State of the Practice for Medical Imaging Software Based on Open Source Repositories

W. Spencer Smith[1,*], Ao Dong[1], Jacques Carette[1], and Michael D. Noseworthy[2]

[1]McMaster University, Computing and Software Department, Canada
[2]McMaster University, Electrical & Computer Engineering Department, Canada
[*]Corresponding Author

May 21, 2024

Abstract

We present the state of the practice for Medical Imaging (MI) software. We selected 29 medical imaging projects from 48 candidates, assessed 10 software qualities (installability, correctness/ verifiability, reliability, robustness, usability, maintainability, reusability, understandability, visibility/transparency and reproducibility) by answering 108 questions for each software project, and interviewed 8 of the 29 development teams. Based on the quantitative data for the first 9 qualities, we ranked the MI software with the Analytic Hierarchy Process (AHP). The four top ranked software products are: *3D Slicer*, *ImageJ*, *Fiji*, and *OHIF Viewer*. Our ranking is mostly consistent with the community's ranking, with four of our top five projects also appearing in the top five of a list ordered by stars-per-year. Generally, MI software is in a healthy state as shown by the following: in the repositories we observed 88% of the documentation artifacts recommended by research software development guidelines, 100% of MI projects use version control tools, and developers appear to use the common quasi-agile research software development process. However, the current state of the practice deviates from the existing guidelines because of the rarity of some recommended artifacts (like test plans, requirements specification, code of conduct, code style guidelines, product roadmaps, and Application Program Interface (API) documentation), low usage of continuous integration (17% of the projects), low use of unit testing (about 50% of projects), and room for improvement with documentation (six of nine developers felt their documentation wasn't clear enough). From interviewing the developers, we identified five pain points and two qualities of potential concern: lack of development time, lack of funding, technology hurdles, ensuring correctness, usability, maintainability, and reproducibility. The interviewees proposed strategies to improve the state of the practice, to address the identified pain points, and to improve software quality. Combining their ideas with ours, we have the following list of recommendations: increase documentation, increase testing by enriching datasets, increase continuous integration usage, move to web applications, employ linters, use peer reviews, design for change, add assurance cases, and incorporate a "Generate All Things" approach.

Keywords: medical imaging, research software, software engineering, software quality, analytic hierarchy process, developer interviews

1 Introduction

We aim to study the state of software development practice for Medical Imaging (MI) software. MI tools use images of the interior of the body (from sources such as Magnetic Resonance Imaging (MRI), Computed Tomography (CT), Positron Emission Tomography (PET) and Ultrasound) to provide information for diagnostic, analytic, and medical applications (Administration, 2021; Wikipedia contributors, 2021d; Zhang et al., 2008). Figure 1, which shows an image of the brain, highlights the importance and value of MI. Through MI medical practitioners and researchers can noninvasively gain insights into the human body, including information on injuries and illnesses. Given the importance of MI software and the high number of competing software projects, we wish to understand the merits and drawbacks of the current development processes, tools, and methodologies. We aim to assess through a software engineering lens the quality of the existing software with the hope of highlighting standout examples, understanding current pain points and providing guidelines and recommendations for future development.

