ICON ComIn - The ICON Community Interface (ComIn version 0.1.0, with ICON version 2024.01-01)

Kerstin Hartung[1,*], Bastian Kern[1,*], Nils-Arne Dreier[3], Jörn Geisbüsch[4], Mahnoosh Haghighatnasab[4], Patrick Jöckel[1], Astrid Kerkweg[2], Wilton Jaciel Loch[3], Florian Prill[4], and Daniel Rieger[4]

[1]Deutsches Zentrum für Luft- und Raumfahrt, Institut für Physik der Atmosphäre, Oberpfaffenhofen, Germany
[2]Forschungszentrum Jülich GmbH, Institute of Energy and Climate Research 8, Troposphere, Jülich, Germany
[3]Deutsches Klimarechenzentrum, Hamburg, Germany
[4]Deutscher Wetterdienst, Offenbach, Germany
[*]These authors contributed equally to this work.

Correspondence: Kerstin Hartung (kerstin.hartung@dlr.de)

Abstract. In 2021, a team of developers from the Deutscher Wetterdienst (DWD), the German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt, DLR), the German Climate Computing Center (Deutsches Klimarechenzentrum, DKRZ) and the Forschungszentrum Jülich (FZJ) started the Icosahedral Nonhydrostatic (ICON) Model Community Interface (ComIn) project: ICON ComIn is a library with multi-language support for connecting third-party modules ('plugins') to the ICON model using the dynamic loader of the operating system. ComIn is intended for a wide range of use cases, from the integration of simple diagnostic Python scripts to full chemistry models into ICON. ICON ComIn is distributed with the ICON model code under an open source license. Its application programming interface (API) provides a low barrier for code extensions to ICON and reduces the migration effort in response to new ICON releases. ComIn's main design principles are that it is lightweight, interoperable (Fortran, C/C++, Python) and flexible, and required changes in ICON are minimised. During the development of ComIn the ease of getting started and the experience during plugin development were guiding principles to provide a convenient tool. The extensive documentation and a variety of test and example plugins are results of this process.

This paper motivates the underlying design principles and provides some concrete reasoning for their selection. Further, current limitations are discussed and the vision for the future is presented.

1 Introduction and motivation

The overarching motivation for the ICOsahedral Non-hydrostatic model system (ICON) Community Interface (ComIn) is to facilitate the extension of the ICON host model by so-called plugins. Plugins can range from individual externalised features, like diagnostics or output functionalities, to full Earth System Model (ESM) components, as ocean or land models. ComIn supports scientists in extending ICON for their scientific question and provides access to the ICON data structures, without affecting ICON's operational use for numerical weather prediction (NWP). The interface aims to minimise the amount of code changes within the host model ICON when adding an extension and removes the need for users to modify the host model directly. ComIn's application programming interface (API) is designed to provide a stable interface, which in turn significantly reduces the maintenance required for plugins when switching to new releases of the host model. The inclusion of ComIn in the ICON repository and the co-development of ComIn with ICON are relevant factors for this. The ComIn API includes multi-language support for Fortran, C/C++ and Python.

The newly developed interface offers advantages compared to existing options for extending ICON, like adding features directly into the host model (internal coupling), via external coupling, or through integrated frameworks (IFs). These approaches typically require larger modifications of the host model code and, in case they are not integrated into the main repository, frequent updates are necessary while the host model evolves. Furthermore, through dynamic loading it is possible to change the plugins attached to ComIn without recompilation, and for plugins written in Python no compilation is required at all. It should be noted that ComIn also supports external coupling through couplers and IFs as plugins and thus brings together a variety of methods. As with all existing extension methods for the ICON code, it is the responsibility of the developer to ensure optimal resource usage of any extension, also with ComIn.

Before presenting use cases and more detailed insight into ComIn, background on ICON and the existing options to extend ESM components are briefly presented.

1.1 The host model ICON

According to the ICON website (ICON Partnership, 2024), ICON (Zängl et al. 2015) is a "flexible, scalable, high-performance modelling framework for weather, climate and environmental prediction". ICON is developed by the German Climate Computing Center (Deutsches Klimarechenzentrum, DKRZ), the Deutscher Wetterdienst (DWD), the Karlsruhe Institute of Technology (KIT), the Max Planck Institute for Meteorology (MPI-M), and the Swiss Center for Climate Systems Modeling (C2SM). The atmospheric component of the model system is based on a non-hydrostatic dynamical core for application over a broad range of temporal and spatial scales, i.e. for high-resolution Large Eddy Simulations (LES), NWP, and global general circulation model (GCM) based climate projections. The ICON website (ICON Partnership, 2024) also summarises ICON's role as a model that provide "actionable information for society", "advances our understanding of the Earth's climate system" and "allows [...] users to solve challenging problems of high societal relevance".

The spatial discretisation is realised on an unstructured triangular C-grid, which is derived from a spherical icosahedron by iterative refinement. Details on the discretisation of the equations of motion on the triangular C-grid and the numerical implementation of the non-hydrostatic dynamical core are described by Zängl et al. (2015). ICON comes with a set of parameterisations for physical processes, ranging from radiation, orographic drag, turbulence, cloud and convection processes to parameterisations describing atmosphere-land interaction. A detailed overview is beyond the scope of this article. More information on ICON's dynamical core and physical parameterisations, including a publication list, is available from the ICON model website (ICON Partnership, 2024). The ICON Tutorial (Prill et al., 2024) also provides useful information and includes a section on ComIn. Some aspects relevant to this article are listed below.

– The notation R*n*B*k* for the resolution of ICON's icosahedral grid denotes *n* root division steps of the icosahedron and *k* bisection steps of the resulting triangles. Details can be found in Section 2.1 of Zängl et al. (2015).

– In ICON's dynamical core a two–time level temporal discretisation (the predictor–corrector scheme) is applied.

– Within the dynamical core, ICON uses a fast time stepping to solve the fully compressible non-hydrostatic Navier-Stokes equations. Outside of the dynamical core, physical processes are divided in fast physics and slow physics with different temporal integration using the basic time step Δt of the simulation, which is set by the simulation configuration (Zängl et al. (2015) and Prill et al. (2024), especially Figure 3.8). Fast physics processes are the saturation adjustment, the surface turbulent transfer scheme, the land-surface scheme, the turbulent vertical diffusion, and the microphysics. Each fast physics process is calculated every time step $t_{\text{fast}} = \Delta t$ and the model state is updated sequentially by each process (time splitting or sequential-update split). Convection, subgrid-scale cloud cover, radiation, non-orographic and orographic gravity wave drag are considered as slow physics processes. These are called less frequently, with larger time steps $t_{\text{slow}} = C \cdot \Delta t$, which can be set for each of the slow physics processes individually as multiples of the basic time step (with constant integer C) and are rounded up automatically if t_{slow} is not chosen as multiple of the basic time step. Tendencies from each of these processes are updated with this lower frequency, they are kept constant between two subsequent calls of each process, and applied to the model state independently of each other (in parallel-split manner). Depending on the prognostic variable, tendencies are applied either inside the dynamical core (for the edge-normal velocity and the Exner pressure) or as part of the tracer advection (for mass fractions), but at each basic time step Δt. This approach may lead to a slow physics process being updated during time step t_i when the tendency based on time step t_{i-1} was already applied earlier during the time step.

– For each Message Passing Interface (MPI) process in ICON, data is only available in the region attributed to it as part of the parallelisation, i.e only the process-local part of a data field can be accessed. This restriction enhances the scalability of ICON as the communication necessary during a simulation is reduced.

– Within parameterisations (and in other parts of the code) the access to grid data is structured in two nested loops. The outer loop is often called *block-level loop*. This splitting of *DO* loops over grid elements is implemented for reasons of cache efficiency and also allows to optimise code for Graphics Processing Units (GPUs).

