UK Atomic Energy Authority

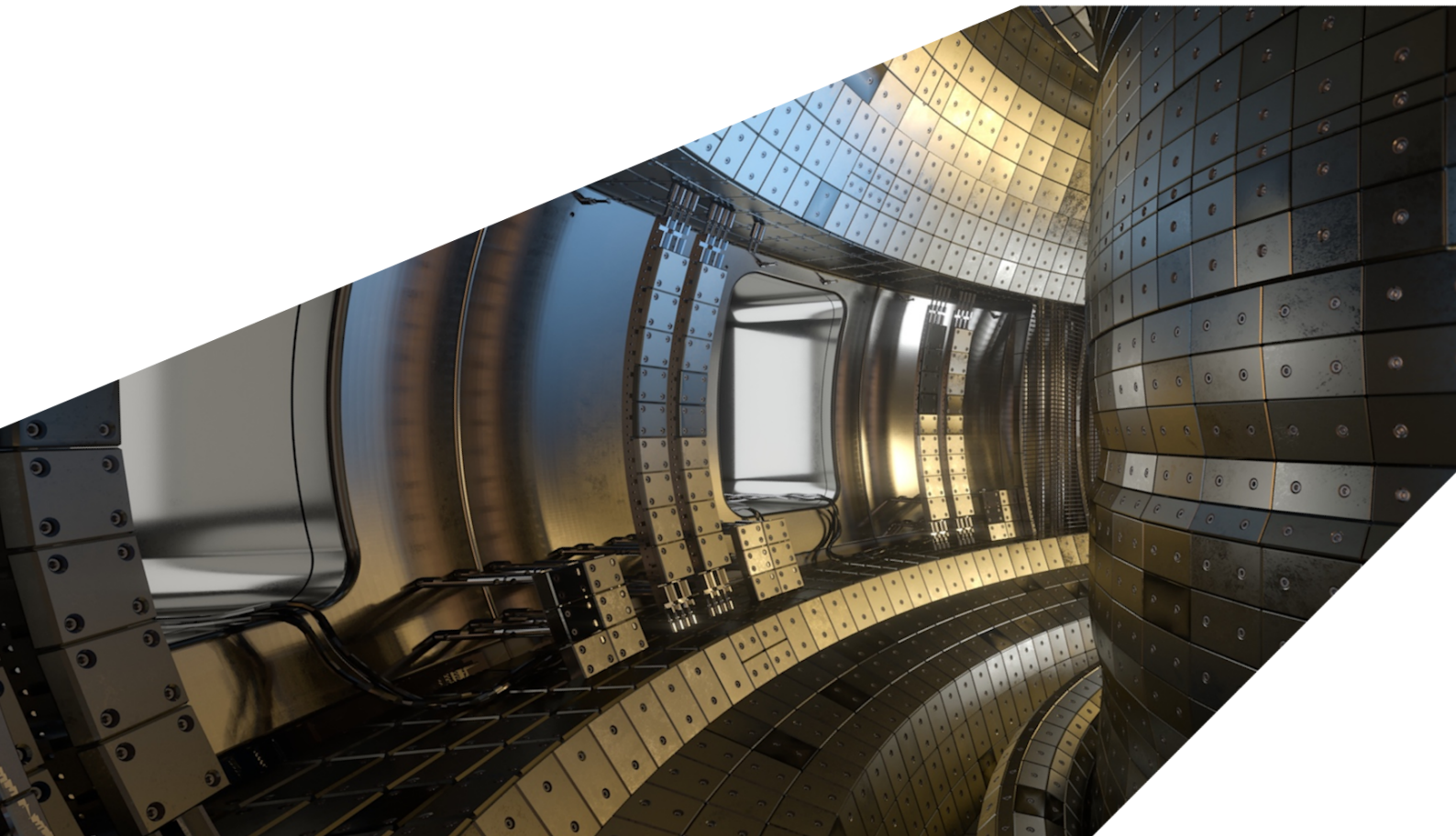
ExCALIBUR

Specification and Integration of Scientific Software

M3.1.4

Abstract

The report describes work for ExCALIBUR project NEPTUNE at Milestone M3.1.4. This report explains how design options are to be decided and communicated for a generic opensource development of scientific modelling software by a community. The document focuses on the mechanics of such a development highlighting important issues that need to be agreed as early as possible. It includes practical points concerning frequency of meetings and software releases, together with workflows, organisation of project repositories and notes the importance of code reviews. In places it descends into fine detail, notably regarding the use of the `git` version control system and of naming conventions for the C++ and Object Fortran programming languages. Each prescription is accompanied by arguments for the choice made, often referring to the published literature. There are also policy recommendations. For efficiency and accuracy, the report recommends 'write once, re-use many times', not only for code, but also and especially for documentation. For the package to remain contemporary over a generation, the report argues for a 'rule of two' to enable exploitation and possible incorporation of promising new software and algorithms.



UKAEA REFERENCE AND APPROVAL SHEET

	Client Reference:	
	UKAEA Reference:	CD/EXCALIBUR-FMS/0042
	Issue:	1.00
	Date:	31 October, 2021
Project Name: ExCALIBUR Fusion Modelling System		

	Name and Department	Signature	Date
Prepared By:	Ed Threlfall Wayne Arter BD	N/A N/A	31 October, 2021 31 October, 2021
Reviewed By:	Rob Akers Advanced Computing Dept. Manager		31 October, 2021
Approved By:	Rob Akers Advanced Computing Dept. Manager		31 October, 2021

1 Introduction

This document, drawing on the "Development Plan" document [1] and on the "Charter" [2], forms the basis for a generic opensource development of scientific software by a community. Whereas ref [1] discusses the documents that need to be produced, the present document presents guidance concerning the mechanics of such a development particularly other important issues that need to be agreed as early as possible. These include practical points concerning frequency of meetings, code review etc., designed to ensure efficient collaboration between a wide group of project partners. It is assumed that all the community has signed up to the "Charter" [2], key generic points of which are reproduced in Annex Section A.

It is important to distinguish use-cases. There appear to be three use-cases worth treating separately:

1. code produced for immediate and local use only, eg to test out an idea or to illustrate a scientific paper.
2. software for long-term use, where execution speed is time-critical, eg for real-time control or inter-shot discharge analysis.
