Ronald Lehnigk*, Martin Bruschewski, Tobias Huste, Dirk Lucas, Markus Rehm and Fabian Schlegel

Sustainable development of simulation setups and addons for OpenFOAM for nuclear reactor safety research

https://doi.org/10.1515/kern-2022-0107
Received November 18, 2022; published online February 16, 2023

Abstract: Open-source environments such as the Computational Fluid Dynamics software OpenFOAM are very appealing for research groups since they allow for an efficient prototyping of new models or concepts. However, for downstream developments to be sustainable, i.e. reproducible and reusable in the long term, a significant amount of maintenance work must be accounted for. To allow for growth and extensibility, the maintenance work should be underpinned by a high degree of automation for repetitive tasks such as build tests, code deployment and validation runs, in order to keep the focus on scientific work. Here, an information technology environment is presented that aids the centralized maintenance of addon code and setup files with relation to reactor coolant system safety research. It fosters collaborative developments and review processes. State-of-the-art tools for managing software developments are adapted to meet the requirements of OpenFOAM. A flexible approach for upgrading the underlying installation is proposed, based on snapshots of the OpenFOAM development line rather than yearly version releases, to make new functionality available when needed by associated research projects. The process of upgrading within so-called sprint cycles is accompanied by several checks to ensure compatibility of downstream code and simulation setups. Furthermore, the foundation for building a validation data base from contributed simulation setups is laid, creating a basis for continuous quality assurance.

Keywords: computional fluid dynamics; OpenFOAM; reactor cooling system; verification and validation.

1 Introduction

Open-source tools are becoming increasingly popular in the field of nuclear reactor analysis and safety research. Simulation software exists for several aspects, both specialized and general purpose. This includes neutronics codes like OpenMC (Romano et al. 2015) or OpenMOC (Boyd et al. 2014), codes for thermal-hydraulic analysis like NekRS (Fischer et al. 2021) or TrioCFD (Angeli et al. 2015), addons to existing Computational Fluid Dynamics (CFD) codes like containmentFOAM (Kelm et al. 2021) or OFFBEAT (Scolaro et al. 2020) as well as multi-physics applications or libraries like GeN-Foam (Fiorina et al. 2015) or MOOSE (Lindsay et al. 2022). Evidence for the growing recognition of these developments is the ONCORE initiative (Open-source Nuclear Codes for Reactor Analysis) facilitated by the International Atomic Energy Agency to support both research and development as well as education and training (Fiorina et al. 2021).

Developing and using open-source software is beneficial in several ways. Most apparent is the independence from the licensing policy of a software manufacturer, making it a very good basis for collaboration. Of particular interest for research groups is the possibility to study an existing implementation and the freedom to add code for prototyping new concepts or models. This freedom often alleviates the need to start writing entirely new software and is frequently used to either fork a project or to create an addon for it. Depending on the project, contributions to the original software repository may also be welcome. While forking is simple at first, it implies taking responsibility for release, quality, software, knowledge and communication management. Essentially, all the work of the principal developers is repeated without their fundamental knowledge about the design of the code. Contributing to the main code base on the other hand requires compliance with the guidelines and design concepts set forth by the core developers. Further,

The content of this article was initially presented at the 33rd German CFD Network of Competence Meeting, held on March 22–23 2022 at GRS in Garching, Germany.

***Corresponding author: Ronald Lehnigk**, Helmholtz-Zentrum Dresden – Rossendorf e.V., Bautzner Landstraße 400, 01328 Dresden, Germany, E-mail: r.lehnigk@hzdr.de. https://orcid.org/0000-0002-5408-7370
Martin Bruschewski, University of Rostock, Chair of Fluid Mechanics, Albert-Einstein-Straße 2, 18059 Rostock, Germany
Tobias Huste, Dirk Lucas and Fabian Schlegel, Helmholtz-Zentrum Dresden – Rossendorf e.V., Bautzner Landstraße 400, 01328 Dresden, Germany
Markus Rehm, Framatome GmbH, Paul-Gossen-Straße 100, 91052 Erlangen, Germany

they require the resources, both financial and human, to check contributions and maintain them in the long run. Depending on the domain, these resources may be scarce. Contrary to forking and contributing, creating and distributing an addon and engaging with the core developers to supply the interfaces for it can be an efficient compromise. Still, downstream developers have the responsibility to control the quality and reliability of their additions and, most importantly, need to ensure that they remain compatible with the main code base as it is developed further. An analogue situation exists for users who do not develop the code, but the setup and configuration files that need to be created to model a certain application. An obvious way out is to stick to a certain version release of the main code, but this is clearly unsustainable since new features cannot be used. If other scientists intend to build on former developments they inevitably repeat the implementation work. This is particularly problematic for qualification work like PhD projects and publicly funded projects that build on each other. The clear need to ensure long-term reproducible and reusable developments is increasingly coming into focus, particularly for publicly funded research projects (German Science and Humanities Council 2020). For actively developed codes this is a continuous task and the resources for it should ideally be bundled among research groups of a certain field.