1.2 Common methods for extending an ESM model component

It is important to understand the ComIn approach within the context of already existing methods for model extension. Therefore we briefly review existing ones, which all have their validity and use cases based on their properties. This overview is kept general in terms of the host model. Specifics of ICON and the setup with ComIn are discussed in Section 1.3.

1. The simplest way of connecting codes of different complexity, ranging from individual subroutines to comprehensive ESM components, is the **internal coupling**, in which different components are part of the same program unit and exchange data via the working memory (i.e. utilising the same computational resources as the host model). New features, added directly within the code (red circle within the large blue circle representing the host model in Fig. 1a), need to follow the model internal data structures, which can be accessed directly. It is a very efficient way of interacting with

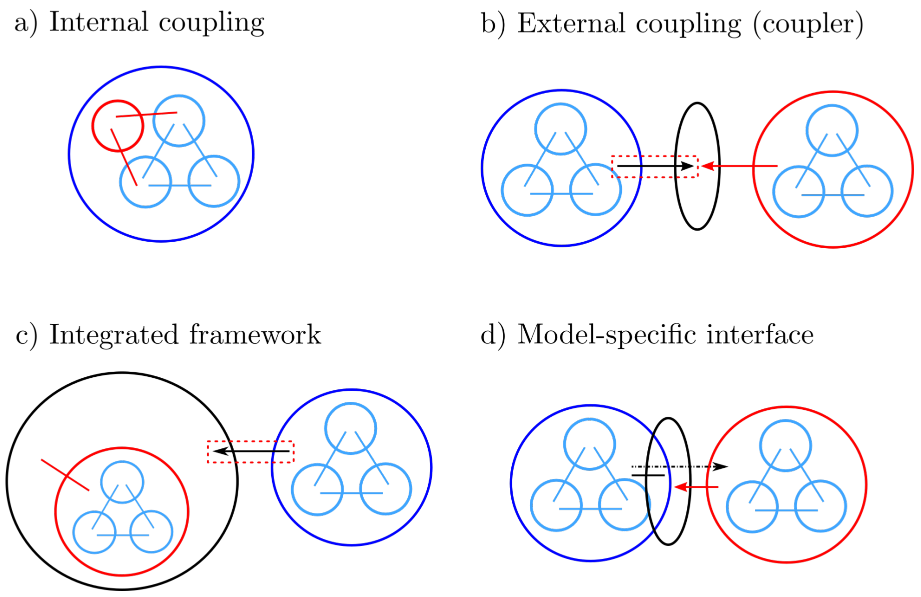

Figure 1. Overview of model extension methods. In each approach one ESM component acts as the host model (blue circle with smaller light blue circles indicating internal functionalities like parameterisations and dynamical core), which can be extended by these four methods. Extensions are highlighted by red colour and software to support these extensions in approaches b) to d) by black circles. Red circles are set around new ESM components (approach b)-d)) or smaller extensions (approach a)). Straight lines indicate exchange of data and access to routines. An arrow is added if access to routines between components is only possible one-way. For example software A can call routines of software B but not vice versa, where software A could be an ESM component and software B a coupler. In this case, the arrowhead is located at the software B (the coupler), to indicate the software, from which routines can be called. Red dashed rectangles indicate that additional infrastructure is required for each host model component when it becomes connected to the coupler or IF for the first time but that the corresponding coding effort is only required once. The red line in approach c), connecting the new component and the IF, indicates that the component needs to be integrated into the IF. For a model-specific interface (approach d)) the dot-dashed line from the host model to the new component visualises the fact that routines are called, but only indirectly via the interface itself (callback mechanism introduced in Section 2).

an ESM component, as data exchange can be done via memory (see red lines in Fig. 1a) and cache-optimisation at the CPU core level is possible. Although this approach requires the least additional infrastructure for individual changes, it has a major disadvantage. Especially for ESM components with time-critical applications (e.g. numerical weather prediction and flood modelling), code modifications need to fulfil stringent constraints on memory consumption and runtime. Modifications, which are not fed back into the repository of the host model require frequent updates of the host model on the user side. The last two points are also true for the other two methods described in this section (Figs. 1b and c).

2. With **external coupling**, and only considering on-line coupling here (in contrast to exchanging information offline between components via writing to and reading from files), information is exchanged between different program units (i.e. the program units can request and use their own computational resources). This approach is traditionally used for coupling ESM components of different domains (like ocean, atmosphere, land, sea-ice, etc.), and originally external coupling was designed with this application in mind. The aim of a coupler (i.e. a software to support external coupling) is to perform data communication between the externally coupled software components, while requiring as few changes as possible in the existing components and be able to out-source the task of grid transformations for multiple components. A coupler can connect ESM components of different domains which are separately developed as individual models (see also Fig. 1b). It also provides advantages when connecting ESM components acting on different timescales. Additionally, couplers have well-defined APIs and avoid namespace conflicts, i.e. they even support coupling one ESM component with itself. However, load balancing and debugging are typically complicated and most approaches are limited to the exchange of 2D model fields. When the setup requires a large amount of model fields to be exchanged, this method can create significant additional overhead compared to sharing the information directly via working memory. Furthermore, the data to be transferred depends on the experiment setup and other component(s) connected to the coupler. Even if a coupler is integrated to an ESM component, new coupling setups might require updates in the data preparation in all connected components.

   Two examples of coupler libraries are OASIS3-MCT (Craig et al., 2017) and YAC (Hanke et al., 2016). Both are designed for the purpose of coupling ESM components of different domains, and both are limited to some extent to 2D data (YAC can transfer 3D data but is limited to 2D interpolation). Other examples are the CCSM Coupler Version 7 (National Center for Atmospheric Research, 2024b), the MpCCI mesh-based parallel code coupling interface (Joppich and Kürschner, 2006) and former versions 3 and 4 of the already mentioned OASIS (Ocean Atmosphere Sea Ice Soil) coupler (Valcke, 2013; Redler et al., 2010). A special case is the Multi-Model Driver (MMD; Kerkweg et al., 2018) for hierarchical on-line nesting as part of the Modular Earth Submodel System (MESSy, namely of COSMO/MESSy into ECHAM/MESSy), which also supports exchange of 3D data, including grid-transformations. Another overview of couplers has been published by Valcke et al. (2012).

3. The third approach is the application of an **integrated framework** (IF), visualised in Fig. 1c. IFs provide generic data structures and methods (sometimes called model infrastructure) for building comprehensive models out of individual

(ESM) components, which become interoperable by utilisation of these data structures and methods. The level of control is usually finer (e.g. individual parameterisations can be addressed) compared to the coupler approach (which acts on the domain-level, see Fig. 1b), thus providing more flexibility. IFs offer well defined APIs, which are independent of any specific model component, a fact which, however, also implies a drawback. Additional effort is required to associate data structures of a host model (often called legacy models in the context of IFs) with the data structures of the IF (red dashed box in Fig. 1c). Still, the coding effort is minimised compared to direct internal coupling, since the translation of the host model data structures to the host-model agnostic IF data structures needs to be performed only once.

Several integrated frameworks have been developed and are commonly used. Examples are the Earth System Modelling Framework (ESMF; Earth System Modeling Framework, 2024), MESSy (MESSy Consortium, 2024), the Community Earth System Model (CESM; National Center for Atmospheric Research, 2024c), the Flexible Modelling System (FMS; Geophysical Fluid Dynamics Laboratory, 2024) and the Common Community Physics Package (CCPP)-framework (National Center for Atmospheric Research, 2024a; Heinzeller et al., 2023).

1.3 A new tool for extensions to ICON: the community interface

None of the approaches 1 to 3 described in Section 1.2, and summarised in Fig. 1, offers the desired flexibility to extend the ICON model, while providing access to the host model's data structures (i.e. similar to internal coupling), and introducing minimal changes to ICON. This flexibility in extending ICON is beneficial or even required by applications such as

– simple diagnostics, which are computationally inexpensive (compared to the ICON model),

– complex atmospheric chemistry, which requires a tight coupling to the atmospheric physics, e.g. for tracer transport (by large scale advection, convection and turbulence), radiation, and cloud microphysics,

– land models with a tight coupling to the hydrological cycle of the atmosphere.