3. other software intended for widespread long-term usage.

This note relates most closely to the third case. Its recommendations are broadly consistent with those laid out by Bungarth & Heister [3], and in particular those for the usage of `git` conform to practice recommended by the ITER organisation. This document is not the place for a general discussion of software engineering practices, and does not cover code coupling, both of which topics are discussed in the open literature, see in particular Lawrence et al [4] for HPC software engineering and Belete et al [5] for code coupling, also see other NEPTUNE reports, particularly refs [6, 7, 8].

The present document seeks not merely to prescribe, but to give compelling arguments for the choices made in respect of guidelines. Generally, efforts will be made to ensure consensus or at least agreement between the two most affected project partners on any decisions taken. However, in the event of continuing disagreement, the technical leader or 'Lead' for the project will ultimately decide on the basis of technical evidence presented, subject to ratification by higher management. A general rule is always to allow two options ('rule of two'), intended to enable exploitation and possible incorporation ofeany promising new software (eg. package, library or language) or relevant algorithm which emerges during the course of the project. Since however, each option doubles the potential cost of developing and maintaining software, a good case must be made to the Lead for a new option, and the innovator include provision for retiring one of the existing options should there already be two. Implicitly thereby, as discussed at the end of Section 3.2, a third exploratory option is also allowed.

A similar recommendation (rather than rule) regarding both code and documentation is to 'write once, re-use many times'. This to a large extent explains a preference for LaTeX2HTML as enabling multiple reuse of the same text and mathematical expressions in different documents and on different webpages, via use of LaTeXinput command.

The guidelines for a development are set out in the body of this document in an arrangement consistent with the concordance, such that separate sections correspond to the different documents/webpages required. Thus appearing first are items relevant to management related points (MGT) in Section 2, then the technical specification (TS) in Section 3, and finally operational aspects (OP) in Section 4.

2 Management MGT

Meetings, whether on-line or in-person are regarded as critical for good collaboration, and are discussed in Section 2.1. The other key collaborative element centres naturally on the software, where use of the `git` control system, see Section 2.2 and consequent use of repositories, see Section 2.3, is becoming universal.

