Reviewer 1 (RC1)

The paper by Hartung et al. describes a new library that is developed to couple external software, ranging from Python diagnostics to ESM components, to the ICON model. The article provides an extensive description of the ComIn library's capabilities, underlying motivations and an overview of potential applications that can benefit the ICON community. I think the paper is well structured, providing necessary details to describe the ComIn library and the reasoning behind the software design. I recommend this article for a publication in GMD, however some minor comments should be addressed.

Thank you for this positive review of our manuscript. We respond to the specific minor comments below and highlight additions to the manuscript in bold font.

Specific comments:

- L187-193: Could you specify in which cases plugins are executed asynchronously? Is this possible for couplers and I/O servers when used as plugins through ComIn v0.1.0 or do these tasks also rely on blocking callbacks such as for synchronous plugins?
  - At the moment no plugins have been tested asynchronously. We mainly wanted to make the point here, that this is technically possible. On the ICON side all callbacks are executed as blocking functions, so any asynchronicity needs to be handled by the plugin. Indeed, couplers and I/O servers present potential applications that could run asynchronously to ICON. However, this functionality is outside the scope of the presentation of the design principles of ComIn within this manuscript.
- The authors cautiously point out that plugin implementation and the use of additional resources to run plugins are the user's responsibility. Indeed, synchronous execution of a plugin can pause the ICON model for quite some time for a computationally inefficient plugin, which is indicated in the manuscript. On the contrary, the article does not really show evidence of efficient execution of ICON/ComIn for an "efficient plugin" (e.g. a simple monitoring Python script to generate a map or compute a mean field value). A test with ComIn disabled in ICON-NWP is described and shows that the ICON/ComIn implementation introduces very little overhead for ICON, which is an important result to stress. But the current test runs do not show

the impact of activating ComIn for ICON. An additional testcase would be desirable to show how much of a typical ICON timestep is spent on running a fast plugin.

- The main test simulation has ComIn enabled but no plugin attached, i.e. all entry points are called from the ICON-side but are not connected to an execution of a plugin routine. In one way this setup thus corresponds to an extremely efficient plugin (i.e. one with no time and memory overhead). To clarify this, we extend the description in section 4 (from line 397 in the old version) as
- "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 the 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."
- In this paper on the design of ComIn we think that this basic test of the performance is sufficient. Further simulations with simple and complex plugins are planned for the future.

- L255: Compatibility between the ICON-ComIn versions is verified by ComIn but what about the ComIn-plugin(s) compatibility? Does the software development process guarantee API backward compatibility so that plugins can be used with newer versions of ICON/ComIn? If not, do users need to develop several versions of a plugin depending on the ICON version in use?

  - The approach of the ComIn Application Programming Interface (API) is to guarantee backward compatibility for minor ComIn version updates. A major release version indicates that the interface was changed in such a way that backward compatibility can not be guaranteed. We realise that this information was

missing from the manuscript. We will add a few sentences on this, also regarding the dependency on the ICON version, in Section 3.2:

- "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."

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

- Sect 3.4.2: It can be complex for a Python script developer to work with ICON data produced on an unstructured grid in a parallel setup. Providing data post-processing functionalities (or at least guidelines on how to proceed) would improve the user experience and facilitate ComIn adoption. Does ComIn provide functionalities (through its API or as plugin examples) to reconstruct fields globally and to regrid fields on the target grids of plugins? Could you clarify how plugin developers should work with ICON data provided by ComIn? Sect 5 provides a valuable list of possibilities offered by the library but lacks details on how users should proceed.

  - At the moment there is a simple plugin that uses YAXT (also mentioned in Section 3.9), which processes the global mean temperature from all participating MPI processes. Users can start off from this example for their applications. For regridding of the target grids, ComIn in combination with YAC might be interesting.

  - Similar to the question about a lightweight example plugin, we think that instructions on this question are not in line with the intention of this manuscript, namely to give an insight into the design process of ComIn and not to showcase

example plugins. However, we take this questions as feedback for the documentation, were some hints can be provided to plugin developers.

