_Fortran_COMPILER=mpinfort -DCMAKE_Fortran_FLAGS='-vh' -
       DCMAKE_CXX_COMPILER=mpinc++ -DCOMIN_ENABLE_EXAMPLES=ON .
make
```

- Build ICON on vector engine.

```
cd ../VE
../../config/dwd/rcl.VE.nfort --enable-comin
make -j6
```

- Build plugins on vector engine.

```
cd externals/comin/build
module purge
module load sx/default nfort/5.1.0 nc++/5.1.0 mpi/3.5.0 libfyaml/0.8-sx unsupported cmake/3.26.4
cmake -DCMAKE_C_COMPILER=mpincc -DCMAKE_Fortran_COMPILER=mpinfort -DCMAKE_CXX_COMPILER=mpinc++ -DCOMIN_ENABLE_EXAMPLES=ON
       .
make
```

- Modify the run script. This step is almost the same as is explained for Levante_gcc except that one also must add the path of shared library of plugin(s) to `VE_LD_LIBRARY_PATH`.

```
export LD_LIBRARY_PATH="${path_to_plugin_on_VH}:$LD_LIBRARY_PATH"
export VE_LD_LIBRARY_PATH="${path_to_plugin_on_VE}:$VE_LD_LIBRARY_PATH"
```

or use the auxiliary function `add_comin_setup` . This function does the same for both vector host and vector engine automatically.

```
path_to_plugin=`/externals/comin/build/plugins/simple_fortran`
add_comin_setup "$path_to_plugin"
```

- Run the experiment.

```
cd /build/VE
./make_runscripts --all
cd run
qsub name_of_your_runscript.run
```

Writing ComIn plugins: Building blocks

The initial stage involves choosing the preferred programming language, which can be either Fortran 2003, C/C++ or Python. As an illustration, provided here is a guide on creating a plugin using Fortran.

Each plugin must have three parts:

- Primary constructor
- Secondary constructor
- Callback function(s)

Primary constructor

The plugin allows users to write subroutines that can be called at predefined events (entry points) throughout the model simulation. The primary constructor registers the plugin, and it especially registers additional variables and callback functions. Basically, the primary constructor contains the following steps:

Compatibility checks

- It is explicitly checked that the major **versions of the ComIn library** that is used by the ICON and the one that is used by the plugin match.

```
version = comin_setup_get_version()
```

  As many components of the development are still in the testing phase, the initial public release is set to version number 0.1.0. There is a **finish subroutine** which can be called in different occasions. For example here it can be used to check if the component `version_no_major` in the data structure `version` is `0`.

```
IF (version%version_no_major > 1) THEN
   CALL comin_plugin_finish("comin_main (simple_fortran_plugin)", "incompatible version!")
END IF
```

- Another library function that is used in some contexts returns the ComIn-internal ID that is used to identify a specific plugin during the subsequent operations:

```
CALL comin_current_get_plugin_info(this_plugin)
```

Registering additional variables

Plugins are allowed to register additional model variables for ICON. A list of to-be-created variables made known to the ICON via the function `comin_var_request_add`.

```
CALL comin_var_request_add(var_descriptor, lmodexclusive)
```

- The `var_descriptor` is required to describe (and uniquely identify) a model variable in ICON.

```
var_descriptor=t_comin_var_descriptor( id = domain_id, name = "variable_name")
```

- Flag `lmodexclusive`: Whenever a plugin calls `comin_var_request_add`, there is a check to determine if the requested variable is already registered. In this

case, the existing variable will be used instead of creating a new one. However, if the variable exists and is either exclusively requested in the current call or was exclusively requested before, the model aborts based on the `lmodexclusive` setting.

- Variables may also be appended to ICON's container of tracer variables through the `tracer` flag (part of the metadata). Apart from that aspect it is not possible to create additional variable containers via the adapter library. Note that it cannot be assumed (if only because of the "sharing" of variables between multiple ComIn plugins) that the tracers generated by a module are stored consecutively.