2.1 Meetings and Workshops

To start the project, a kick-off meeting should bring together all partners who will contribute significant code to the project. The aim of this meeting will be to build personal links among the team, and to establish community practices consistent with the charter. Efforts should be made to build consensus and a community spirit within the project team.

A regular project planning and monitoring meeting should be set up, at least monthly. The agenda would include short updates on progress of each project component, and focus on the project planning and coordination. In addition, a separate series of seminars and training should be organised, where each partner might give a longer talk on an aspect of their work, for example showing other partners how to use recently developed capabilities.

Development and collaboration mechanisms should include:

1. A system of code repositories for version control (eg. `github`)
2. Automated testing infrastructure (eg. `github` actions)
3. Documentation infrastructure, ie. as a website
4. A repository for long-term storage of large files, records of meetings, presentations etc. (eg. Google shared drive)
5. A chat/messaging service such as Slack, to facilitate interactions between developers

As these are established, a series of training workshops should be arranged. These should include talks on the “high level” objectives, on the near-term plans of each partner, and also hands-on training in the tools being used.

2.2 Version control

The standard `git` version control system should be used; there is no viable competitor to this in terms of capabilities, widespread adoption, or integration into other tools and services (eg. `github`).

A common complaint against `git` is the user interface, which can be intimidating to new users. There are very strong reasons why even programmers with plenty of other experience, should seek guidance and preferably training in use of the command line interface (CLI). For those who have time enough to attempt to do so without, a few hints are provided:

1. The complexity of the interface can be mitigated by restricting usage to a few well-chosen subcommands such as `clone`, `add`, `commit`, `push`, `pull`, `diff`, `log` and `status`.

2. Exercise caution before using other subcommands or new options to the core subcommands, eg. by first committing all files, adding a suboption which indicates what will be done without actually modifying any files, and avoiding forcing options.

3. For the purpose of the key subcommands such as 'pull' and 'push', it is important to remember than these are are defined from the user's point-of-view, so that 'pull' brings source from the repo towards the user, and 'push' sends it away. There are other non-intuitive aspects so that it is important to study very carefully the description of any new sub-command/option and particularly its ordering of options.

4. Since the software is widely used, error messages can invariably be 'googled' for further explanation.

5. Should conflicts occur, these are recorded by the insertion of strings '+++...', '>>>>...' and '<<<<...' in disc files to indicate lines where the clashes lie. Many users find resolving conflicts very difficult on the basis of such information, however making up for the absence of a GUI mechanism within `git` to do this, it is possible to integrate GUIs such as **meld**, being aware of possible system dependences.

Otherwise, the experience of `git` can be mitigated through:

- Training: Links to training material for adopted tools should be made available as part of the project documentation. This should be supplemented by training, both one-to-one and as part of a programme of talks and training.

- Adoption of, and training in, tools to provide easier interfaces. `github` itself allows browsing of history; `Magit` is an excellent interface integrated into Emacs; and similar tools exist for eg. Visual Studio Code. The ITER organisation uses `bitbucket` and UKAEA uses `gitlab`.

2.3 Code repositories

The structure of ExCALIBUR will result in a number of different components, experimental proxyapps, and increasingly complex applications. There are two main different approaches as to how these different components could be split between git repositories, namely (1) Several large code bases are kept in a single repository (a 'monorepo') and (2) projects are kept in separate repositories, with dependencies being included as `git` submodules.

Advantages of 'monorepos' include simplified dependencies, synchronisation of changes to different components, and a central location for documentation. The main argument for having separate repositories is for the occasion when several components already pre-exist as separate repositories. For new modules, strong coupling between components should be discouraged, so that components can be reused in a range of applications.

The recommendation is a compromise approach whereby:

- A `github` 'organisation' is created to host new repositories. Organisations allow permissions for groups of administrators and developers to be managed.
- Individual components and proxyapps are hosted in separate repositories under this organisation. These contain the code, unit tests, documentation etc. specific to these components.
- A central repository under this organisation includes components as sub-modules. These could be organised into a directory structure, with documentation explaining the relations or coupling between components. In this repository would go:
  - Integration tests which couple components and ensure that they work together
  - Documentation of the interfaces between components, project conventions (eg. style guides), and overall project aims.