All three approaches to extend the existing ICON model are intrusive (1: due to direct internal coupling, 2: due to calls to coupler routines and exchange of data, 3: due to calls to IF routines and translation of data structures to the IF), even though the coupler and IF approach aim to minimise the intrusion. Once an IF or coupler is integrated into ICON, and especially also the ICON repository, an additional intrusion when connecting a new ESM component is reduced for any further extensions (red dashed rectangles in Fig. 1b and c indicate which steps only need to be performed once). As the ICON consortium is a closed consortium, with gatekeeping for quality control reasons, inclusion into the ICON repository is restricted (currently to the YAC coupler) to prevent additional maintenance overhead. Regular updates of the extension in response to ICON developments are required in any case, both in- and outside of the ICON repository. However, this process is simplified for extensions included in the ICON repository as regular consistency checks are automatically done and new contributions need to work alongside of the extension. Updating the ICON version takes much more time for extensions maintained outside of the repository and the likelihood of introducing errors is increased, as small changes can lead to incompatibilities between ICON and the other ESM components.

ComIn intends to fill this gap of extension methods which are minimally intrusive and which cause low maintenance overhead. It aims to further reduce the amount of changes within ICON when connecting plugins via the ICON-specific interface. Update efforts on the plugin side in response to new ICON versions can be considerably reduced with the stable API provided by ComIn. Developments of plugins and the host model are essentially disentangled from each other. Even so, ComIn also requires changes in the ICON code in response to newer versions of ICON. Thanks to the inclusion in the ICON repository and the co-development with ICON, the amount of work to maintain ComIn in ICON is minimised. External couplers and IFs can also be attached as ComIn plugins to ICON and thus benefit from the reduced maintenance offered by ComIn.

Another interface library is currently under development at the European Center for Medium-range Weather Forecasts (ECMWF). The *plume* library's (plugin mechanism; European Centre for Medium-Range Weather Forecasts, 2024, Bonanni and Quintino, 2023) host model is the ECMWF's Integrated Forecasting System (IFS) and the library allows to load plugins dynamically at runtime (via Plugin Manager) and offers access to their data during model runtime (via Plugin Data). To this end, plume provides APIs that control the dynamic loading during runtime and access of the model data for the plugins. Plugins can be individual models, data analysis or individually implemented specific calculations. The *plume* library is set up to be compatible with C++ and Fortran and already includes some example plugins and an NWP emulator to test the interaction of the plugins. In general, implementing an interface to a model system depends to a large extent on the model system's data structures and control flow. However, the design concepts applied are expected to be largely independent and thus easily transferable to other models.

2 What does ComIn offer?

ComIn allows plugins to be called by ICON and to access data and metadata of the host model. Additionally, it is possible to register further variables to be added to the ICON list of data, which are then available during runtime in memory and can be added to the output. Despite these extensive functionalities, the ambition of ComIn is to provide a lightweight interface which requires minimal code changes in the host model and plugins. Furthermore, the compatibility of the API with regard to different ICON releases minimises the expected maintenance and migration effort on the plugin side. To support language interoperability, ComIn is currently written in Fortran and can be used by plugins in other languages such as C/C++ and Python via the use of ISO-C bindings.

ComIn is developed to work together with ICON but maintained separately from ICON and creates a well-defined interface between ICON and any plugin. It enables ICON-specific internal coupling at runtime (at compile time, the host model and plugins are treated as different program units) which is formalised through the interface. ComIn is not a traditional coupler, as it does not just handle data exchanges but also supports integration into the ICON control flow. Fig. 1d) visualises both, the separation of ICON and the plugin (full black line and red line with arrow), and the integration into the ICON control flow (dot-dashed line with arrow). The exchange of data is done via the working memory. For ease of use and because the different program units are developed separately, there is not just one executable compiled. Instead, ICON and a plugin are both separately compiled (and linked) with the ComIn library; ICON as the executable, the plugin as a shared library (note that this is not visualised in Fig. 1d). At runtime, ICON dynamically loads the shared library of the plugin (and ComIn) via ComIn on demand. Several plugins can be individually compiled with ComIn and dynamically loaded at run-time within the same simulation. Callbacks to ComIn plugin functions are implemented as blocking calls (as part of an MPI parallel job, but not asynchronously), so the execution of ICON is paused until the callback returns. ICON loads a plugin as a shared library and provides information primarily on a per-task basis. Depending on their specific implementation, plugins can be executed serially within the ICON control flow, using the same computational resources (i.e. cores/nodes requested for the simulation) as ICON. This is the case if a plugin does not further specify a (parallel) setup. However, plugins can also be implemented to use additional computational resources in an MPI environment (e.g. via a coupler) and be executed asynchronously, spawning new tasks and immediately returning control to ICON. In all these possible approaches, managing the computational demands of a plugin lies in the responsibility of the plugin developer, as with any ICON internal development. ComIn itself provides access to variables and their meta-data, grid information, and information on parallel decomposition and MPI communicators. However, it does not provide any tools to access performance or manage load balancing.

As background for the discussion of the ComIn design principles in Section 3, a brief summary of the main components and terminology for ComIn is provided here, and also in the glossary at the end. A documentation of the ComIn procedures, data types as well as usage instructions are offered in a white paper as part of the ComIn repository (ComIn authors, 2024) and as a supplement to this article. ComIn itself consists of two components:

– The callback library (see schematic diagram in Fig. 2) gathers plugin routines to be executed at specific entry points from the host model. At these entry points, located throughout the model, ICON executes function callbacks. Entry points during the initialisation phase, the time loop, and when finalising ICON can be addressed by a plugin. While not all entry points need to be associated with a plugin routine, it is possible to attach one plugin to several entry points and several plugins to the same entry point. It should also be noted that, depending on the settings of the ICON simulation as given by the ICON namelist, not all entry points are necessarily always reached.

– The adapter library (see schematic diagram in Fig. 3) organises data sharing between the host model and the plugin(s). ComIn exposes (i.e. shares) pointers to ICON's own data fields and provides access to their metadata. Data structures containing information on, for instance, domain specific settings and the parallelisation are mainly shared as pointers as well. These data structures are termed descriptive data within the context of ComIn. ComIn also supports registering additional data fields from a plugin to ICON's variable list, and ICON then determines internally if they are added to its internal memory. The metadata of these additional fields can be defined by the plugin. As with native ICON fields, they can then be accessed via the adapter library. Registration of additional data fields is mainly intended for atmospheric tracers, to add fields from a plugin to the output of the host model, and to share data between two different plugins (e.g. when a diagnostic plugin analyses results of another plugin).

In summary, ComIn aims to minimise the effort to maintain code that is tightly integrated with ICON as a plugin, secures past and future time investments in code development by plugin users, and simplifies sharing of code between different communities.