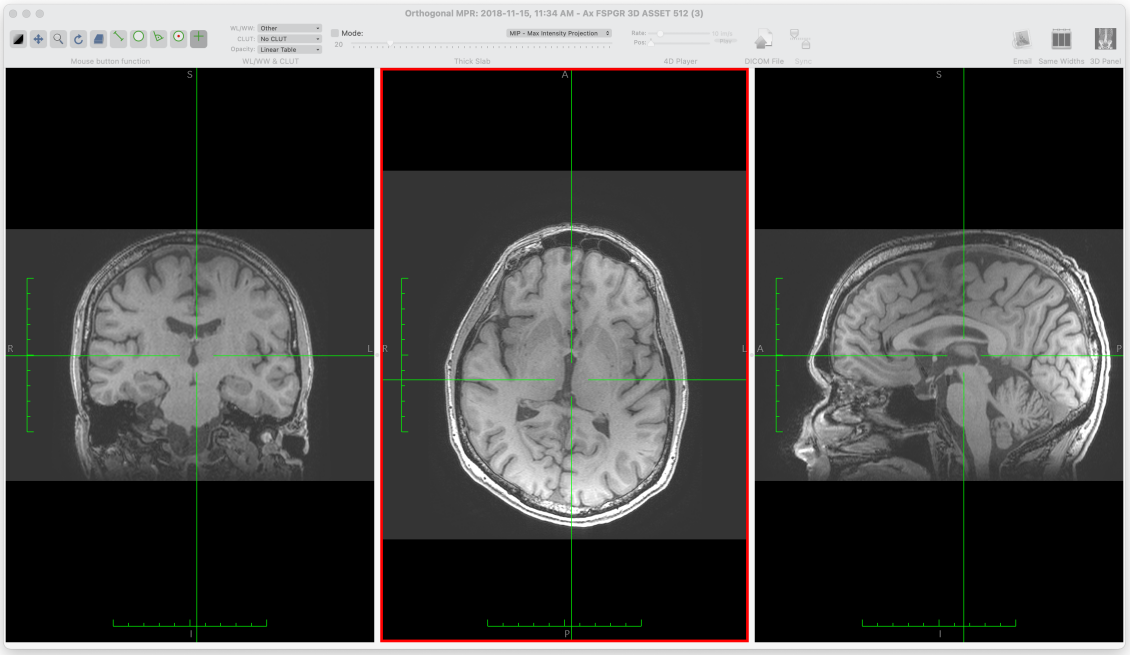

Figure 1: Example brain image showing a multi-planar reformat using Horos (free open-source medical imaging/DICOM viewer for OSX, based on OsiriX)

1.1 Research Questions

Not only do we wish to gain insight into the state of the practice for MI software, we also wish to understand the development of research software in general. We wish to understand the impact

of the often cited gap, or chasm, between software engineering and research software (Kelly, 2007; Storer, 2017). Although scientists spend a substantial proportion of their working hours on software development (Hannay et al., 2009a; Prabhu et al., 2011), many developers learn software engineering skills by themselves or from their peers, instead of from proper training (Hannay et al., 2009a). Hannay et al. (2009a) observe that many scientists showed ignorance and indifference to standard software engineering concepts. For instance, according to a survey by Prabhu et al. (2011), more than half of their 114 subjects did not use a proper debugger when coding.

To gain insights, we devised 10 research questions, which can be applied to MI, as well as to other domains, of research software (Smith and Michalski, 2022; Smith et al., 2021). We designed the questions to learn about the community's interest in, and experience with, software artifacts, tools, principles, processes, methodologies, and qualities. When we mention artifacts we mean the documents, scripts and code that constitutes a software development project. Example artifacts include requirements, specifications, user manuals, unit tests, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code. Once we have learned what MI developers do, we then put this information in context by contrasting MI software against the trends shown by developers in other research software communities. Our aim is to collect enough information to understand the current pain points experienced by the MI software development community so that we can make some preliminary recommendations for future improvements.

We based the structure of the paper on the research questions, so for each research question below we point to the section that contains our answer. We start with identifying the relevant examples of MI software for the assessment exercise:

RQ1: What MI software projects exist, with the constraint that the source code must be available for all identified projects? (Section **??**)

RQ2: Which of the projects identified in RQ1 follow current best practices, based on evidence found by experimenting with the software and searching the artifacts available in each project's repository? (Section **??**)

RQ3: How similar is the list of top projects identified in RQ2 to the most popular projects, as viewed by the scientific community? (Section **??**)

RQ4: How do MI projects compare to research software in general with respect to the artifacts present in their repositories? (Section **??**)

RQ5: How do MI projects compare to research software in general with respect to the use of tools (Section 4) for:

RQ5.a development; and,

RQ5.b project management?

RQ6: How do MI projects compare to research software in general with respect to principles, processes, and methodologies used? (Section 5)

RQ7: What are the pain points for developers working on MI software projects? (Section 6)

RQ8: How do the pain points of developers from MI compare to the pain points for research software in general? (Section 6)

RQ9: For MI developers what specific best practices are taken to address the pain points and software quality concerns? (Section 6)

RQ10: What research software development practice could potentially address the pain point concerns identified in RQ7? (Section 7)

1.2 Scope

To make the project feasible, we only cover MI visualization software. As a consequence we are excluding many other categories of MI software, including Segmentation, Registration, Visualization, Enhancement, Quantification, Simulation, plus MI archiving and telemedicine systems (Compression, Storage, and Communication) (as summarized by Bankman (2000) and Angenent et al. (2006)). We also exclude Statistical Analysis and Image-based Physiological Modelling (Wikipedia contributors, 2021c) and Feature Extraction, Classification, and Interpretation (Kim et al., 2011). Software that provides MI support functions is also out of scope; therefore, we have not assessed the toolkit libraries VTK (Schroeder et al., 2006) and ITK (McCormick et al., 2014). Finally, Picture Archiving and Communication System (PACS), which helps users to economically store and conveniently access images (Choplin et al., 1992), are considered out of scope.