  Sub-modules are pinned to a particular `git` commit, so that at any point the versions included are those which are known to work with each other. A developer who wants the latest version of a component would clone the individual repository, while a user who wants something that "just works" would clone the central repository.

There is the disadvantage of a tie specifically to `github`, but loss of the 'organisation' capability would be expected to be an inconvenience rather than a disaster for a project.

2.4 Development workflow

The standard `git` work flow should be adopted, since this is widely familiar and has been developed as best practice based on industrial experience. Exceptions are allowed for minor issues, such as typographical errors and broken links in documentation .

Each code component should maintain a `main` branch (often referred to as the 'master' as in 'master copy'), which can only be modified through a pull request mechanism which ensures peer review and testing. Bug fixes and feature development should be done in separate branches, either in the same repository, or in forked repositories. When someone encounters a bug, or wishes to develop a new feature, a good approach has been found (with BOUT++) to be:

1. An issue is opened, describing the bug or feature request or proposal. This allows discussion of the issue, and possible approaches to addressing it.
2. A pull request is opened as early as possible, marked "Work in progress" or similar. This can contain only minimal code or outline of the code structure. This links to the issue, lets other people know that it is being worked on, and enables peer review and input into the development direction.
3. Once ready for merging, and consensus has been reached that the proposed change should be made, then it is merged.

If a code is sufficiently large, then a further degree of separation between the stable `main` branch and active development is needed. A common pattern is to only branch off and merge features

into a `next` branch. Periodically this branch is merged into `main` as a new release, once the new features are judged to be sufficiently mature and tested.

Whether into `main` or `next`, pull requests should be reviewed using a checklist the remind the reviewers. Review involves testing, aspects of which are addressed in Section 4.2.

It must be stressed that code review is not a job separate from code development: **All developers should be expected to participate in and carry out code reviews**. Reviewing code benefits not only the original author, but also the reviewer. Through the discussion, it contributes to a sense of shared ownership of the code base, and spreads good practices. There is the implication that code should be written 'for the other guy', ie. so that the other guy can understand it without much difficulty. It also ensures that at least two developers know how each part of the code works.

2.5 Code release

Code releases should be a regular occurrence. Code release helps with project branding and user engagement, and ensures that the project is seen as active. It also helps project administration by ensuring new features are shared in a timely fashion, and by reducing the number of long-lived divergent branches.

The project NEPTUNE codebase will consist of proxyapps, and infrastructure code that interfaces them. A code release will consist of a version of this infrastructure code, plus commit hashes that fix the versions of the proxyapps. As proxyapps might be independent projects with their own established release cycle, the following release policy applies only to the infrastructure code. It is the recommended policy for new proxyapps written under Project NEPTUNE.

Release numbering should follow (a modified) Semantic Versioning approach [9], summarized as

> "Given a version number MAJOR.MINOR.PATCH, increment the:
>
> 1. MAJOR version when you make incompatible API changes,
> 2. MINOR version when you add functionality in a backwards compatible manner, and
> 3. PATCH version when you make backwards compatible bug fixes."

Here it is understood that "API" refers to user-facing interfaces; APIs to functions internal to proxyapps may break backwards compatibility in MINOR releases. There is however an absolute guarantee that no backwards incompatible changes are made for end users in MINOR and PATCH releases, except those that arise from fixing a bug. That is, physics results are permitted to change in such releases if the new release's results are "correct" and the previous release's results were "wrong".

It is also understood that releases with MAJOR number 0 are considered beta releases, for which there are no guarantees of backwards compatibility.

Each release will be uploaded as a Zenodo[10] record with its own DOI. This gives a clear citation for the project (to be included in the project's README or CITATION.cff file), while ensuring that

developers receive credit for their work, without the need for associating each release with a publication. Encouraging researchers to use release versions and to cite by version number also aids scientific reproducibility.

While some technical aspects of the release process can be automated, many of the tasks, such as curating issues and writing release notes, are inherently manual. To prevent NEPTUNE relying on a single person to make releases, the exact workflow will be codified and included in the project documentation. An example of such a workflow for the GS2 project may be found online [11].

3 Technical Specification

3.1 Code licence and availability

Code should be made available to collaborators at the earliest opportunity, to maintain close alignment between groups.