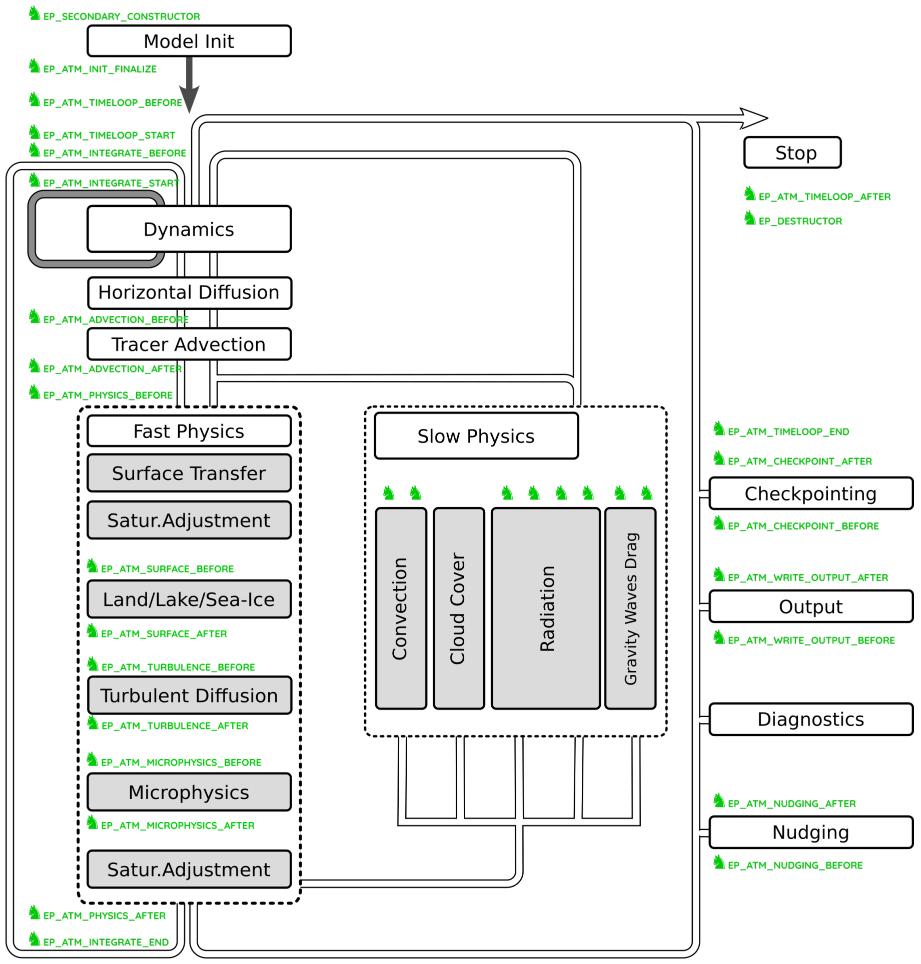

Figure 2. Schematic representation of the integration of entry points into the ICON control flow. At specific entry points, plugin subroutines can be called from within ICON, if they have been registered for these entry points through the ComIn library. After initialisation, the model's integration is conducted in a time loop (main part of the schematic). The time integration can be divided into three parts: (i) the dynamical core "sub-stepping" (black) with a time step shorter than Δt, (ii) the fast physics processes loop (red), with the models basic time step Δt, and (iii) the slow physics process loop with a larger time step than Δt, depending on the slow physics process. Entry points are marked in green and for some, names are provided. The naming scheme and placement is further discussed in Section 3.6 and the white paper (see e.g. in the Supplement) provides a full list of entry points.

3 Main ComIn design principles

The main design principles of ComIn cover the two, sometimes contradictory and strongly related areas of (i) minimising the impact of ComIn on the host model, and (ii) maximising flexibility of ComIn for one or more plugins, and state that ComIn should be lightweight, interoperable and flexible.

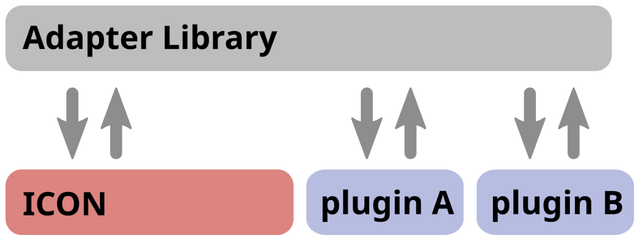

Figure 3. Schematic representation of the adapter library. The adapter library gathers all data structures required to describe the ICON state and externalises them from ICON. Through it all information is exchanged between the adapter library and a plugin, adding some independence from the description of data structures in ICON. Specifically, this also implies that a plugin can be compiled and linked to ComIn (including the adapter library), linking to ICON (built with ComIn) is replaced by loading the shared library/ies of the plugin/s at runtime.

The main motivation for minimising the impact of ComIn on ICON is to simplify the maintenance of ComIn (and ICON), and to increase the acceptance for ComIn at the ICON core institutions and general user base. The impact is measured, for example, as the difference in memory and runtime of a setup with ICON/ComIn (plus plugin(s)) compared to an ICON setup without ComIn. The number of lines added to the ICON source code is considered as well. For the about half a million lines of Fortran code present in ICON (*src/*), less than 1000 lines (around 870 lines) were added related to ComIn (excluding comments in both cases and runscript/namelist updates). Details on the technical evaluation (memory, runtime and evaluation of overhead) are provided in Section 4.

Furthermore, introducing the option to fully disable ComIn within ICON caters to the two very different use patterns of ICON: the operational NWP model, which requires negligible impact and vanishing risks introduced by any extensions, can be run with ComIn disabled. At the same time, with ComIn enabled, the code can be flexibly extended beyond the operational NWP setup to suit the scientific research community. Therefore, all expansions and calls related to ComIn are encapsulated in pre-processor directives and ComIn can be fully disabled at compile time within ICON to avoid even minimal overhead and the risk of negative side-effects.

The second main goal of ComIn is to support the user community of ICON, while also making ICON and ComIn attractive to new users, for example, for scientific research beyond NWP. For this reason, several functionalities of ComIn have been developed to support the extension of ICON for a variety of use cases.

This section provides motivation for the design principles of ComIn and information on how they are implemented.

3.1 ComIn adds minimal restrictions on ICON (lightweight)

During the initialisation phase of ICON, the sequence of setup calls of ICON and ComIn are intertwined. In particular, this concerns questions like: Until when can additional fields be registered?, From which point are the descriptive data available?, or When can pointers to data fields be requested? The entry points of ComIn therefore add restrictions to the potential re-location of ICON routines during the initialisation phase. Any restrictions arising from the integration of plugins via ComIn are kept to a minimum and communicated to the ICON developers.

Atmospheric tracers, such as ozone and methane, can be added to ICON's variable list by registration via ComIn. During the model integration they can be subject to different types of advective tracer transport schemes, diffusion and convection processes in ICON, which are set by the plugin via ComIn metadata methods. To support the modularity and use with ComIn, the setup of additional tracers in ICON had to be made more flexible. Instead of defining the (total) number of tracers in the ICON namelist, the tracer numbers are now determined at runtime, depending on the dynamically loaded plugins and the additionally requested tracers therein.

3.2 ComIn and a plugin can be built without ICON (flexible)

To simplify and shorten the build process, especially if the selection of plugins changes frequently, it is advantageous to support the compilation of a plugin with ComIn independently of the host model. By separating the variable list and data structures from those of ICON and sharing them via the ComIn adapter library, a plugin can be built by linking just ComIn. Only at runtime does ICON dynamically include any activated plugins. The callback and adapter libraries thus enable a separation of concerns and provide a modular approach for the connection plugins to the host model.

Ensuring compatibility when building ICON and plugins separately with ComIn, and thus potentially several versions of ComIn at the same time, is handled by ComIn version checks during the initialisation phase of ICON. The ComIn API design is independent of the ICON version. ComIn versioning follows the Semantic Versioning[1] and ComIn is released in major versions, minor versions and patches. A major version indicates that the interface was changed in such a way, that backward compatibility is not guaranteed, but plugins remain compatible with the ComIn API for different minor versions. Checking compatibility on the plugin side is encouraged but not enforced by ComIn. Similarly, consistency of the definition of standard data types, such as floating point precision, between ComIn and plugin(s) can be tested.

3.3 ComIn offers multi-language support (interoperable)

To immediately enable a wide range of applications to use ComIn, the first release of ComIn supports language interoperability (from the Fortran core implementation to C/C++ and Python). ComIn procedures are either defined to be natively interoperable in Fortran with the *BIND(C)* attribute or, where this is not feasible due to the handling of strings or use of dynamic memory (pointers or allocatable arrays), corresponding routines for C/C++ and Python access are provided. The routine interfaces are kept as similar as possible to assist users working with ComIn in different programming languages. In addition, the ComIn white paper (ComIn authors, 2024) and the Doxygen documentation directly compare how to use the API with different languages in translation tables. Two simple python plugins (see *plugins/python_adapter/examples/* in the ComIn repository) showcase the simplicity and readability with which plugins can be implemented in Python, using the so-called Python adapter.