- L438-441: Prototyping with Python to develop parametrizations would first require a translation of the Fortran code into Python. This may not be straightforward, considering that the two programming languages handle efficient array computations very differently. It is unclear why switching languages would speed-up the model development process, not to mention the optimization process which can only be carried out in the model's native language. Nevertheless, I wonder to what extend parametrization development could rather benefit from ComIn when carried out in Fortran and outside the model, given that the library provide access to input/output parametrization fields and separate program build.

  - The point "prototyping with Python" was not referring to a possible extension or update of an existing parameterisation written in Fortran but rather to a completely new development. When developing a new parameterisation, some physically-motivated experiments need to be done (e.g. deciding on the correct range of parameters) before optimising the setup numerically. The first experiments can be done more efficiently in Python and speed up the development process overall. With ComIn, the parameterisation can then be either used as a Python plugin or translated to Fortran, if desired. It is of course also true that with ComIn Fortran parameterisations can be developed outside of ICON.
  - We will rephrase the point in the manuscript, as follows: **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.**

– L459-462: Opening up the possibility of using or developing Python packages (AI, ML, . . .) to emulate parametrized processes should be of interest to the ICON community. I think this point should be put forward instead of L438-441.

  – Thank you for your comment. Apart from the first points, which are already implemented in test setups, there is no reason for the ordering of these list of possible applications. We will add a corresponding point in line 424 ("and also introducing some which are possible with ComIn **(in no particular order)**") to emphasise this.

– L434-437: The purpose of the evaluation and performance analysis referred to here is unclear. Do you employ the mentioned tools (a sentence to describe these tools is missing) to monitor the use of computing resources for the simulation job (computing time, memory usage, . . .)? Or do you intend to compute relevant metrics to analyze physical phenomena of the simulation (perhaps in the form of timeseries)? Could you also clarify the statement "a well set-up simulation does not need to be restarted after evaluation"?

  – Thank you for the hint on the confusing phrasing. We reformulate the sentence "a well set-up simulation does not need to be restarted after the simulation" to "... **a simulation with satisfactory performance can be continued and does not need to be restarted after the evaluation.**"

  – To clarify the first point we update the text as follows:

  – "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 [. . .]. 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 [. . .]"

  – As the performance analysis is not the main focus of the work presented in the manuscript, we will not add further details on the described tools. More information can be obtained from their respective websites.

– L160-167: You could add a statement emphasizing that the interface library approach depends mainly on the model data structure and the model execution flow.

– Thank you for this idea to summarise this section. We add at the end of this paragraph:

– "**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.**"

– L185: Could you explain on how the dynamical loading works in ComIn? Does it require usage of specific dependency management tools and other libraries that were not previously part of ICON-NWP?

– Dynamic loading is a functionality provided by the operating system. It can be used independently of ICON. For dynamic loading, libraries are compiled not as so-called static libraries and linked into one executable, but as shared libraries, which are only loaded at run-time by the main executable.

– We think that an introduction to the concept of dynamic loading is outside of the scope of this manuscript.

– L225: The goal of limiting software maintenance effort (ICON, ComIn) is a valuable one and this is mentioned several times. Could you provide an estimate of the number of lines added to ICON in response to the inclusion of ComIn v0.1.0? Or perhaps provide the fraction of lines of code that were modified/added for ComIn into ICON relative to the ICON codebase?

– We are not sure this is relevant for the design document but we can still provide the number of added lines. In the version used in the manuscript, ICON, more specifically the "src" directory, consists of about 800 Fortran90 files which contain a bit more than half a million lines of code. For the introduction of ComIn twelve files were touched and less than 1000 lines of code added (around 870).

– We add in line 225 of the original version of the manuscript: "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)**."

Minor comments:

- L16: host model model -> host model?
- L66: from each of these process -> processes?
- L478: you may want to add a definition or link for CMOR
- L507: adapted -> adopted?

We have adapted these minor comments in the manuscript.