1.3 Methodology Overview

We designed a general methodology to assess the state of the practice for research software (Smith and Michalski, 2022; Smith et al., 2021). Details can be found in Section 3. Our methodology has been applied to MI software (Dong, 2021) and Lattice Boltzmann Solvers (Michalski, 2021; Smith et al., 2024). This methodology builds off prior work to assess the state of the practice for such domains as Geographic Information Systems (Smith et al., 2018b), Mesh Generators (Smith et al., 2016b), Seismology software (Smith et al., 2018e), and Statistical software for psychology (Smith et al., 2018c). In keeping with the previous methodology, we have maintained the constraint that the work load for measuring a given domain should be feasible for a team as small as one person, and for a short time, ideally around a person month of effort. We consider a person month as 20 working days (4 weeks in a month, with 5 days of work per week) at 8 person hours per day, or $20 \times 8 = 160$ person hours.

With our methodology, we first choose a research software domain (in the current case MI) and identify a list of about 30 software packages. (For measuring MI we used 29 software packages.) We approximately measure the qualities of each package by filling in a grading template. Compared with our previous methodology, the new methodology also includes repository based metrics, such as the number of files, number of lines of code, percentage of issues that are closed, etc. With the quantitative data in the grading template, we rank the software with the Analytic Hierarchy Process (AHP) (Section 2 provides details). After this, as another addition to our previous methodology, we interview some development teams to further understand the status of their development process.

2 Background

To measure the existing MI software we need two sets of definitions: i) the definitions of relevant software license models (Section 2.1); and, ii) the definitions of the software qualities that we will be assessing (Section 2.2). In our assessment we rank the software packages for each quality; therefore, this section also provides the background on our ranking process — the Analytic Hierarchy Process (Section 2.3).

2.1 Software Categories

When assessing software packages, we need to know the software's license. In particular, we need to know whether the source code will be available to us or not. We define three common software categories. We will only assess software that fits under the Open Source Software license.

- **Open Source Software (OSS)** For OSS, the source code is openly accessible. Users have the right to study, change and distribute it under a license granted by the copyright holder. For many OSS projects, the development process relies on the collaboration of different contributors worldwide (Corbly, 2014). Accessible source code usually exposes more "secrets" of a software project, such as the underlying logic of software functions, how developers achieve their works, and the flaws and potential risks in the final product. Thus, OSS is suitable for researchers analyzing the qualities of a project.

- **Freeware** Freeware is software that can be used free of charge. Unlike OSS, the authors of do not allow access or modify the source code (Project, 2006). To many end-users, the differences between freeware and OSS may not be relevant. However, software developers who wish to modify the source code, and researchers looking for insight into software development process may find the inaccessible source code a problem.

- **Commercial Software** "Commercial software is software developed by a business as part of its business" (GNU, 2019). Typically speaking, commercial software requires users to pay to access all of its features, excluding access to the source code. However, some commercial software is also free of charge (GNU, 2019). Based on our experience, most commercial software products are not OSS.

2.2 Software Quality Definitions

Quality is defined as a measure of the excellence or worth of an entity. As is common practice, we do not think of quality as a single measure, but rather as a set of measures. That is, quality is a collection of different qualities, often called "ilities." Below we list the 10 qualities of interest for this study. The order of the qualities follows the order used in Ghezzi et al. (2003), which puts related qualities (like correctness and reliability) together. Moreover, the order is roughly the same as the order developers consider qualities in practice.

- **Installability** The effort required for the installation and/or uninstallation of software in a specified environment (ISO/IEC, 2011; Lenhard et al., 2013).

- **Correctness & Verifiability** A program is correct if it matches its specification (Ghezzi et al., 2003, p. 17). The specification can either be explicitly or implicitly stated. The related quality of verifiability is the ease with which the software components or the integrated product can be checked to demonstrate its correctness.
- **Reliability** The probability of failure-free operation of a computer program in a specified environment for a specified time (Musa et al., 1987), (Ghezzi et al., 2003, p. 357).
- **Robustness** Software possesses the characteristic of robustness if it behaves "reasonably" in two situations: i) when it encounters circumstances not anticipated in the requirements specification, and ii) when users violate the assumptions in its requirements specification (Ghezzi et al., 2003, p. 19), (Boehm, 2007).
- **Usability** "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" (ISO/TR, 2002, 2018).
- **Maintainability** The effort with which a software system or component can be modified to i) correct faults; ii) improve performance or other attributes; iii) satisfy new requirements (Boehm, 2007; IEEE, 1991).
- **Reusability** "The extent to which a software component can be used with or without adaptation in a problem solution other than the one for which it was originally developed" (Kalagiakos, 2003).
- **Understandability** "The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use" (ISO, 2001).
- **Visibility/Transparency** The extent to which all the steps of a software development process and the current status of it are conveyed clearly (Ghezzi et al., 2003, p. 32).
- **Reproducibility** "A result is said to be reproducible if another researcher can take the original code and input data, execute it, and re-obtain the same result" (Benureau and Rougier, 2017).