A widely used and actively maintained open-source project to which the situation illustrated above applies is the Computational Fluid Dynamics code OpenFOAM released by the OpenFOAM Foundation. Its core developers put a strong focus on ensuring a high degree of maintainability. Reorganizing interfaces and generalizing existing functionality is given priority over forcing a possibly compromising implementation of new functionality that features a too application-specific design. Ultimately this facilitates downstream developments of derived functionality, but existing addon code and setup files have to be adapted quite frequently. While care is taken by the core developers to support backward compatibility for a certain time period in order not to disturb simulation workflows, setups should be updated in a timely manner to avoid a backlog of maintenance work.

In order to improve the situation for all addon code and setup developments in Germany concerned with the reactor coolant system, the Helmholtz-Zentrum Dresden – Rossendorf e.V. (HZDR) has created an information technology (IT) environment that supports a centralized and continuous maintenance. In the project, tools that are widely used in software engineering are adapted to facilitate the collaborative creation and maintenance of developments that cannot be integrated into the official release of OpenFOAM or are not intended to, respectively.

The paper gives an overview of the main features and ideas. It is structured as follows: Section 2 presents the relevant tools and main components of the project. Section 3 illustrates how developments are conducted, i.e. the process of a contributing partner integrating code or simulation setup files. Section 4 explains how maintenance work is carried out, i.e. the upgrade of the underlying OpenFOAM installation. Finally, Section 5 presents the approach to achieve the long-term goal of creating a validation database.

2 Components of the IT environment

The main components of the IT environment on the basis of which collaborative developments are done are presented in Figure 1.

The core of the project are two separate Git repositories, wherein files are placed under version control, which enables distributed and non-linear development by means of branching and merging. The repositories are made available through the community edition of the web-based open-source software development platform GitLab. The corresponding GitLab instance is self-hosted by the Helmholtz Federated IT Services[1] (HIFIS) and directly accessible to all members of the Helmholtz Association of German Research Centers and collaboration partners with institutional logins. On the basis of Git, GitLab supplies a wealth of tools for managing collaborative editing of files, e.g. simulation code and setups, including an issue tracking system, merge requests where changes or additions to a repository can be reviewed as well as so-called Auto DevOps tools to automatically build, test and deploy developments.

In the first repository, code that cannot be contributed to the OpenFOAM main development line, e.g. due to export control limitations or a prototype status, is to be integrated. The code within the repository forms an addon to a regular OpenFOAM installation. Apart from code, the first repository also contains fast running test setups for verifying certain pieces of functionality, also referred to as integration tests, as well as tutorials to demonstrate the corresponding setup in an application-related context. The second repository contains all simulation setups created by partners.

The main goal of the project is to keep the content of both repositories compatible with the recent state of OpenFOAM. Since the OpenFOAM development line OpenFOAM-dev is public domain, this state doesn't have to match a version release. However, following OpenFOAM-dev on a per-commit

1 https://hifis.net.

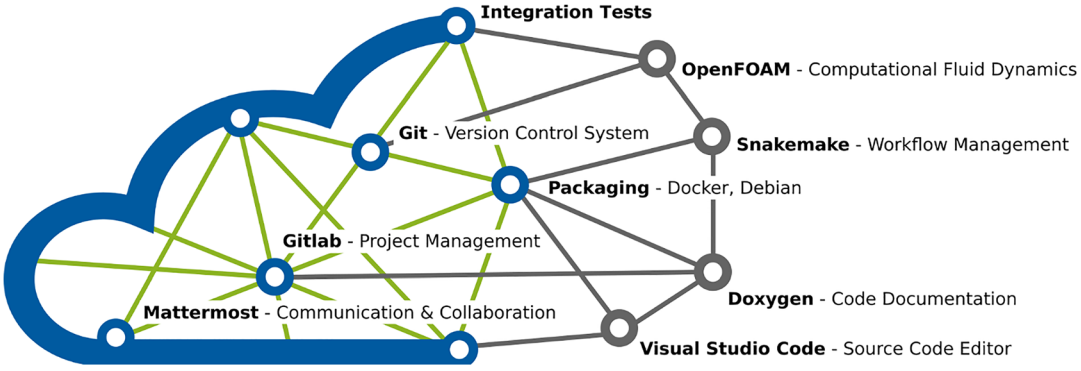

Figure 1: Illustration of the main components of the IT environment using the logo of the Helmholtz Federated IT Services (HIFIS). Blue nodes indicate services provided by HIFIS which are configured to meet the needs of the project. Nodes in grey depict additional services added to the IT environment by its maintainers.