```
CALL comin_metadata_set(var_descriptor, "tracer", .true.)
```

While it is possible to create variables only for certain domains, ICON has the restriction that tracer variables have to be present on *every* domain. For this reason, it is necessary to choose domain id −1 (meaning all domains) as part of the `var_descriptor` for variables with `tracer = .true.`

- Newly created fields can be added to ICON's set of restart variables.

```
CALL comin_metadata_set(var_descriptor, "restart", .true.)
```

Registering callbacks

The primary constructor appends subroutines of the 3rd party module to the callback register via the adapter library subroutine `comin_callback_register`.

```
CALL comin_callback_register(entry_point_id, fct_ptr)
```

- `entry_point_id`: entry points denote events during the ICON model simulation, which can trigger a subroutine call of the plugin. Entry points are denoted by named integer constants. The table of available entry points is available in the **technical documentation**.
- `fct_ptr`: this is the callback function. Callback functions do not have additional arguments or return values. The callback function has to be interoperable with the C processor (for Fortran, this requires the `BIND(C)` attribute; see the **technical documentation**).

Getting descriptive data structures

The descriptive data structures contain information on the ICON setup (e.g. Fortran `KIND` values), the computational grid(s), and the simulation status. All descriptive data structures are treated as read-only (seen from the perspective of the 3rd party plugins). However, this read-only nature is (currently) not enforced. For efficiency reasons, the adapter library directly uses pointers to ICON data structures where possible. This holds mostly for components of `p_patch`, while non `p_patch` descriptive data are copied from the host model.

- **Global data** is available for the plugins primary constructor and all subsequent subroutine callbacks. Global data is never changed or updated and invariant w.r.t. the computational grid (logical domain ID). The detailed table of the components of global data can be found in the **technical documentation**.

```
TYPE(t_comin_descrdata_global), POINTER :: p_global
p_global => comin_descrdata_get_global()
```

- **Grid information** is available for the 3rd party module's primary constructor and all subsequent subroutine callbacks. Grid information is never changed or updated. The data structures in this section are replicated for each computational domain (logical domain ID).

```
TYPE(t_comin_descrdata_domain), POINTER :: p_patch
 p_patch => comin_descrdata_get_domain(jg)
```

- **Timing information** on the simulation.

```
TYPE(t_comin_descrdata_simulation_interval), POINTER :: p_simulation_interval
p_simulation_interval => comin_descrdata_get_simulation_interval()
```

- Time step length per domain.

```
dtime=comin_descrdara_get_timesteplength()
```

Secondary constructor

A secondary constructor is called *after* the allocation of ICON variable lists and fields and *before* the time loop. It needs to be registered by the primary constructor as one of the plugin's callbacks.

- Access to ICON data fields happens via an accessor function `comin_var_get`. Basically, `comin_var_get(context, var_descriptor, flag, var_handle)` returns a variable handle `var_handle` of type `t_comin_var_handle`, with which the pointer to the data can be accessed in a second step.
  - `context`: the name of the entry point.
  - `var_descriptor`: same as described in primary constructor part.
  - `flag`: the optional argument `flag` provides information w.r.t. the data flow. Flags may be combined like `flag = IOR(COMIN_FLAG_READ, COMIN_FLAG_WRITE)`. It is important to highlight that when the `comin_var_request_add` procedure is executed, a variable is not immediately created. This step only involves the registration of a new variable. To use this variable later, it must be queried, similar to the other variables, using the `comin_var_get` function with `flag=COMIN_FLAG_WRITE`.
  - `comin_var_get` registers the access to a variable and returns a variable handle.

Code example:

```
TYPE(t_comin_var_handle) :: var1
CALL comin_var_get(context, var_descriptor, flag, var1_handle)
```

The 5D pointer is than accessed via thr type-bound procedure `get_ptr`.

Code example:

```
REAL(wp), DIMENSION(:,:,:,:,:) :: var1_ptr => null()
CALL var1_handle%get_ptr(var1_ptr)
```

Note, it is important here to know in advance, of which data type the delivered pointer will be. The integer metadata `datatype` provides the data type. COMIN_VAR_DATATYPE_DOUBLE, COMIN_VAR_DATATYPE_FLOAT, and COMIN_VAR_DATATYPE_INT are the currently available data types.

There exists a convenience procedure `to_3d` for accessing 2D/3D fields: In practice, access to fields can be simplified, under the condition that the sequence of dimensions is `(jc,jk,jb)`. This exact dimension sequence is (currently) fulfilled by the ICON model. In this case, a 3D pointer variable `REAL(wp) :: slice(:,:,:)` can be generated directly from the handle using the procedure.