2.3 Analytic Hierarchy Process (AHP)

Saaty developed AHP in the 1970s, and people have widely used it since to make and analyze multiple criteria decisions (Vaidya and Kumar, 2006). AHP organizes multiple criteria in a hierarchical structure and uses pairwise comparisons between alternatives to calculate relative ratios (Saaty, 1990). AHP works with sets of n *options* and m *criteria*. In our project $n = 29$ and $m = 9$ since there are 29 options (software products) and 9 criteria (qualities). We rank the software for each of the qualities, and then we combine the quality rankings into an overall ranking based on the relative priorities between qualities.

The first step for ranking the software choices for a given quality involves a pairwise comparison between each of the n software options for that quality. AHP expresses the comparison through an $n \times n$ matrix A. When comparing option i and option j, the value of A_{ij} is decided as follows, with the value of A_{ji} generally equal to $1/A_{ij}$ (Saaty, 1990): $A_{ij} = 1$ if criterion i and criterion j are equally important, while $A_{ij} = 9$ if criterion i is extremely more important than criterion j. The natural numbers between 1 and 9 are used to show the different levels of relative importance between these two extremes. The above assumes that option i is of equal, or more, importance compared to option j ($i \geq j$). If that is not the case, we reverse i and j and determine A_{ji} first, then $A_{ij} = 1/A_{ji}$.

Section 3.3 shows how we measure the software via a grading template. For the AHP process, the relevant measure is the subjective score from 1 to 10 for each quality for each package. To turn these subjective measures x_{sub} and y_{sub} into Saaty's pair-wise scores for option x versus option y, respectively, we use the following calculation:

$$\begin{cases} \min\{9, x_{\text{sub}} - y_{\text{sub}} + 1\} & x_{\text{sub}} \geq y_{\text{sub}} \\ 1/\min\{9, y_{\text{sub}} - x_{\text{sub}} + 1\} & x_{\text{sub}} < y_{\text{sub}} \end{cases}$$

For example, we measured the usability for 3D Slicer and Ginkgo CADx as 8 and 7, respectively; therefore, on the 9-point scale, 3D Slicer compared to Ginkgo CADx is 2 and Ginkgo CADx versus 3D Slicer is 1/2, as shown in the sample AHP calculations (Table 1).

The second step is to calculate the priority vector w from A. The vector w ranks the software options by how well they achieve the given quality. The priority vector can be calculated by solving the equation (Saaty, 1990):

$$Aw = \lambda_{\max} w, \tag{1}$$

where $\lambda_{\max}$ is the maximal eigenvalue of A. In this project, w is approximated with the classic *mean of normalized values* approach (Ishizaka and Lusti, 2006):

$$w_i = \frac{1}{n} \sum_{j=1}^{n} \frac{A_{ij}}{\sum_{k=1}^{n} A_{kj}} \tag{2}$$

Table 1 summarizes the above two steps for the quality of installability. The matrix A is shown in the first set of columns, then the normalized version of A and finally the average of the normalized values to form the vector w in the last column.

We repeat the first and second steps for each of the qualities. The third step combines the quality rankings into an overall ranking. Following AHP, we need to first prioritize the qualities. The AHP method finds the priority of quality i (p_i) in the same way that the score (w_j) was found for software package j evaluated for a given quality (as shown above). That is, we conduct a pairwise comparison between the priority of different qualities to construct the $m \times m$ matrix A, and then we take the mean of normalized values for row i to find the priority value p_i for quality i. If we introduce the notation that w_j^i is the score for quality i for package j, then the overall score S_j for package j is found via:

$$S_j = \sum_{i=1}^{m} w_j^i p_i$$

	A_{ij}					$A_{ij}/\sum_{k=1}^{n} A_{kj}$					
	3D Slicer	Ginkgo	XMedCon	⋯	Gwyddion	3D Slicer	Ginkgo	XMedCon	⋯	Gwyddion	AVG
3D Slicer	1	2	4	⋯	2	0.071	0.078	0.060	⋯	0.078	0.068
Ginkgo	1/2	1	3	⋯	1	0.036	0.039	0.045	⋯	0.039	0.041
XMedCon	1/4	1/3	1	⋯	1/3	0.018	0.013	0.015	⋯	0.013	0.015
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋱	⋮	⋮
Gwyddion	1/2	1	3	⋯	1	0.036	0.039	0.045	⋯	0.039	0.041
SUM =	14.01	25.58	66.75	⋯	25.58	1.000	1.000	1.000	⋯	1.000	1.000