The Python adapter itself is implemented as a ComIn plugin written in C++ using the C language bindings of ComIn. It is shipped with the ComIn library and embeds the Python interpreter so that the actual Python script is interpreted at runtime. The

[1]Semantic Versioning 2.0.0, https://semver.org/, last access: 21.11.2024

filename of the Python script is passed to the adapter as an option in the ICON *comin_nml* namelist. Callbacks are realised as custom Python functions that are registered with ComIn by adding a function decorator. This makes it easy to define callbacks on the fly. Furthermore, no recompilation is required during plugin development. Access to the variable data is provided via NumPy/CuPy objects, which act as pointers to the actual data. The other API functions are also exposed in the Python module *comin*, which can be imported into the script. The Python language bindings allow for a variety of practical applications, including visualisation and machine learning. The wide range of Python packages available allows rapid prototyping.

The available examples for Python and C demonstrate the interoperability. The C interface is used in a YAC example plugin. By supporting interaction with the YAC coupler, the ComIn API does not only provide interoperability in terms of programming languages, but also in terms of coupling types (i.e. moving from extension method in Fig. 1b to d). ComIn thus offers a pathway of maintaining the YAC coupler outside of ICON.

3.4 ComIn provides a minimalistic adapter library (lightweight)

As a way to keep the adapter library as lightweight as possible, the shared variables and data structures are limited to the essentials, the meaning of "essentials" is defined below for each topic.

To make the interface as stable as possible long-term, functionalities and properties of ICON, which might be adapted or removed in the future, are not shared.

3.4.1 Essential descriptive data exposed via ComIn

ComIn delivers all the information on the MPI-based parallelisation (communicators, grid decomposition information) to enable a plugin to set up communication and data transpositions (broadcasting, gathering, scattering, boundary exchange, etc.). For the selection of any other descriptive data component the guiding principle is to avoid redundancy. From the view of the descriptive data, "essential" thus means that enough information is exposed (i) to support plugins setting up their own parallelisation (a plugin can still also operate on the same transposition as ICON) and (ii) to reconstruct omitted descriptive data within the plugin.

A very simple example is that ComIn does not provide access to `nlevp1`, which is the number of vertical half levels. This parameter can easily be re-calculated from the provided number of full levels `nlev` as `nlevp1 = nlev + 1`. Another simple example is the fact, that the Cartesian coordinates of the model grid are not separately shared but this information is only exposed via the longitude/latitude fields as part of the descriptive data. Similarly, several MPI communicator properties, which might become obsolete when changing to a different parallel implementation, and which are not required by plugins to set up the communication and data transpositions, are not exposed.

3.4.2 ICON variables are available for one-time level

Direct access to the updated variables at the time step used inside the dynamical core is not available in ComIn. The entry points to the ComIn callback library inside ICON's time loop (see Fig. 2), i.e. considering both, fast- and slow physics, are called at each time step Δt. This means, after each fast physics process, the model variables correspond to the sequentially updated state. After the slow physics processes, model variables are not directly updated by the slow physics tendencies. This update is only applied at the next call to the dynamical core or tracer advection, depending on the prognostic variable for which the tendency is added.

"Essential" data are thus pointers to variables and tracers at the current time level[2] of the host model. During the time loop the pointers reflect the sequentially updated value from the fast physics processes and the slow physics processes are added at different times, but are present in the fully updated state at the end of the time step. This behaviour is intended, but the developer of a plugin may keep in mind that the tendencies from the slow physics processes are calculated and used to update the model state at different parts of the time step.

Not sharing data at the internally used two-time levels means that the time integration scheme of ICON can be updated without requiring major changes of ComIn, and consequently of the plugins. More importantly for the plugin users, this means that no knowledge of the complex time-stepping scheme in ICON is required to avoid erroneous results. The implementation is such that when requesting a data field, a pointer wrapper is provided via ComIn, i.e. a structure which contains a pointer. The pointer underlying the pointer wrapper is updated to the current time level state at the requested entry points.

3.4.3 ICON fields available on process-local MPI partitions

To allow ComIn users to benefit from the optimised ICON scaling, ICON fields are shared essentially as they are set up in ICON, i.e. only the MPI-local information on each process. This approach prevents a huge communication overhead which would limit scalability of ICON, but still exposes all the data available.

3.5 ComIn adds minimal memory overhead (lightweight)

The implementation of how the ICON data fields and descriptive data are made available to ComIn impacts the overhead and usability of ComIn. The smallest memory overhead is achieved when sharing pointers to the ICON memory addresses. However, this adds the possibility that a plugin can (inadvertently or incorrectly) change the value of such a field. Via the usage of Fortran pointers, no access restrictions are implemented, as these are not supported by the language standard. Implementing memory access restrictions would add additional undesired execution overhead. It was thus decided to share pointers to data fields via ComIn without safety mechanisms.

So far the memory argument (ComIn should be lightweight) was considered more important also in the case of descriptive data, even though most descriptive data are constant in time and their size not excessive. Thus, with a few exceptions, pointers to the ICON descriptive data are shared directly. Another advantage of this approach is that inconsistencies in data copies can be prevented. Exceptions include the cell properties longitude and latitude, as their storage format has been simplified in ComIn to eliminate a further dependence on the host model's data structures. It is thus the responsibility of the user to ensure that the descriptive data are handled correctly and that descriptive data are not accidentally modified.

[2]See Section 1.1 for two-time level scheme.

3.6 ComIn plugin routines can replace ICON parameterisations (flexible)

The guideline for the placement of the entry points during the time loop was to allow maximum flexibility for the plugins. A plugin should be able to modify the input going into an ICON parameterisation and read the modified state afterwards, or alternatively to replace an existing parameterisation of ICON by a parameterisation of the plugin. The entry points during the time loop are therefore mostly located before and after physics parameterisation, outside of any *IF*-guards checking if a particular parameterisation is enabled. This means that entry points around parameterizations are reached even if they are disabled in the ICON namelist. Following this approach a few redundant entry points were created if two parameterisations are executed directly in sequence. In this way intuitive naming of the entry point associated with the respective parameterisation is possible, i.e. *EP_ATM_TURBULENCE_BEFORE* for the entry point (EP) before the (atmospheric) turbulence scheme. At the same time the likelihood is increased that an entry point is located at the same place relative to a parameterisation even after code changes in ICON, for example when adding another parameterisation. The very comprehensive addition of entry points during the time loop, i.e. for every physical parameterisation, enables a wide range of applications to use ComIn.

Plugins can have very different requirements on the interaction with ICON. It is thus, for instance, possible to disable the turbulence parameterisation within ICON and use the entry point *EP_ATM_TURBULENCE_BEFORE* to instead call the turbulence scheme of a plugin. In another model setup, the user wants to execute the ICON turbulence scheme but adapt settings in *EP_ATM_TURBULENCE_BEFORE* before the ICON turbulence scheme and then retrieve modifications afterwards at the entry point *EP_ATM_TURBULENCE_AFTER*. With the chosen entry point locations both use cases are supported.

As an intended design constraint, all ComIn entry points are placed outside of the block-loop level[3] in ICON. Entry points below the block-loop level would require an extension of the callback functionality to support both, the current MPI-collective callback, and one on the sub-block-loop level. Such a callback thus leads to additional communication overhead and necessitates another set of descriptive data to describe the current block-loop instance. Entry points below the block-loop level could enable direct interaction with existing parameterisation, as access to local variables would be possible. However, the large list of drawbacks, which would make ComIn less lightweight, outweighs the potential benefits at the current stage.