- To minimise friction and unnecessary legal restrictions on combining code components, a common licence should be adopted, and BSD 3 is recommended. Licences less restrictive than the common licence such as MIT may be used.
- There is little reason to put artificial barriers to obtaining code, or to embargo code for periods of time. Unless there are strong arguments, code development should be carried out in public repositories. The benefits of minimising delay to code use, feedback and peer review, outweigh any potential for embarrassment or code misuse.

3.1.1 Code style

There are many different code styles, each of which have their proponents, and can be debated at length. While everyone has their own favourite style, it seems likely that the choice of style makes little difference to objective quality or productivity. Anecdotal experience and experience from the gaming world in developing large, complicated packages (eg. Gregory [12, § 3]), indicate however that it is very important that there is a well-defined code style and that developers stick to it, since a mixture of styles in a code base adds unnecessary mental load and overhead. It is recommended to choose a style and enforce it.

1. Formatting

   The popularity of prescriptive code formatting tools (e.g BLACK, GOFMT, CLANG-FORMAT, RUSTFMT) is a testament to the popularity of this pragmatic approach, and provides ready-made tools which can be adopted and are likely already familiar to many developers.

   - Choice of style: Universal agreement is unlikely, because it is a matter of taste rather than objective fact. A style should be chosen by the Lead, who could choose to adopt a style used by an underlying framework, eg. Nektar++ or BOUT++.

- Developer tools: Code formatting tools should be used to automatically format code. For C++ there is `clang-format`, while for python `black` is widely used. Similar tools should be chosen for any other languages adopted by the project.
- Enforcement: Tests run on pull requests and code pushes to the shared repository should include code formatting: The automated formatter is applied to the code, and if the output is different from the input then the code is incorrectly formatted, and the test fails.
- Documentation: LaTeX or Markdown should be used and conventions enforced regarding line length, restriction to ASCII character set, abbreviations, hyphenation, capitalisation, minimal use of 'z', and use of fonts to denote code names. Usage of LaTeX2HTML implies a restricted set of packages.

2. Naming

   Naming of code components (modules, classes, functions, variables etc.) is less easy to enforce automatically than formatting. There are different styles, but some widely applicable good practices which should be adopted:

   - Consistency: Whatever convention is used, stick to it.
   - Be descriptive: Names should be meaningful, not cryptic, and need not be very short. In some cases it is tempting and even useful to use symbols (e.g single characters) which correspond to a mathematical expression. In this case the mathematical expression should be in the documentation within or linked to the code.
   - Generally prefer nouns for variables, and verbs for functions

   Some code styles for C++ (eg. BOUT++, adapted from LLVM and with features of the the JSF coding style [13, § 6.6]), use different cases for different types of things: `snake_case` for variables, `camelCase` for functions/methods, and `PascalCase` for classes and types. However, given a need to mix with Object Fortran, which is not case-sensitive, 'pot-hole', ie. separating name elements by underscores, is recommended.

   Occasionally a convention is used where the name includes a part which indicates the type of the variable. For example, the JSF style for C++ recommends that pointer names begin 'p_' and that private or protected ('member') variables should have names beginning with 'm'. In general naming conventions are probably not essential, since the type can be read in the code, and modern IDEs will easily provide this information to the developer. Nonetheless, there is no objection to employing a convention, and a project should recommend one (but only one) for coders who wish to do so.

   For Object Fortran naming conventions, Arter et al [14] codifies best practice. For C++, based on the recommendations of the book "Professional C++" (eg. ref [15, § 7], the following prefix strings should be employed: 'm_' for member (particularly useful for indicating scope), 'p_' for pointer, 's_' for static, 'k_' for constant, 'f_' for flag (Boolean value), and aggregations thereof, eg. 'ms_variable'. Use of global variables is deprecated, so the 'g_' prefix should not be used. It may also be useful to reserve the single letters 'i', 'j', 'k' etc. for the names of loop-count variables.

3.2 Programming languages

It is generally recommended a small set of "approved" languages be used in the project consistent with the rule of 'two' described in the Introduction Section 1. These rules should cover the high performance code itself, but also the input/output, testing scripts and other infrastructure included in the code repository.

Important factors in the choice made included:

1. Widespread use. It must be possible for several project members at any one time to understand the language, and be able to maintain the code.