```
CALL var1_handle%to_3d(slice)
```

ComIn plugins written in the Python programming language

Python plugins can be attached to ComIn via the Python adapter which is located in the `plugins` directory in the ComIn source code. It is compiled with ComIn if `COMIN_ENABLE_PYTHON_ADAPTER` is enabled in the CMake configuration, see the build instructions above. The Python adapter embeds a Python interpreter, which also has the `comin` Python module available. This module contains all the functions, variables, constants and data structures of the **Python language API**. When including the Python adapter in the namelist, the Python plugin script must be specified as the `options`, which can be modified while the actual ComIn plugin Python adapter (`libpython_adapter.so`) remains unchanged. This script is executed in the primary constructor of the Python adapter. Further callbacks can then be registered by decorating functions with entrypoints.

```
plugin_list(2)%name           = "simple_python_plugin"
plugin_list(2)%plugin_library = "libpython_adapter.so"
plugin_list(2)%options        = "${basedir}/externals/comin/plugins/python_adapter/examples/simple_python_plugin.py"
```

Record & Replay functionality

In principle the technical development process of a plugin can be carried out without the presence of the ICON host model, see the section "Build Instructions for ComIn Plugins" below. In general, however, the objectives of a plugin are too complex and involve too many variables to allow development independent of an ICON run. On the other hand, starting the complete model run is resource and time intensive, which in turn limits the plugin development. For this case, the ICON Community Interface offers the `replay_tool` tool, which is described in the following.

Replay tool

Located in the `replay_tool` subdirectory, a small executable `comin_replay` is distributed together with the ComIn source code. This tool performs the task of making previously recorded data sets (ICON variables, descriptive data) available for ComIn plugins. It therefore acts as a fake host model, which simply reproduces data records that were previously captured with the ICON model.

Let us assume, for the time being, that such data records already exist. They are stored in the NetCDF file format, and it is implicitly assumed that the replay process is executed with as many processes as the ICON simulation itself (each NetCDF file stores the partition of a single MPI task).

To illustrate the replay process, we attach the "simple_python_plugin" explained above to `comin_replay`. This will enable us to develop additional functionality in the "simple_python_plugin", using real data input recorded from the ICON model.

The replay tool expects a fortran namelist as command-line argument, that contains a definition of the `comin_nml` and a `replay_tool_nml`. It looks quite similar to the usual `comin_nml` for ICON, with an additional plugin `libcomin_var_replay_plugin.so` that loads the variables back into memory. Note that there is no need to specify the list of available variables or the entry point where the variable has been recorded - this is automatically retrieved from the recorder NetCDF files.