basis is quite challenging for downstream developers. Rather, a snapshot of OpenFOAM-dev is used, which allows for flexible upgrade intervals, such that new functionality that is not yet part of an OpenFOAM version release can be made available in the IT environment when required. Upgrades are done in the scope of so-called sprint cycles, whereby a range of OpenFOAM-dev commits, i.e. sets of changes, is played on top of the OpenFOAM version supplied with the project. The version number for a sprint cycle is defined as <release-version>-s.<sprint-cycle>, e.g. 8-s.5 or 9-s.1. The first digit represents the OpenFOAM stable release that precedes the selected commit. The letter *s* is short for sprint cycle and the last digit represents the current sprint cycle with respect to the preceding release. To facilitate the installation of a sprint cycle version, a well-structured guide is provided describing various options, including compilation from sources, using Debian packages as well as pulling or building Docker images that provide OpenFOAM together with the addon in containerized form. The packages and images are made available through the GitLab registry. Creating containers from Docker images is a particularly attractive installation option for developers, since this creates an isolated software environment with identical properties independent of the underlying operating system, guaranteeing full reproducibility of software behavior.

An important aspect of the project is the targeted use of Continuous Integration and Deployment (CI/CD) pipelines, which are set up to automate repetitive tasks. They are executed on dedicated nodes supplied by HIFIS and registered as so-called runners within the GitLab project. The software environment on these runners is established using Docker images, which, depending on the task, contain the OpenFOAM reference installation. An example is the nightly build pipeline shown in Figure 2.

It generates and deploys an up-to-date source code documentation for the combination of OpenFOAM and addon. Also it updates images, packages and a high-performance computer (HPC) cluster installation. Another pipeline exists to update the sprint cycle version, i.e. to rebuild the OpenFOAM reference installation. Particularly important for a sustainable development of code and setups are merge request pipelines wherein changes and additions to either repositories are checked before they become a part of the main content. Merge request pipelines are presented in more detail in Sections 3 and 4.

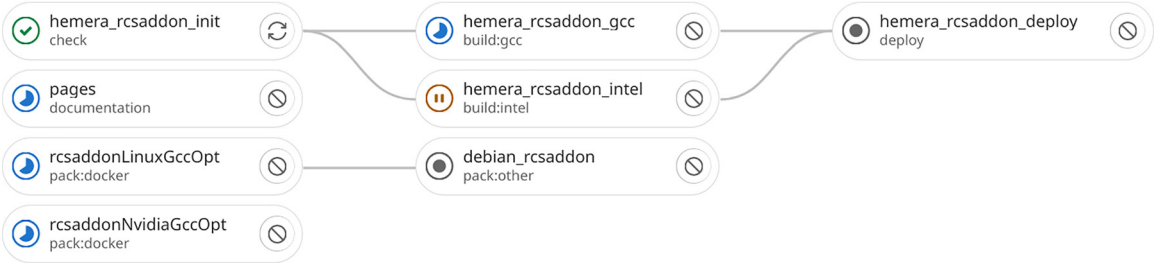

Figure 2: Jobs and dependencies within the nightly pipeline, which forms a part of the GitLab project containing the addon code. It includes updates of a high-performance computer cluster installation (GCC and Intel compilers), Docker images (for on-board and Nvidia graphics) and Debian packages as well the documentation.

For editing code and setup files, the use of the Visual Studio Code (VS Code) integrated development environment is recommended. If properly configured, useful features such as code completion, syntax highlighting and problem matching will also work properly for OpenFOAM. Many useful extensions for VS Code exist, including a particularly useful one that establishes a direct connection to GitLab and allows for in-editor processing of merge requests. To support novel users in the configuration process, template files for setting up VS Code projects are provided.

The GitLab web platform is particularly efficient for discussing code and setup developments with maintainers and team members. Apart from that, the instant messaging service Mattermost, which runs on HIFIS servers as well, can be used. It allows for private and team discussions as well as general announcements in dedicated channels. Discussion threads remain accessible and searchable for future reference.

A final important component to be mentioned here is Snakemake (Mölder et al. 2021), a workflow management system that is used for a scalable execution of all simulations setups placed in the second repository. It enables the execution of validation pipelines and can create self-contained reports that include all plots generated during post-processing. Details are given in Section 5.

3 Development phase: creation of simulation setups and peer review

Starting point for adding a simulation setup to the case repository is the creation of an issue in GitLab. The term issue is general and covers both bugs, feature or maintenance requests as well as the addition of new cases. Dedicated issue description templates exist for each scenario. The template for adding a new case requests the developer to supply a short description of the simulation setup together with documentation such as articles, reports or websites. Further, interested users or possible reviewers can be informed about the planned work, simply by mentioning them with @<*username*> in the description or a comment, which triggers a notification email. Issues are the first place to discuss a suggested setup before any actual work is done, e.g. to determine a suitable OpenFOAM solver. Issues can be assigned to project milestones wherein larger work packages are organized. Also, if applicable, an issue can be marked with one or more predefined labels allowing for better categorization.