2. Stability. The code developed will potentially have a long life-span, and there are insufficient resources to continually update code to respond to upstream changes.

3. Previous usage in HPC and scientific computing. There should be an existing ecosystem of code packages, tutorials, and potential users.

The above considerations implied that the following options should be extensively discussed.

- C++14, Fortran (eg. 2008), C and Python all satisfy the above criteria. For configuration, CMake, Autotools and Bash also qualify.

- SYCL (building on C++) and Julia are both less widely adopted so far, but both appear to be heading towards satisfying the above criteria and might be considered.

The recommended languages are

- As higher level DSL : Python and Julia

- For lower level HPC compatibility/DSL: Kokkos and SYCL

- General scientific work : the latest versions of C++ and (Object) Fortran, provided they are compatible with pre-existing packages and reliable compilers are available (eg. as of mid-2021 usage of SYCL implies a need for C++17.)

- For code compilation and linking etc. : CMake

Other languages may have technical merits in particular areas, or are being adopted outside scientific computing but not to a significant degree within the community. Use of these languages should be limited to isolated experiments, rather than core components. If shown to be useful in these experiments, to a level which is worth the additional overhead and risk of maintaining it, then the Lead should consider expanding the list of approved languages, consistent with the 'rule of two'.

4 Operational documentation OP

4.1 Documentation

Documentation refers to a particular version of the code. It should therefore be dynamic, under version control, and tightly coupled to the source code itself.

- All new code features should be documented, and this should be checked as part of the peer review process.
- Within the code, comments should use a convention, such as that accepted by DOXYGEN, to document the intent of functions, and any assumptions on their environment, input or outputs.
- Alongside the code `README` files explaining the file/directory layout typically use the Markdown format due to its simplicity, standardising on the variant defined by PANDOC as described in [1].
- The more formal documentation should be in a format which can include elements such as equations, code blocks, graphs and figures. It should also be easily convertible to other formats, and in particular online documentation. LaTeX as used to produce the current document can be easily converted to .html as explained in ref [1] provided the restrictions (as to accepted packages) noted in the reference are observed.

4.2 Testing

Requirements before merging changes in `git` include:

- Tests must pass. (Merge blocking can be enforced eg. on `github`.)
- The code must be in the standard style, which will be at least partly enforced as part of the automated testing.
- Documentation must be updated or added to reflect changes in the code.

4.2.1 Source code testing

Testing of code is essential to ensure correctness, reduce incidents of accidental breakage or regression of code features, and enable code changes to be made with confidence. These tests must be automated, and as far as possible be “unit” tests, which test isolated components of the code. A strictly Test Driven Development (TDD) approach is not always appropriate, but encouraging incremental development and testing in small pieces has several advantages in terms of the resulting code structure and maintainability:

1. It encourages the writing of code which has “clean” interfaces ie. a well defined set of inputs and outputs, with minimal side channels (eg. global state).
2. Having to test components individually discourages strong coupling between code, because then these dependency components have to be “mocked” up in testing.
3. Good code test coverage makes later maintenance, modification and refactoring of the code easier. The tests also function as a type of documentation of the intended use of the code, and also of the corner-cases which may not be obvious to a new user or developer.

The most important types of tests are for correctness. These can use standard services such as `github actions`, `Travis` etc. Performance is however a crucial property of the code, and should also be monitored.

4.2.2 Performance testing

It is useful to include timing information in test output, which is then contained in the test logs. This is valuable as a quick way for developers to observe the impact of changes on performance. It is however not very accurate, especially under virtual machines on shared hardware as is typical for testing services. These tests also only typically use a small number of processors (less than four), usually without accelerator support, making them of limited use in evaluating performance of high performance code for the Exascale.

Periodic testing of code versions on a range of hardware will be needed to monitor performance, and catch performance regressions. This could be carried out by a researcher, but the possibility of automating this process and making use of services such as Amazon AWS HPC and GPU servers. Studies carried to date indicate a lack of appropriate software for ensuring performance portability and a consequent need at least to enhance existing packages.

Acknowledgement

The support of the UK Meteorological Office and Strategic Priorities Fund is acknowledged. Based on a document provided by Ben Dudson, "Suggestions for ExCALIBUR community guidelines planning", dated December 16th, 2020.