```
&replay_tool_nml
  replay_data_path = "path/to/the/replay_data/"
  msg_level        = 42
/
&comin_nml
  plugin_list(1)%name           = "var_replay_plugin"
  plugin_list(1)%plugin_library = "$COMIN_DIR/build/replay_tool/libcomin_var_replay_plugin.so"
  plugin_list(2)%name           = "simple_python_plugin"
  plugin_list(2)%plugin_library = "libpython_adapter.so"
  plugin_list(2)%options        = "$PLUGINDIR/simple_python_plugin.py"
/
```

Execution happens in a run script with the same parallel queue settings as the ICON run. You might, for example, create a copy of the full ICON run script, simply replacing the `MODEL=.../icon` setting by the `comin_replay` executable. Note, however, that usually the ICON model run comprises additional MPI tasks, e.g., for asynchronous output writing. Therefore, the number of MPI tasks has to be *decreased* accordingly for the replay run by adjusting the `--ntasks` argument of the `srun` command.

Note: It is currently not supported to add the `var_replay_plugin` plugin multiple times to the `comin_replay` run.

Recorder plugins

Two separate plugins are provided which *capture* the ICON host model data during a full model run. Both are located in the `replay_tool` subdirectory and compiled during ComIn's build process:

- `build/replay_tool/libcomin_run_recorder_plugin.so`: This plugin dumps all descriptive data to disk. It is attached to the ICON model as a secondary constructor callback, which collects most of the descriptive data. During the remaining callbacks, additional time-dependent descriptive data is recorded.
- `build/replay_tool/libcomin_var_recorder_plugin.so`: This plugin captures the data arrays of a given set of variables and for a given entry point. Before you attach this plugin to the ICON model run, the hard-coded entry point constant `ep` has to be set in the source code file `replay_tool/comin_var_recorder_plugin.F90`. The list of variables which shall be recorded is provided as a comma-separated string via the namelist parameter `comin_nml :: plugin_list(1)options`.

Example: The default entry point in the ´`comin_var_recorder_plugin.F90` is `ep = EP_ATM_TIMELOOP_END`. We change this to `EP_ATM_WRITE_OUTPUT_BEFORE` and rebuiild the recorder plugins.

Afterwards, we can activate the recorder plugins with the following ICON namelist setting, capturing the pressure field `pres`:

```
&comin_nml
  plugin_list(1)%name           = "run_recorder_plugin"
  plugin_list(1)%plugin_library = "$COMIN_DIR/build/replay_tool/libcomin_run_recorder_plugin.so"
  plugin_list(2)%name           = "var_recorder_plugin"
  plugin_list(2)%plugin_library = "$COMIN_DIR/build/replay_tool/libcomin_var_recorder_plugin.so"
  plugin_list(2)%options        = "pres"
/
```

During the ICON run, various NetCDF files are created in the experiments folder; the usual path would be `build/experiments/...`.

- The descriptive data files have the name<prefix>XXX.nc , where XXX denotes the MPI rank and <prefix> is an optional name prefix.
- The variable contents are stored in files vars_XXX.nc, where XXX denotes the MPI rank.

All files contain certain meta-data attributes, e.g. the ComIn version. As described above, they can now be used by the comin_replay tool to facilitate stand-alone plugin runs for development.

Note: Currently, collecting ICON data for multiple entry points requires several independent model runs.

ComIn plugins on accelerator devices (GPUs)

ICON supports massively parallel accelerator devices such as GPUs (Graphics Processing Units). For a detailed description of this parallelization model, see the ICON tutorial (DOI: 10.5676/DWD_pub/nwv/icon_tutorial2025), Section 8.5 "ICON on Accelerator Devices". In the context of the Community Interface, the most important aspect is the handling of the separate GPU and CPU memory. ComIn provides a set of API functions that organize the transfer of variables between a plugin running on either CPU or GPU and a GPU-based ICON simulation. This section describes the use of this API by means of a practical example, executed on the Levante supercomputer of the DKRZ.

Preparation: GPU-enabled ICON binary

As a first step, we set up a Python virtual environment called venv. We need this environment to install the CuPy library for GPU-accelerated computing. CuPy shares the same API set as NumPy and SciPy, allowing it to be a drop-in replacement for running NumPy/SciPy code on the GPU.

```
/sw/spack-levante/miniforge3-24.11.3-2-Linux-x86_64-rf4err/bin/python -m venv venv
source venv/bin/activate
pip install numpy setuptools cupy-cuda12x
```

The next step is to build a GPU-enabled ICON binary on Levante. We clone the ICON repository,

```
module load git
git clone git@gitlab.dkrz.de:icon/icon-model.git
cd icon-model/
git submodule update --init --recursive
```

and create a build directory. Here, we already copy a suitable ComIn+GPU test script to the run/ folder. This test exp.atm_tracer_Hadley_comin_portability is provided together with the ICON source code and runs a ComIn plugin written in the Python programming language, externals/comin/plugins/python_adapter/test/gpu_test.py.