In the next step, a merge request is created from the issue whereby GitLab automatically creates a branch from the repository's main content. Within merge requests, developers present their changes or additions and invite a maintainer or other team member to review them. At least two people are involved in the process, an assignee and a reviewer. Assignees are responsible for preparing the merge request and are usually also assigned to the corresponding issue. In the same way as for the issues, merge requests feature a description for which a default template is given. The template contains a check list which is to be followed by the assignee and the reviewer. Among other things, the check list requires confirmation by the assignee that the contribution guidelines available for the project were read and understood. Further, it demands that case setups are accompanied by a meaningful Readme file, for which there is a template in the repository as well. The actual work of the assignees starts by adding content to the branch generated by GitLab, either through the web interface or within a local clone of the repository. Any set of changes can be added to the Git version control in the form of a commit. It is advisable, also in a development branch, that the changes associated with a commit serve a single purpose about which reviewers are informed in the commit log message. This has the advantage that a searchable record of the work is created, opening the possibility to revert certain change sets or to jump back in history entirely. The commits within the branch become visible for the reviewer once the assignee has pushed them to GitLab. By removing the *Draft* prefix of the merge request title, reviewers are signaled that the changes are ready to be checked. Reviewers can either look at the change set associated to a certain commit or view all differences to the main branch of the repository at once. The check list reminds them to check for compliance with the OpenFOAM code style and understandable documentation. They can leave general comments or open discussion threads containing criticism or suggestions of better solutions. Both, comments and discussion threads, can be associated with one or more lines within files that were added or changed by the assignee. GitLab demands that threads are resolved before merging is allowed. Optionally, follow-up issues can be created from threads to be addressed in a later merge request.

Every push event to a merge request also triggers a merge request pipeline as mentioned in Section 2. The individual stages and jobs of this pipeline in its current form are shown in Figure 3.

The pipeline is split according to the usual steps within a CFD simulation. All case setups that are a part of the branch are scheduled to run, but they are configured to run only for a single time step. Hence, the merge request pipeline

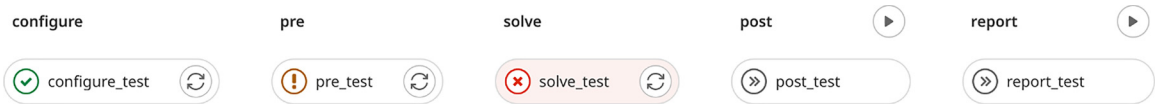

Figure 3: Example merge request pipeline for the simulation setup repository. The pipeline is split into stages, indicated by the bold text at the top, which are executed in order and can contain one or more jobs. GitLab highlights successful jobs (configure_test), jobs that fail but are allowed to (pre_test), failed jobs (solve_test) and jobs that either have not started yet or are not allowed to due to a failure of a previous job or stage (post_test, report_test).

constitutes a check of the setup file input syntax, ensuring compatibility with the combined sprint cycle OpenFOAM version and the addon code. If any of these jobs fail, GitLab will discourage merging.

Once the pipeline succeeds and reviewers are satisfied with the changes, they approve the merge request. Finally, it is the task of a maintainer, which can be either the assignee or reviewer or a third person, to trigger the merging of the changes. It is their task to squash the changes within the development branch into a single descriptive commit, in order to have a clean and well-structured version history in the main branch.

4 Maintenance phase: keeping OpenFOAM setups compatible with OpenFOAM

Ensuring that the addon code compiles for a more recent state of OpenFOAM and that setups files remain compatible is a primary task of the maintainers of the IT environment. Usually, changes are required to both repositories. If needed and available, maintainers will seek support of the original authors of a piece of addon code or a simulation setup during the update process.

A sprint cycle upgrade (see Section 1) is initiated with a dedicated issue and merge request in the addon code repository. First, the OpenFOAM-dev commit, to which the reference installation should be lifted, is defined. A convenience script can be used to update all relevant instructions and configuration files with respect to the corresponding commit sha (identifier) and the sprint cycle version. Next, a GitLab pipeline that generates Docker images and Debian packages containing the new reference installation is triggered through the web platform. With the changes to OpenFOAM, i.e. the upstream repository, the addon frequently doesn't compile anymore.

Following these initial preparations, the process of manually aligning the addon starts and a series of changes is pushed to the associated branch in GitLab. This triggers a merge request pipeline for the branch containing the addon code as shown in Figure 4.

The pipeline does several style checks on code files using a modified OpenFOAM pre-commit hook and linters for Python and Shell scripts. After the initial check stage, compilation with GCC and Clang compilers, which are both supported by the main developers, is tested. Finally, test cases are executed and a comparison of the final result against an archived reference solution is made. At the same time, tutorial setups are tested for input syntax compatibility by running them for a single time step. For the most recent pipeline, a test summary is provided in the overview tab of the corresponding merge request, with links to the individual log files as well as highlighting of failed steps.