3.7 A ComIn plugin can set up additional MPI parallel communication (flexible)

As mentioned in Section 3.4, the ICON-internal MPI communicators and exchange patterns are not exposed. However, within the descriptive data all relevant MPI information is made available to be able to set up data transpositions in a plugin. The MPI communicator, which contains all MPI tasks taking part in the ComIn initialization routines within ICON, is shared via ComIn. In addition, each plugin can specify a name of a communicator which is then created at initialisation. External processes can use this name in the MPI handshake[4] to be part of this communicator. In this way users can create communicators to exchange data with external processes via MPI, for example, to operate in a different parallel decomposition (on the same computational

[3]see Section 1.1 for a definition

[4]https://gitlab.dkrz.de/dkrz-sw/mpi-handshake

resources as ICON), or execute plugins on different computational resources than ICON, e.g. driving external couplers via ComIn.

3.8 ComIn supports several plugins to be connected simultaneously (flexible)

The callback and the adapter library were developed specifically to support the connection of several plugins to ICON within the same simulation. A potential use case is a land model plugin which is connected to ICON together with a diagnostic plugin to determine parameters during runtime, which are not natively calculated in either the land model or ICON. Such a setup could work with two plugins connected to ICON via ComIn.

To easily put together a flexible setup of plugins, ComIn is built as a shared library together with each potential plugin. As the decision on the finally used plugins is set via the ComIn namelist, no re-compilation in response to modifications of the setup is necessary. Plugins are then loaded dynamically at runtime within ICON by ComIn in the order they are defined in the *comin_nml* section of the ICON namelist (which is listed in the white paper as part of the repository (see ComIn authors (2024)) and as supplement to this article). Currently the order in which plugins are executed is the same for all entry points. Additionally, requested variables can be shared across all plugins, but the adapter library also allows to create additional fields, which are exclusive to just one plugin. In summary, the ICON variable list can be extended and customised to fit different plugin setups and their combination.

3.9 ComIn ships with tests and simple use cases

In order to aid the switch from ICON to ICON/ComIn for users, ComIn is kept as simple as possible and developed with various examples demonstrating its use "in action". The documentation in form of a manual (see appendix and ComIn authors (2024) for an up-to-date version) and of example plugins is of course also advantageous for new users of ICON.

The functionality of the available ComIn routines is showcased in short technical examples (i.e. not necessarily physically meaningful). These are set up and maintained to cover all *IF*-guards implemented in the code, both through tests cases intended to succeed and tests intended to deliberately fail. For example, the tests indicate at which points during a simulation new variables can be requested or pointers to existing fields can be obtained, but also ensure that an error is reported when additional variables are registered too late during the initialisation phase. These examples are also used for testing as part of the CI (continuous integration) pipeline, which provides automated build and runtime tests and supports the ComIn development process and code review. In addition, a few simple use cases were developed as a reference for plugin developers implementing ComIn in their own model or software. These example plugins inspired by real-world usage (i) calculate an average across all MPI ranks to get global mean temperature (C and Fortran plugins), (ii) add new microphysical variables (liquid and ice water path) and calculate column total statistics (Fortran plugin), (iii) add emissions from a point source to the list of ICON tracers (Python plugin as part of the Python adapter/Python API shipped with ComIn) and (iv) retrieve ICON input fields via coupling to YAC (C plugin). The first test case uses YAXT (Behrens et al., 2024) for MPI communication and confirms that the ComIn descriptive data contain enough information to set up ICON-independent MPI communication.

4 Computational aspects

The test simulations to measure the overhead from ComIn are based on a typical operational NWP setup. This time-critical setup requires maintaining a minimal additional overhead. Currently such a simulation is run with ComIn disabled so that no change to the operational behaviour is expected. The 7.5 day forecast on a global R3B06 grid (approximately 26 km) with a nested R3B07 domain over Europe (approximately 13 km) is typically run as part of the ICON ensemble (ICON-EPS) with 40 members. The forecast lead time varies from short to medium range forecasts and here the longest lead time of 180 hours is selected.

This model setup was executed for ICON and ICON/ComIn, i.e. the only difference is, whether data preparation for plugins and execution of callback routines is included, which are encapsulated by ComIn pre-processor directives. For measuring the overhead added by ComIn itself to ICON, no plugin is attached to ICON via ComIn in this setup ICON/ComIn. So, no callbacks are triggered at any entry point but since the ComIn setup and data preparation are still executed, the overhead of ComIn to ICON can be determined. This setup thus practically corresponds to an extremely efficient plugin (i.e. one without any runtime or memory overhead). It should be noted, that in realistic configurations with ComIn plugins, total performance is also impacted by the computational efforts from the plugins themselves. For both model configurations, a setup with output and without output was run on the DKRZ[5] high-performance computing (HPC) system Levante. Based on the output, it was confirmed that ComIn does not impact the ICON simulation results. Three additional simulations per setup (ICON and ICON/ComIn) without output were used for the performance tests. The additional overhead in terms of runtime (elapsed time and total max time from the ICON timer) is approximately 1.5%. The additional memory overhead considering the maximum memory required, sampling once per second, is similarly 1.5%. The variability among simulations is relatively large even at this sampling frequency so that the range of standard deviations overlaps and indeed in two ICON/ComIn experiments the maximum memory is lower than in any of the ICON setups. Considering the average memory requirement (sampled every second) the increase using ICON/ComIn is slightly lower at 0.8%. Again the variability is larger than the average difference between the two setups.

It can thus be concluded that even with ComIn enabled as part of ICON, the additionally required memory and runtime are small.

5 Example applications and use cases

The motivation for the development of ComIn is to support internal coupling of external ESM components into ICON, while keeping the intrusion to ICON minimal. The implementation of ComIn in the integrated framework MESSy (Jöckel et al., 2005, 2010) is currently ongoing as a complex example for the use of ComIn. However, during the development and from first user interest, it has become obvious that many more applications of ComIn are possible. The list given below is certainly not complete, but is expected to increase and diversify as the ICON user community applies ComIn to answer their scientific questions.

[5]https://www.dkrz.de/de

Before presenting examples and their motivation, it should be emphasised again that by design none of the scientific applications of ComIn outlined below interfere with the operational ICON model, when being connected via ComIn. Also, by design, any developments described below can be relatively easily ported between different ICON versions if they are based on ComIn. Additionally, language interoperability is an essential feature of ComIn, which enables many applications and use cases. Exemplary use cases, starting with some which are already implemented and tested, and also introducing some which are possible with ComIn (in no particular order), are:

– With ComIn **additional diagnostics** can be introduced into ICON much quicker, and with fewer side effects. Instead of the need to find an appropriate routine in ICON, the correct time-level, the procedure in which to add the new diagnostic to the list of ICON variables, the different locations in the code to add metadata, etc., a few calls to the ComIn API produce the same result.

– Via ComIn not only Fortran routines, but also C and Python routines, can be easily called during an ICON simulation. This means that **(online) visualisation** is possible without much effort, for example via Catalyst[6], a tool which supports in situ workflows. Online visualisation has many benefits, for example reducing the size of output files and thus the time it takes to write them. Additionally, online visualisation allows to screen the model state at every time step, which is often not possible with output files written per default at a lower frequency.

– Similarly to online visualisation to evaluate physical results of a simulation while it is ongoing, it is possible to perform **online performance analysis** during a model simulation, for example with the tools Prometheus[7] or Grafana[8]. Information on the computing resources consumed by a simulation (for example computing time and memory/energy usage) are available right away so that inefficiencies become apparent quickly, and a simulation with satisfactory performance can be continued and does not need to be restarted after the evaluation.

– **Fast-prototyping** can be used to develop parameterisations. Based on the ICON-independent API of ComIn, parameterisations can be developed outside of a complex NWP or climate model and extensively tested. Once the implementation of the parameterisation is finalised as a stand-alone code, it can easily be attached via ComIn for further testing and productive application in the complex numerical model. Fast-prototyping is possible in all the supported programming languages but especially efficient with the intuitive programming language Python.