```
mkdir build && cd build/
cp ../run/checksuite.infrastructure/comin/exp.atm_tracer_Hadley_comin_portability ../run/
```

As usual with ICON, users are recommended to run an appropriate platform- or machine-specific configuration wrapper that sets the required compiler and linker flags. This is the config/dkrz/levante.gpu script in this case:

```
../config/dkrz/levante.gpu.nvhpc-24.7 --enable-comin --disable-jsbach --disable-quincy --disable-rte-rrtmgp --enable-bundled-
      python=comin  --disable-silent-rules
make -j16
```

Now all the necessary preparations are done and we have a GPU-enabled ICON binary along with a ComIn Python adapter. Note that these preparations only need to be done once.

Generate the run script with the following command:

```
./make_runscripts --all
```

Then adjust the account in run/exp.atm_tracer_Hadley_comin_portability.run. Afterwards, to test the setup:

```
sbatch run/exp.atm_tracer_Hadley_comin_portability.run
```

GPU-enabled Python plugin: `gpu_test.py`

In the following, we will focus on the plugin script itself, i.e. the Python script `build/externals/comin/plugins/python_adapter/test/gpu_test.py`. In particular, we will explain ComIn's host device copy mechanism, summarized in the following table:

	CPU plugin (Python: NumPy)	GPU plugin (Python: CuPy)
GPU ICON	auto. host-device copy	no memcopy required
CPU ICON	no memcopy required	- model abort -

ComIn initially assumes that the plugin is to be executed on the CPU host, regardless of whether the ICON model runs on GPUs or not. This basic setting enables the execution of unmodified "legacy" plug-ins. If the test script is unmodified, fields are copied to the host prior to the ComIn callback.

Besides this, the plugin developer can check the availability of the GPU using the `has_device` descriptive data info. The following statement prints if an accelerator device is available:

```
glb = comin.descrdata_get_global()
print(f"{glb.has_device=}", file=sys.stderr)
```

See the **Python API documentation** for other GPU-related contained in the ComIn descriptive data.

If the plugin is also to run on GPUs (which is often possible without major adjustments thanks to the NumPy replacement module CuPy), then *read* access to a variable on the GPU can be specified using a flag for **`comin.var_get`**: **`comin.COMIN_FLAG_READ | comin.COMIN_FLAG_DEVICE`**.

```
pres = comin.var_get([comin.EP_ATM_WRITE_OUTPUT_BEFORE], ("pres", 1),
                     comin.COMIN_FLAG_READ | comin.COMIN_FLAG_DEVICE)
```

ComIn also catches the error that a plugin should run on the GPU, although ICON itself was not started on GPUs.

As explained above, we can use the CuPy library as a drop-in replacement for running NumPy code on the GPU. To do this, our Python example replaces

```
import numpy as xp
```

with

```
import cupy as xp
```

Our test script automatically recognizes the availability of Cupy, enabling it to run in both execution modes: as a GPU or CPU plugin.

We will now discuss *write* access to ICON variables. The non-trivial case would be when ICON runs on GPUs, yet a non-GPU plugin needs to write to a variable array. Similar to read access, the porting effort is minimal. A non-GPU plugin that uses ICON on GPUs only needs to set **`comin.COMIN_FLAG_WRITE`**; then, `comin_callback_context_call` automatically takes care of the data transfers. This corresponds to the first column of the following table ("Execution in GPU Section"):

`COMIN_FLAG_DEVICE`	`COMIN_FLAG_READ`	`COMIN_FLAG_WRITE`	execution in GPU section	ex. in CPU section (`lacc=.FALSE.`)
	x		update host memory before callback	-
		x	update device memory after callback	-
	x	x	update host memory before callback; update device memory after callback	-
x	x		-	warning
x		x	-	warning
x	x	x	-	warning

Detail: The right column, "execution in CPU section", on the other hand, refers to sections in the ICON code which have not yet been ported to GPUs. In the (rare) case that an entry point is located in such a section, the access to GPU-enabled variables triggers a warning in ComIn, but no special host-to-device update for read and device-to-host for write access has been implemented.

One final remark on *descriptive data structures*: Descriptive data structures, such as domain data, are not automatically transferred to the GPU device. Plugins only receive CPU pointers. This is because data structures are complex and plugins do not register for specific items. For GPUs, keeping all data consistent would be necessary, even if much of it is unnecessary.

On the other hand, it is relatively straightforward to synchronise the descriptive data arrays on the fly on the plugin side, and the runtime overhead is negligible since most plugins only require these structures during initialization. We demonstrate this in the following code snippet:

```
domain = comin.descrdata_get_domain(1)
comin.print_info(f"{np.asarray(domain.cells.clon[:10])=}")
```

Similarly, it is possible to create a CuPy array and move the data to the current GPU device:

```
domain = comin.descrdata_get_domain(1)
clon = cp.asarray(domain.cells.clon[:10])
```

Build instructions for ComIn plugins

ComIn plugins are shared libraries, attached to ICON using the dynamic loader of the operating system. For building a ComIn plugin we recommend to use CMake. In the first step you should create a separate CMake project and place your plugin there.

In the next step, one must build ComIn. We strongly recommend the out-of-source build (the instructions can be found in the next section). Following that, ComIn offers a CMake config (`ComInConfig.cmake`) such that it can be easily found in your CMake project. Then, establish a connection between your CMake project and ComIn.

```
cd your_project
```

In the next step, generate a `CMakeLists.txt` in your CMake project with the following lines:

```
project(name_of_your_project LANGUAGES Fortran)
find_package(ComIn)
add_library(your_plugin MODULE your_plugin.F90)
target_link_libraries(your_plugin ComIn::ComIn)
```

Note: In the example above is assumed you want to build a Fortran plugin. In case of C plugin, the `LANGUAGES` does not need to be specified. Afterwards, you can create a build directory and build your plugin:

```
mkdir build && cd build
export ComIn_DIR=path_to_the_comin_build_directory
cmake ..
make
```

Using ComIn's testing mechanism

ComIn offers the functionality to test your plugin with the `replay_tool` emulator using CTest. In particular this can be used in a CI/CD setup for validating that the plugin builds and can be executed. A detailed documentation of the `replay_tool` and how to generate the input data can be found in the **User Guide**

To add and configure tests in your projects ComIn provides **utility functions**.

Online replay data can be added by the CMake command `comin_add_replay_data`.

```
comin_add_replay_data(NAME your_replay_data
  URL https://example.com/data.tar.gz
  MD5HASH abcdefg)
```

This adds a test to your projects which takes care of the download of the data.

To add a test, you can use the `comin_add_replay_test` CMake function in `CMakeLists.txt`.

```
comin_add_replay_test(NAME your_test
  REPLAY_DATA your_replay_data)
```

This generates a CTest test with the name `your_test` and sets up everything to run the `replay_tool`. Alternatively you can pass a local path to the replay data as `REPLAY_DATA_PATH`. Further arguments can be found in the **CMake documentaton**.

To add a plugin to the test use the function `comin_test_add_plugin` in `CMakeLists.txt` .

```
comin_test_add_plugin(TEST your_test
  NAME "your_plugin"
  PLUGIN_LIBRARY $<TARGET_FILE:your_plugin>
  PRIMARY_CONSTRUCTOR "your_plugin_main"
  OPTIONS "some options"
  COMM "your_comm")
```

The parameters correspond to the parameters in the namelist (`t_comin_plugin_description`) for configuring a plugin.

The tests can be executed with `ctest` or `make test`. Note that the CMake variable `BUILD_TESTING` must be set to `ON` to build the tests.

ComIn-standalone setup on Levante (DKRZ)

You can build ComIn without ICON as the host model and test plugins using the standalone replay_tool that comes with ComIn.

If you plan to use the Python API, it is recommended to create a virtual environment for ComIn. For Python 3.12, use these commands:

```
spack load /mnm # tells mpi4py which MPI to use
/sw/spack-levante/miniforge3-24.11.3-2-Linux-x86_64-rf4err/bin/python -m venv venv
source venv/bin/activate
python -m pip install numpy mpi4py xarray # add any other packages you need
```

Load the modules needed to build ComIn using this command:

```
module load gcc/11.2.0-gcc-11.2.0 netcdf-c/4.8.1-gcc-11.2.0 netcdf-fortran/4.5.3-gcc-11.2.0
export MPI_ROOT='/sw/spack-levante/openmpi-4.1.2-mnmady'
```

Clone the ComIn git repository with:

```
module load git
git clone git@gitlab.dkrz.de:icon-comin/comin.git
```

Alternatively, download a public release tarball from https://gitlab.dkrz.de/icon-comin/comin.

Then follow the standard CMake workflow: create a build directory, configure and build.

```
mkdir comin/build
cd comin/build
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCOMIN_ENABLE_EXAMPLES=ON -DCOMIN_ENABLE_PYTHON_ADAPTER=ON -
       DCOMIN_ENABLE_REPLAY_TOOL=ON -DBUILD_TESTING=ON -DCMAKE_C_COMPILER="${MPI_ROOT}/bin/mpicc" -
       DCMAKE_CXX_COMPILER="${MPI_ROOT}/bin/mpic++" -DCMAKE_Fortran_COMPILER="${MPI_ROOT}/bin/mpif90" ..

make -j6
```

The CMake command enables:

- Python adapter by setting `-DCOMIN_ENABLE_PYTHON_ADAPTER=ON`,
- Example plugins by setting `-DCOMIN_ENABLE_EXAMPLES=ON`,
- CI/CD tests (`ctest` command) by setting `-DBUILD_TESTING=ON`.

For debugging, you can use the `VERBOSE=1` option in CMake.

Run tests in the build directory with:

```
ctest
```

If **parallel tests** fail, add these environment variables to the runscript in the build directory `comin/build/test/parallel/run.sh`:

```
export OMPI_MCA_osc="ucx"
export OMPI_MCA_pml="ucx"
export OMPI_MCA_btl="self"
export UCX_HANDLE_ERRORS="bt"
export OMPI_MCA_pml_ucx_opal_mem_hooks=1
```

(Further details can be found here)

For **YAXT** functionality, eg. for the replay_tool's halo synchronization feature, you must first install YAXT at a specific path. Then reconfigure by adding this CMake option: `-DCMAKE_PREFIX_PATH=path_to_install_yaxt` When using YAXT, set the `LD_LIBRARY_PATH` environment variable to include the YAXT installation path before running `cmake`.

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:<YAXT_INSTALL_PATH>
```

ComIn-standalone setup on NEC (DWD)

Stand-alone building ComIn on NEC requires separate builds for both the vector engine (VE) and the vector host (VH).

```
mkdir build_VH build_VE
```

First step: stand-alone build for VE.

```
 bash
cd build_VE
module purge
module load sx/default nfort/5.1.0 nc++/5.1.0 mpi/3.5.0 libfyaml/0.8-sx unsupported cmake/3.26.4
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_C_COMPILER=mpincc -DCMAKE_Fortran_COMPILER=mpinfort -DCMAKE_CXX_COMPILER=nc++ -
       DCMAKE_CXX_FLAGS="-stdlib=libc++" -DCOMIN_ENABLE_EXAMPLES=ON ..
make
```

Second step: stand-alone build for VH.

```
cd build_VH
module purge
module load apps sx/default gcc/11.2.0 mpi/3.5.0 libfyaml/0.8-VH-gnu unsupported cmake/3.26.4
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DCMAKE_C_COMPILER=mpincc -DCMAKE_C_FLAGS='-vh' -DCMAKE_Fortran_COMPILER=mpinfort -
       DCMAKE_Fortran_FLAGS='-vh' -DCMAKE_CXX_COMPILER=mpinc++ -DCMAKE_CXX_FLAGS='-vh' -DCOMIN_ENABLE_EXAMPLES=ON ..
make
```