Once the pipeline succeeds up to that point, the maintainer can perform checks on the setup repository by adding a specific label like *Requirement: Application test* or *Requirement: Application validation* to the merge request. As a result, the build step is followed by the creation of a Docker

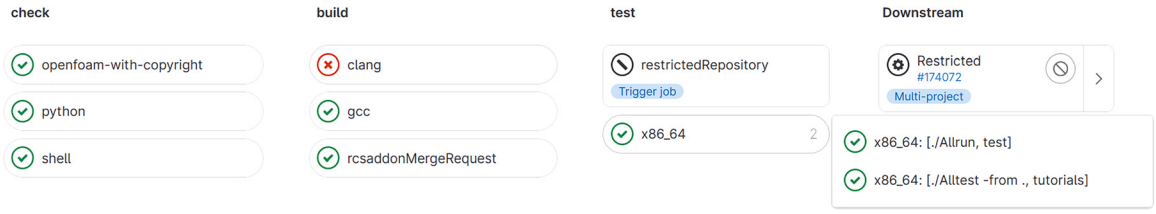

Figure 4: Merge request pipeline within the addon code repository, including style (copyright, python, shell) and build checks (clang, gcc) as well as full runs of test cases and single time step runs of tutorials (×86_64). Furthermore, a Docker image is created from the code in the sprint cycle update branch which is then picked up in a downstream pipeline in the setup repository (Restricted).

image that contains the addon code from the branch in compiled form. This image is then used as basis within a downstream pipeline triggered in the setup repository that runs the same pipeline already presented in Figure 3. The downstream pipeline is either triggered for the main branch of the setup repository, or, if available, for a branch whose name matches the sprint cycle upgrade branch in the code repository. Once the single time step runs succeed for all setups, a full run of all simulation setups on the connected HPC cluster can be triggered manually. At the end, the validity of the individual results is checked by comparison against reference results. Also, a complete HTML-based validation report is created. More details on the validation pipeline are given in the next section.

5 Long term goal: building a validation data base

The pipelines mentioned in the previous sections primarily check the code style, perform build as well as integration tests and further test the input syntax of setups for compatibility with the addon code and the current sprint cycle version. These checks do not reveal whether the simulation setups in the second repository still deliver the same or sufficiently similar results. This requires a full run, followed by an automated comparison against a stored reference solution. The obvious obstacle for such a validation pipeline is the supply of an appropriate amount of computational resources. While a test pipeline is not as demanding and in principle allows for a serial execution of all cases on two or more cores, a validation pipeline will involve the simultaneous execution of parallelized jobs. The available resources differ depending on whether the pipeline is executed locally on a workstation or an HPC cluster. The required resources on the other hand depend on the set of cases that are selected to run within the pipeline and will increase with each case added to the repository. Hence, a core requirement for the validation pipeline is that it must be fully scalable, i.e. simply run in serial for the minimum number of two cores and, for the other extreme, run all cases simultaneously if the number of available cores is equal to or larger than the number of required cores. Between these bounds, the simulation runs should be scheduled such that the overall execution time is minimal for a given amount of resources. Another important requirement is that the post-processed results of all simulations, i.e. plots generated from line samples or probes, are easily accessible in a compact report without the developer being required to browse through potentially nested directory structures. Finally, while triggering the validation pipeline through a web interface is convenient, it should possible to easily reproduce the individual steps on a workstation or HPC cluster outside the GitLab environment.

A popular and well-maintained tool, which allows to create workflows that, among other things, fulfill the aforementioned core requirements of being reproducible and scalable, is the Python-based workflow management system Snakemake. It was originally created to conduct data analysis in the field of bioinformatics, but can be used flexibly to program any workflow. Following the design of the build system Make, the creation of Snakemake workflows is based on the definition of rules that contain commands that are executed to generate a specific output from a particular input file or directory. Snakemake detects the dependencies between these rules and creates a directed acyclic graph that defines the order in which the commands are executed in to create the final outputs. Depending on the available resources, Snakemake tries to process independent branches of the workflow in parallel. In the context of running a set of different OpenFOAM simulations, there are usually no dependencies, i.e. all simulations could be run simultaneously, provided there are enough resources. An exception from this rule is the case of a precursor simulation on a coarser grid, followed by the actual simulation on a finer grid. Other than that, a workflow may be split according to the logical steps of a CFD simulation, i.e. pre-processing, solution and post-processing. Beyond being reproducible and scalable, Snakemake also allows to generate HTML-based reports that contain results, i.e. plots generated during post-processing, as well as auxiliary information as runtime statistics and topology of the workflow. In this work, the validation pipeline is set up as a Snakemake workflow, facilitating an efficient use of available resources and an automatic generation of detailed validation reports. In the following, the principal design of the workflow is presented with selected snippets for a representative setup.

For each step, i.e. configuration, pre-processing, solution, post-processing and validation, a dedicated target rule is formulated within a so-called Snakefile located at the top level of the working directory. Within these target rules, the outputs of all cases for the corresponding step are listed as input. For the example of solving all cases, the target rule is presented in Listing 1.