– Instead of working directly with a full NWP/climate model, the model-specific interface simplifies the interaction with the model. This can be very useful for e.g. **teaching** and student theses. Students can relatively easily insert simple diagnostics into ICON and generate additional output without the need to make modifications directly in ICON. This benefits them individually in their studies and introduces them with a relatively low threshold to modelling. At the same time, they are trained to become the next generation of NWP and climate modellers, which is very valuable to the community.

[6]https://docs.paraview.org/en/latest/Catalyst/index.html
[7]https://prometheus.io/
[8]https://grafana.com/

- With YAC and an I/O server as plugin, **interpolated I/O** can be enabled. Instead of an extra step after writing output, the interpolation can be executed (asynchronously) during the ICON simulation, thus reducing the post-processing steps. Similarly, the setup of YAC and an I/O server can be used to pre-process datasets of various resolution and prepare them for the simulation.

- ComIn provides all necessary information to drive the YAC coupler. Use with other couplers was not tested but should also be possible. This means that ComIn facilitates **external coupling** outside of the ICON code. Individual coupler interfaces inside the ICON code are no longer required with ComIn, but instead couplers can be flexibly attached through ComIn.

- Using ComIn it is possible to **externalise ICON internal functionalities** (i.e. developments similar to the land surface scheme JSBACH (Reick et al., 2021) or the Hydrological Discharge model HD (Hagemann et al., 2020)) and to modularise ICON further.

- **Machine learning and artificial intelligence** applications are becoming more prevalent in NWP and climate modelling. Attaching a plugin to train a machine learning dataset (independent of the programming language) is possible via ComIn. As Python has a large collection of libraries and packages that are suitable for machine learning and artificial intelligence, this is a very relevant use-case of ComIn's Python API.

6 Missing functionalities, planned and envisioned developments

Although ComIn already provides a wide range of functionalities, it is clear that expansions and additional features can further increase the flexibility and usability of ComIn, while still keeping it lightweight. Several extensions for later versions of ComIn are currently planned and partly in preparation, of which some are briefly introduced here:

- ICON can be accelerated by using GPUs. Considering setups on CPUs (also called "host") and GPUs (also called "device"), possible configurations are thus (i) that ICON and a plugin are executed on the device, (ii) ICON runs on GPUs and a plugin on CPUs and (iii) ICON/ComIn and plugins only utilise CPUs. In practise this differentiation can be handled per entry point, i.e. ICON informs ComIn which entry points are integrated into accelerated code. ComIn thus needs to be able to receive information for each variable requested by a plugin, if it is needed on the device or not (the default). If a variable is requested on the device in configuration (i), no data transfers need to be initiated. Similarly, this is the case for configuration (iii). In case (ii), ComIn needs to trigger data transfers before and after each entry point executed on GPUs. The implementation is currently ongoing and will be part of the next release version of ComIn.

- Currently metadata is stored in a pre-defined and thus inflexible data structure. An update, which will not be visible to the users, is a more flexible implementation of metadata information via a hash table/key-value storage[9]. This will simplify the introduction of further metadata to ComIn. This development is planned to start during the second half of 2024. An

[9]A key-value database stores data in a “key-value” format and is optimised for reading and writing that data

additional use case of this key-value storage would be to associate CMOR[10] names with each variable to support users taking part in model inter-comparison studies.

- When a plugin requests an ICON data field it does this for a specific list of entry points. This information can be used by ICON to determine the exchange of data throughout the simulation. With the additional information of the desired access, i.e. if data is just read, adapted by a plugin or also required for halo synchronization, the host model could detect incorrect access patterns and return an error or a warning if access restrictions are violated. The latter part is currently not implemented but a potential expansion for the future, probably available only in a debug mode of ComIn as this option would add overhead. Additionally, this information can be used to support asynchronous execution of plugins and to enable efficient halo synchronization through ComIn.

- To ensure a flexible use of ComIn with several plugins, their call sequence at different entry points should be adaptable. This is currently not possible, but control via an additional ComIn namelist is planned, in which execution priorities of plugins at each entry point can be listed.

- Currently all plugins are executed in sequence when an entry point is reached. The callback routines are called in a blocking manner, so the execution of ICON is paused until the callback returns. Considering that several plugins are independent of each other, or of the subsequent procedures in ICON, an option for asynchronous execution could increase the run-time performance especially for computationally expensive plugins. At the same time, this requires careful use to prevent deadlocks or erroneous results.

Some additional extensions will be motivated by plugins, but might require changes in the overall concept of ComIn. Suggestions will be gathered and evaluated in terms of their feasibility among the ComIn (and if necessary ICON) developers, before potentially being implemented. Any changes to ICON itself, e.g. access to currently local fields, required by ComIn users, will not be supported by the ComIn developers, but must be discussed with the ICON core developers instead.

7 Conclusions and outlook

In this article ComIn was introduced as a lightweight, interoperable, and flexible model-specific interface library for ICON, which minimises the required changes to the ICON code. A variety of use cases were introduced in Section 5, which are considered to be beneficial to the NWP and climate community using ICON. Going along with an increasing adoption of ComIn, some additional steps in assessing performance and usability of ComIn are necessary:

- The evaluation of the interface itself in terms of minimal computation and memory overhead to ICON is presented in this article. Preparations for a large-scale test case for a setup with a plugin, namely the integrated framework MESSy, are currently ongoing. As each plugin is differently interacting with ICON, such an evaluation is recommended with each new plugin connected to ICON. In this way the optimal setup of parallel resources can be determined.

[10]https://cmor.llnl.gov/

– As ComIn is adopted by the community, the user experience needs to be reviewed, e.g. if the complexity and usability from the user perspective are balanced. Here, for example, the ease of getting started with ComIn, the experience during plugin development, and the support for testing are key factors. In addition, a public online tutorial might be beneficial for new users.

– Based on the results of the large-scale test case and feedback from the first users on their experience of adopting ComIn, but also on currently missing features, a critical review of the benefits and issues of ComIn needs to be performed at regular intervals: What are current shortcomings and how can they be resolved?, What future applications and thus required developments can be envisioned?, How can developments be kept sustainable?, and How can the implementation of ComIn be advanced, for example through the use of key-value storage for metadata?

– The re-usability of the ComIn concept for other models than ICON can be discussed. The design was developed as general as possible. Other ESM components can adopt ComIn (and become its host model) to increase their modularisation and facilitate expansion from outside their core community. However, as the descriptive data and entry points are of course specific to the host model, some modifications to the interface would be required for such a transfer to a separate, ComIn-like implementation. If the benefits of the ComIn approach are expected to outweigh the additional effort of a ComIn-like implementation for other host models, then this article along with the white paper (ComIn authors, 2024) can act as a guideline for the adaptation process.

Appendix: Glossary

descriptive data Descriptive data are ComIn data structures that provide metadata on ICON and the simulation, for example on domain specific settings, the parallelisation and experiment start/stop dates. They are typically exposed (i.e. shared) as pointers but should be used read-only. 8, 10, 12–15, 20

entry point Entry points in ICON allow to call plugin routines directly from ICON, if they are registered via the ComIn callback library. 8

host model ICON is the host model of ComIn. That means that ComIn is based on ICON's data structures and that entry points are implemented within the ICON control flow. 1

plugin ICON-external software which is attached to ICON via ComIn is called a (ComIn) plugin. Plugins can range from individual externalised features to full Earth System model components. 1

Code availability. ComIn is released under the BSD-3-clause license[11] and is available from https://gitlab.dkrz.de/icon-comin/comin. A version of the ICON model was released under the Open Source BSD 3-clause license at the end of January 2024 (ICON Partnership, 2024).