References

[1] W. Arter, J. Parker, and E. Threlfall. Development Plan. Technical Report CD/EXCALIBUR-FMS/0033-D3.4, UKAEA, 2021.

[2] W. Arter. EXCALIBUR NEPTUNE Charter. Technical Report CD/EXCALIBUR-FMS/0020, UKAEA, 2020.

[3] W. Bangerth and T. Heister. What makes computational open source software libraries successful? *Computational Science & Discovery*, 6(1):015010 (18 pages), 2013.

[4] B.N. Lawrence, M. Rezny, R. Budich, P. Bauer, J. Behrens, M. Carter, W. Deconinck, R. Ford, C. Maynard, S. Mullerworth, et al. Crossing the chasm: how to develop weather and climate models for next generation computers? *Geoscientific Model Development*, 11(5):1799–1821, 2018.

[5] G.F. Belete, A. Voinov, and G.F. Laniak. An overview of the model integration process: From pre-integration assessment to testing. *Environmental modelling & software*, 87:49–63, 2017.

[6] W. Arter, E. Threlfall, J. Parker, and S. Pamela. Report on user frameworks for tokamak multiphysics. Technical Report CD/EXCALIBUR-FMS/0022-M3.1.2, UKAEA, 2020.

[7] E. Threlfall, J. Parker, and W. Arter. Design patterns evaluation report. Technical Report CD/EXCALIBUR-FMS/0026-M3.3.3, UKAEA, 2020.

[8] J. Parker, E. Threlfall, W. Arter, and W. Saunders. Code coupling and benchmarking. Technical Report CD/EXCALIBUR-FMS/0053-M7.2, UKAEA, 2021.

[9] Semantic Versioning 2.0.0. `https://semver.org`, 2021. Accessed: November 2021.

[10] Zenodo, to ensure that everyone can join in Open Science. `https://zenodo.org`, 2021. Accessed: November 2021.

[11] A fast, flexible, physicist's toolkit for gyrokinetics. `https://gyrokinetics.gitlab.io/gs2/page/admin_manual/release_instructions/index.html`, 2021. Accessed: November 2021.

[12] J. Gregory. *Game engine architecture 3rd Ed.* AK Peters/CRC Press, 2017.

[13] J. Pitt-Francis and J. Whiteley. *Guide to scientific computing in C++*. Springer, 2017.

[14] W. Arter, N. Brealey, J.W. Eastwood, and J.G. Morgan. Fortran 95 Programming Style. Technical Report CCFE-R(15)34, CCFE, 2015. http://dx.doi.org/10.13140/RG.2.2.27018.41922,https://scientific-publications.ukaea.uk/wp-content/uploads/CCFE-R-1534.pdf.

[15] N.A. Solter and S.J. Kleper. *Professional C++*. John Wiley & Sons, also https:/www.wrox.com, 2005. 5th edition dated 2021 exists in Wrox series with different author M. Gregoire.

A Key features of Project Charter

A high-level objective is to ensure that developed software is of the highest quality, implying a rigid requirement around the production of high-quality documentation and reproducible verification and validation tests for the codebase as it evolves. Since development work may transfer between institutions, it is important that common standards for documentation and testing be available and easy to deploy.

The community is therefore expected to be agile and amenable to change once it is clear which are the most promising long–term solutions. One selection for the long-term framework/code(s) might force refactoring of code developed consistent with another choice code base, and where feasible, community members should support this process.

Source code for all development should be accessible by the entire community and all tests should be repeatable by different workers without the need for re-training and/or any possible confusion as to the procedures and metrics needed to declare a test successful.

All proxyapps and related infrastructure/documentation across the project should meet the demands of project standards as they develop, to:

- adopt a consistent choice of definitions (ontology) of objects or equivalently classes,
- adhere to clearly defined common file formats and interfaces to components for data input and output.
- provide suitably flexible data structures for common use by all developers,
- are established through good scientific software engineering best practice,
- demonstrate performance portability and exploit agreed DSL-like interfaces where possible targeting Exascale-relevant architectures,
- can be integrated into a VVUQ framework and
- are embedded within a coordination and benchmarking framework for correctness testing and performance evaluation.