Listing 1: Target rule for the solution step within the top-level Snakefile.

```
1 rule solve:
2   input:
3     expand('{case_path}/log.solve', case_path=case_paths)
```

To keep the target rules compact, they are generalized using wildcard expansion. In the above example, the *case_path* wildcard takes all values provided in the list of case paths. Further, the log files generated by the individual cases are unified. In the above example, the target rule expects files named *log.solve* as input. These log files simply contain the standard output generated by common case-level OpenFOAM scripts, e.g. *Allrun*, or, analogously, if the simulation runs are split into the aforementioned steps, *Allsolve*. The list of case paths is extracted from a user-defined configuration file as shown in Listing 2.

Listing 2: Content of configuration file workflow.yml.

```
setup_directory: 'setups'
result_directory: 'results'
include:
  'singlePhase':
    'pimpleFoam':
      '2021_Schmidt_et_al': as_is
  'multiphase':
    'compressibleInterFoam':
      '1900_Joukowski': as_is
    'multiphaseEulerFoam':
      '2005_Lucas_et_al': ['47', '129']
```

The *include* dictionary reflects the directory structure within the setup directory. The setups may be designed to either run *as_is*, or, they are templated, i.e. need to be parameterized first. The configuration file also allows to specify other workflow parameters such as the name of the directory to which the results are written.

At the level of each case setup a Snakefile, as shown in Listing 3, must be provided containing the rules that generate the actual output expected by a certain target rule.

Listing 3: Solution step rule within a case-level Snakefile. The strings marked in green are case-specific.

```
use rule solve_case as
    solve_singlePhase_pimpleFoam_2021_Schmidt_et_al
    with:
    threads: 3
      output:
        'results/singlePhase/pimpleFoam/
        2021_Schmidt_et_al/log.solve'
```

To simplify maintenance and reduce code duplication, the rule inheritance feature of Snakemake is used, i.e. a generic rule containing the shell commands for solving a case, as shown in Listing 4, located again in the top-level Snakefile, serves as parent.

Listing 4: Parent rule for solving cases located within the top-level Snakefile.

```
rule solve_case:
    shell:
      '''
        cd `dirname {output}`
        ./Allsolve &> log.solve
      '''
```

Note, that the shell code in the above rule is kept brief to illustrate the general concept. The actual set of commands includes a check whether the number of processor cores (sub domains) specified in the OpenFOAM setup is equal to the number of threads assigned to the rule by Snakemake. Provided that Snakemake is supplied with enough resources, the maximum number of threads that the rule will use is specified in the case-specific rule (see Listing 3, line 2), which is in turn derived from the number of sub domains in the OpenFOAM setup in its original form. However, if the global resources assigned to Snakemake are insufficient, rules that exceed the maximum number of available threads are scaled down automatically. The shell script in the parent rule, Listing 4, must accommodate for that and adjust the number of subdomains accordingly. By virtue of this functionality, the validation pipeline becomes fully scalable as desired.

Another benefit of relying on rule inheritance is that the layout of the case-level Snakefiles is identical and only the green strings and parameters need to be adjusted, see Listing 3. The obvious consequence is that a case-specific Snakefile can be generated automatically within a dedicated configuration step rule, implemented in top-level Snakefile. To this end, a shell script is created, which expects the relevant parameters and creates the case-level Snakefile, unless it is already present. This way, developers adding a simulation setup to the repository only need to follow a set of conventions and can then add their case to the workflow configuration file without further knowledge about the workflow internals, which is fully capsulated.

The graph which is processed by Snakemake for solving all cases considered in Listing 2, and can be visualized by the command `snakemake --dag solve | dot -Tpng > dag_solve.png`, is shown in Figure 5.

The graphs for other targets (configure, pre, post, validate) all look alike and allow to be fully processed in parallel, in contrast to workflows with more complicated dependencies between rules that can be found in the literature (Etournay et al. 2016; Pauli et al. 2018).

With the workflow design presented above, users can execute the validation workflow locally. The sequence of commands is presented in Listing 5.

Listing 5: Sequence of commands for executing the validation pipeline. User-defined parameters are marked by <...>.

```
1 # Configuration (copy setups to result directory and
  # generate Snakefile)
  snakemake --cores <cores> configure
2 # Pre-processing (parametrization, meshing, etc.)
  snakemake --cores <cores> pre
3 # Solution (decomposition, solution, reconstruction)
  ## Local execution
  snakemake --cores <cores> solve
  ## HPC cluster execution (SLURM scheduler)
  snakemake --cluster "sbatch -n {threads}"
  --jobs <jobs> solve
4 # Post-processing (probing, sampling, plotting)
  snakemake --cores <cores> post
5 # Validation (comparison against reference solutions)
  snakemake --cores <cores> validate
6 # Reporting
  snakemake --report report.html post
```

Herein, the user decides how many cores are assigned to a particular step. The jobs within time-consuming steps, in particular the solution, can also be scheduled to run on an HPC cluster as shown in line 3. The number of cores requested for a job is then read from the corresponding rule in the case-level Snakefile. In this case the user supplies the maximum number of jobs that should be scheduled in parallel. An important feature of Snakemake for the purpose of the validation pipeline is that it allows the automatic generation of self-contained HTML reports, wherein plots from the individual cases are gathered together with auxiliary information. By means of a generalized post-processing rule, all PNG files created through plot scripts at the case-level are considered in the report. An example plot is shown in Figure 6.