[11]https://opensource.org/license/BSD-3-Clause, 26.03.2024

The version of ComIn used in this article is part of the ICON version 2024.01-01 (tag tags/icon-2024.01-01 in the ICON repository https://gitlab.dkrz.de/icon/icon and branch release-2024.01-public in the public ICON repository https://gitlab.dkrz.de/icon/icon-model), which is available with the DOI https://doi.org/doi:10.35089/WDCC/IconRelease01 at https://www.wdc-climate.de (ICON Partnership, 2024)).

Author contributions. KH wrote a first draft of the manuscript with contributions and feedback from BK, AK and PJ. NAD, JG, MH, WL, FP and DR provided comments and additional input on the first draft, based on which KH prepared the final version. FP and DR provided the input data and setup instructions for the test case. The ComIn project was managed by FP and all other authors conributed to the code development.

Competing interests. AK is an executive editor of GMD and PJ acts a topic editor for GMD.

Disclaimer. TEXT

Acknowledgements. We want to thank all colleagues involved in working groups, discussions and proposals on coupling ICON to other models or model components and elaborating requirements of an open community programming interface to the model system. Especially Heidrun Matthes, Mariano Mertens and all contributers to the "GI3" (Generalised Interface in ICON) proposal. We thank Ali Hoshyaripour, Sven Werchner, Ralf Müller and Florian Ziemen for their input and advice during the formulation and development of ComIn and Daniel Reinert for his involvement in the initial planning and first development. We thank Axel Lauer for the very constructive feedback on a draft version of the article.

The model simulations have been performed at the German Climate Computing Centre (DKRZ) through support from the Bundesministerium für Bildung und Forschung (BMBF). DKRZ and its scientific steering committee are gratefully acknowledged for providing the HPC and data archiving resources (including */pool/*) for the project ESCiMo (Earth System Chemistry integrated Modelling). One project on which this article is based was funded by the German Federal Ministry of Education and Research under the funding code 01LK2203. The responsibility for the content of this publication lies with the authors. The authors acknowledge the support by the National Earth System Modelling Project (natESM), which is funded through the Federal Ministry of Education and Research (BMBF) under grant agreement No. 01LK2107A, and by the DLR project GHGmon.

References

Behrens, J., Hanke, M., and Jahns, T.: Yet Another eXchange Tool, https://swprojects.dkrz.de/redmine/projects/yaxt, last access: 6 May 2024, 2024.

Bonanni, A., J. H. and Quintino, T.: Plume: A plugin mechanism for numerical weather prediction models, https://doi.org/doi:https://doi.org/10.5194/egusphere-egu23-7944, 2023.

ComIn authors: ComIn documentation, https://gitlab.dkrz.de/icon/icon-model/-/blob/release-2024.01-public/externals/comin/doc/icon_comin_doc.md, last access: 17 June 2024, 2024.

Craig, A., Valcke, S., and Coquart, L.: Development and performance of a new version of the OASIS coupler, OASIS3-MCT_3.0, Geoscientific Model Development, 10, 3297–3308, https://doi.org/10.5194/gmd-10-3297-2017, 2017.

Earth System Modeling Framework: Earth System Modeling Framwork, ESMF, https://earthsystemmodeling.org/, last access: 5 May 2024, 2024.

European Centre for Medium-Range Weather Forecasts: Plugin mechanism, plume, https://github.com/ecmwf/plume, last access: 5 May 2024, 2024.

Geophysical Fluid Dynamics Laboratory: Flexible Modelling System, FMS, https://www.gfdl.noaa.gov/fms/, last access: 5 May 2024, 2024.

Hagemann, S., Stacke, T., and Ho-Hagemann, H. T. M.: High Resolution Discharge Simulations Over Europe and the Baltic Sea Catchment, Frontiers in Earth Science, 8, https://doi.org/10.3389/feart.2020.00012, 2020.

Hanke, M., Redler, R., Holfeld, T., and Yastremsky, M.: YAC 1.2.0: new aspects for coupling software in Earth system modelling, Geosci. Model Dev., 9, 2755—-2769, https://doi.org/10.5194/gmd-9-2755-2016, 2016.

Heinzeller, D., Bernardet, L., Firl, G., Zhang, M., Sun, X., and Ek, M.: The Common Community Physics Package (CCPP) Framework v6, Geoscientific Model Development, 16, 2235–2259, https://doi.org/10.5194/gmd-16-2235-2023, 2023.

ICON Partnership: ICON model website, https://www.icon-model.org, last access: 5 May 2024, 2024.

Jöckel, P., Sander, R., Kerkweg, A., Tost, H., and Lelieveld, J.: Technical Note: The Modular Earth Submodel System (MESSy) - a new approach towards Earth System Modeling, Atmospheric Chemistry and Physics, 5, 433–444, https://doi.org/10.5194/acp-5-433-2005, 2005.

Jöckel, P., Kerkweg, A., Pozzer, A., Sander, R., Tost, H., Riede, H., Baumgaertner, A., Gromov, S., and Kern, B.: Development cycle 2 of the Modular Earth Submodel System (MESSy2), Geoscientific Model Development, 3, 717–752, https://doi.org/10.5194/gmd-3-717-2010, 2010.

Joppich, W. and Kürschner, M.: MpCCI—a tool for the simulation of coupled applications, Concurrency and Computation: Practice and Experience, 18, 183–192, https://doi.org/10.1002/cpe.913, 2006.

Kerkweg, A., Hofmann, C., Jöckel, P., Mertens, M., and Pante, G.: The on-line coupled atmospheric chemistry model system MECO(n) – Part 5: Expanding the Multi-Model-Driver (MMD v2.0) for 2-way data exchange including data interpolation via GRID (v1.0), Geoscientific Model Development, 11, 1059–1076, https://doi.org/10.5194/gmd-11-1059-2018, 2018.

MESSy Consortium: Modular Earth Submodel System, MESSy, https://www.messy-interface.org, last access: 5 May 2024, 2024.

National Center for Atmospheric Research: Common Community Physics Package (CCPP)-framework, https://ccpp-techdoc.readthedocs.io/en/latest/Overview.html, last access: 5 May 2024, 2024a.

National Center for Atmospheric Research: CCSM Coupler Version 7, https://www.cesm.ucar.edu/models/cpl/7.0, last access: 5 May 2024, 2024b.

National Center for Atmospheric Research: Community Earth System Model, CESM, http://www.cesm.ucar.edu/models/ccsm4.0/, last access: 5 May 2024, 2024c.

Prill, F., Reinert, D., Rieger, D., and Zängl, G.: ICON Tutorial - Working with the ICON model, Deutscher Wetterdienst, Offenbach, https://doi.org/10.5676/DWD_pub/nwv/icon_tutorial2024, 2024.

Redler, R., Valcke, S., and Ritzdorf, H.: OASIS4 – a coupling software for next generation earth system modelling, Geoscientific Model Development, 3, 87–104, https://doi.org/10.5194/gmd-3-87-2010, 2010.

Reick, C. H., Gayler, V., Goll, D., Hagemann, S., Heidkamp, M., and Nabel, J. E. M. S.: JSBACH 3 - The land component of the MPI Earth System Model: documentation of version 3.2., Berichte zur Erdsystemforschung, Max Planck Institute for Meteorology, https://doi.org/doi:10.17617/2.3279802, 2021.

Valcke, S.: The OASIS3 coupler: a European climate modelling community software, Geoscientific Model Development, 6, 373–388, https://doi.org/10.5194/gmd-6-373-2013, 2013.

Valcke, S., Balaji, V., Craig, A., DeLuca, C., Dunlap, R., Ford, R. W., Jacob, R., Larson, J., O'Kuinghttons, R., Riley, G. D., and Vertenstein, M.: Coupling technologies for Earth System Modelling, Geoscientific Model Development, 5, 1589–1596, https://doi.org/10.5194/gmd-5-1589-2012, 2012.

Zängl, G., Reinert, D., Rípodas, P., and Baldauf, M.: The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core, Q. J. Roy. Meteor. Soc., 141, 563—-579, https://doi.org/10.1002/qj.2378, 2015.