Besides experimental data and simulation results, the plot also contains reference solutions, which have to be provided by the creator of a setup. The comparison against the simulation results allows to verify, whether the results have changed following either a setup modification, an OpenFOAM version update or a change to the addon code. Prior to the report creation, the workflow includes a validation step (Listing 5, line 5). Therein, a post-processing functionality called *deviation* that forms a part of the addon is executed, which compares OpenFOAM fields against stored reference fields, allowing for a user-specified tolerance. In a similar manner, line samples and probes are compared against reference results using the diff command-line tool. The validation step of the workflow will quit with a non-zero exit code if results differ beyond the prescribed tolerance. Users can then view the report to assess the differences, with the consequence that either code needs to be fixed, the simulation setup requires correction or the reference result must be updated.

6 Summary and outlook

In this work, an IT environment for sustainable downstream developments based on the open-source Computational Fluid Dynamics code OpenFOAM is presented. It is designed to support German research work concerned

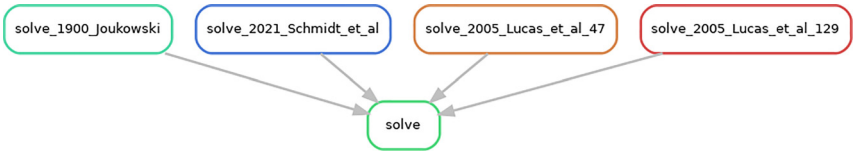

Figure 5: Workflow topology (graph) for the solution step.

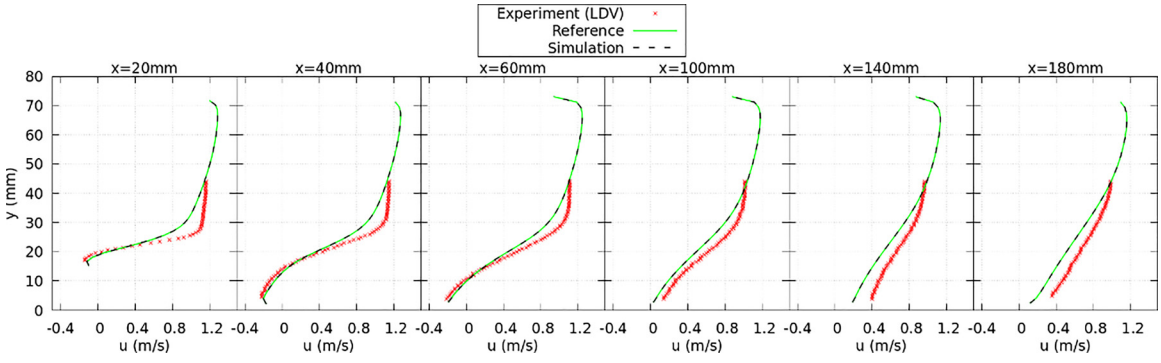

Figure 6: Exemplary plot for case 2021_Schmidt_et_al showing the horizontal velocity component over the domain height at several axial positions (Schmidt et al. 2021). The simulation result graph is dashed to allow for a visual comparison against the stored reference result.

with safety aspects of the reactor coolant system. The environment provides the toolset for an efficient and centralized maintenance of simulation code and setup files, for verification and validation as well as collaboration and communication. These goals are achieved by adapting state-of-the-art software management tools to the requirements of OpenFOAM.

Core of the project is the web-based development platform GitLab, through which two separate Git repositories are supplied to partners. The first contains all code additions as well as fast running test cases and tutorials setups that demonstrate how functionality is used. The second contains the setups representing actual applications. Both repositories are kept compatible with each another, relying on a uniform reference installation of OpenFOAM that is formed from a snapshot of the public development repository of the Foundation release. The reference installation is updated at least for every version release or if new functionality relevant for project partners is released within the development line. The process of integrating new content into either repository or updating the reference installation is underpinned by consequent use of CI/CD pipelines. Based on a containerized installation of OpenFOAM and the addon, style and build checks, verification and input syntax tests as well as validation runs are performed. The setup repository is designed from the ground up to allow for scalable validation runs using the workflow management system Snakemake. With every new setup integrated by partners, it will grow further into an extensive validation data base, creating a basis for continuous quality assurance. Recent validation reports are generated through the reporting feature of Snakemake and distributed through the GitLab platform such that all partners can convince themselves.

While the environment is primarily designed to support nuclear safety research, the same procedures may be applied to other domains as well. Future extensions will be concerned with adding a functionality to the validation workflow that allows to extract validation metrics, e.g. to evaluate the agreement between simulation results, reference solutions and experiments at a higher level, alleviating the need to assess individual plots in the first place. Further, the Snakemake workflow will be designed to rely on a containerized installation of OpenFOAM as well, facilitating its portability to different HPC cluster systems without the need to compile all dependent software from sources.

Author contributions: All the authors have accepted responsibility for the entire content of this submitted manuscript and approved submission.

Research funding: This work was carried out in the frame of research projects funded by the German Federal Ministry for Environment, Nature Conservation, Nuclear Safety and Consumer Protection, project number 1501604.

Conflict of interest statement: The authors declare no conflicts of interest regarding this article.

References

Angeli, P.-E., Bieder, U., and Fauchet, G. (2015). Overview of the TrioCFD code: Main features, V&V procedures and typical applications to nuclear engineering. In: *Proceedings of 16th international topical Meeting on nuclear reactor thermal hydraulics, (NURETH-16)*.

Boyd, W., Shaner, S., Li, L., Forget, B., and Smith, K. (2014). The OpenMOC method of characteristics neutral particle transport code. Ann. Nucl. Energy 68: 43–52, https://doi.org/10.1016/j.anucene.2013.12.012.

Etournay, R., Merkel, M., Popović, M., Brandl, H., Dye, N.A., Aigouy, B., Salbreux, G., Eaton, S., and Jülicher, F. (2016). TissueMiner: A multiscale analysis toolkit to quantify how cellular processes create tissue dynamics. Elife 5: e14334, https://doi.org/10.7554/elife.14334.

Fiorina, C., Clifford, I., Aufiero, M., and Mikityuk, K. (2015). GeN-Foam: a novel OpenFOAM® based multi-physics solver for 2D/3D transient analysis of nuclear reactors. Nucl. Eng. Des. 294: 24–37, https://doi.org/10.1016/j.nucengdes.2015.05.035.

Fiorina, C., Shriwise, P., Dufresne, A., Ragusa, J., Ivanov, K., Valentine, T., Lindley, B., Kelm, S., Shwageraus, E., Monti, S., et al. (2021). An initiative for the development and application of open-source multi-physics simulation in support of R&D and E&T in nuclear science and technology. EPJ Web Conf. 247: 02040, https://doi.org/10.1051/epjconf/202124702040.confproc

Fischer, P., Kerkemeier, S., Min, M., Lan, Y.-H., Phillips, M., Rathnayake, T., Merzari, E., Tomboulides, A., Karakus, A., Chalmers, N., et al. (2021). *NekRS, a GPU-accelerated spectral element Navier-Stokes solver*.

Kelm, S., Kampili, M., Liu, X., George, A., Schumacher, D., Druska, C., Struth, S., Kuhr, A., Ramacher, L., Allelein, H.-J., et al. (2021). The tailored CFD package 'containmentFOAM' for analysis of containment atmosphere mixing, H_2/CO mitigation and aerosol transport. Fluid 6: 100, https://doi.org/10.3390/fluids6030100.

Lindsay, A.D., Gaston, D.R., Permann, C.J., Miller, J.M., Andrs, D., Slaughter, A.E., Kong, F., Hansel, J., Carlsen, R.W., Icenhour, C., et al. (2022). 2.0 – MOOSE: enabling massively parallel multiphysics simulation. SoftwareX 20: 101202, https://doi.org/10.1016/j.softx.2022.101202.

Mölder, F., Jablonski, K.P., Letcher, B., Hall, M.B., Tomkins-Tinch, C.H., Sochat, V., Forster, J., Lee, S., Twardziok, S.O., Kanitz, A., et al. (2021). Sustainable data analysis with Snakemake. F1000Research 10: 33, https://doi.org/10.12688/f1000research.29032.2.

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing polychronization: a guide to maximizing the reproducibility of spiking Network models. Front. Neuroinf. 12, https://doi.org/10.3389/fninf.2018.00046.

Romano, P.K., Horelik, N.E., Herman, B.R., Nelson, A.G., Forget, B., and Smith, K. (2015). OpenMC: a state-of-the-art Monte Carlo code for research and development. Ann. Nucl. Energy 82: 90–97, https://doi.org/10.1016/j.anucene.2014.07.048.

Schmidt, S., John, K., Kim, S.J., Flassbeck, S., Schmitter, S., and Bruschewski, M. (2021). Reynolds stress tensor measurements using magnetic resonance velocimetry: expansion of the dynamic measurement range and analysis of systematic measurement errors. Exp. Fluid 62: 121, https://doi.org/10.1007/s00348-021-03218-3.

Scolaro, A., Clifford, I., Fiorina, C., and Pautz, A. (2020). The OFFBEAT multi-dimensional fuel behavior solver. Nucl. Eng. Des. 358: 110416, https://doi.org/10.1016/j.nucengdes.2019.110416.

Zum Wandel in den Wissenschaften durch datenintensive Forschung | Positionspapier, (Drs. 8667-20) (2020). German Science and Humanities Council.