Table 1: Sample AHP Calculations for the Quality of Usability

3 Methodology

We developed a methodology for evaluating the state of the practice of research software (Smith and Michalski, 2022; Smith et al., 2021). The methodology can be instantiated for a specific domain of scientific software, which in the current case is medical imaging software for visualization. Our methodology involves and engages a domain expert partner throughout, as discussed in Section 3.1. The four main steps of the methodology are:

1. Identify list of representative software packages (Section 3.2);
2. Measure (or grade) the selected software (Section 3.3);
3. Interview developers (Section 3.4);
4. Answer the research questions (as given in Section 1.1).

In the sections below we provide additional detail on the above steps, while concurrently giving examples of how we applied the methodology to the MI domain.

3.1 Interaction With Domain Expert

The Domain Expert is an important member of the state of the practice assessment team. Pitfalls exist if non-experts attempt to acquire an authoritative list of software, or try to definitively rank the software. Non-experts have the problem that they can only rely on information available on-line, which has the following drawbacks: i) the on-line resources could have false or inaccurate information; and, ii) the on-line resources could leave out relevant information that is so in-grained with experts that nobody thinks to explicitly record it.

Domain experts may be recruited from academia or industry. The only requirements are knowledge of the domain and a willingness to be engaged in the assessment process. The Domain Expert does not have to be a software developer, but they should be a user of domain software. Given that the domain experts are likely to be busy people, the measurement process cannot put too much of a burden on their time. For the current assessment, our Domain Expert (and paper co-author) is Dr. Michael Noseworthy, Professor of Electrical and Computer Engineering at McMaster University, Co-Director of the McMaster School of Biomedical Engineering, and Director of Medical Imaging Physics and Engineering at St. Joseph's Healthcare, Hamilton, Ontario, Canada.

In advance of the first meeting with the Domain Expert, they are asked to create a list of top software packages in the domain. This is done to help the expert get in the right mind set in advance of the meeting. Moreover, by doing the exercise in advance, we avoid the potential pitfall of the expert approving the discovered list of software without giving it adequate thought.

The Domain Experts are asked to vet the collected data and analysis. In particular, they are asked to vet the proposed list of software packages and the AHP ranking. These interactions can be done either electronically or with in-person (or virtual) meetings.

3.2 List of Representative Software

We have a two-step process for selecting software packages: i) identify software candidates in the chosen domain; and, ii) filter the list to remove less relevant members (Smith et al., 2021).

We initially identified 48 MI candidate software projects from the literature (Björn, 2017; Brühschwein et al., 2019; Haak et al., 2015), on-line articles (Emms, 2019; Hasan, 2020; Mu, 2019), and forum discussions (Samala, 2014). The full list of 48 packages is available in Dong (2021). To reduce the length of the list to a manageable number (29 in this case, as given in Section **??**), we filtered the original list as follows:

1. We removed the packages that did not have source code available, such as *MicroDicom*, *Aliza*, and *jivex*.

2. We focused on the MI software that provides visualization functions, as described in Section 1.2. Furthermore, we removed seven packages that were toolkits or libraries, such as *VTK*, *ITK*, and *dcm4che*. We removed another three that were for PACS.

3. We removed *Open Dicom Viewer*, since it has not received any updates in a long time (since 2011).

The Domain Expert provided a list of his top 12 software packages. We compared his list to our list of 29. We found 6 packages were on both lists: *3D Slicer*, *Horos*, *ImageJ*, *Fiji*, *MRIcron* (we actually use the update version *MRIcroGL*) and *Mango* (we actually use the web version *Papaya*). Six software packages (*AFNI*, *FSL*, *Freesurfer*, *Tarquin*, *Diffusion Toolkit*, and *MRItrix*) were on the Domain Expert list, but not on our filtered list. However, when we examined those packages, we found they were out of scope, since their primary function was not visualization. The Domain Expert agreed with our final choice of 29 packages.

3.3 Grading Software

We grade the selected software using the measurement template summarized in Smith et al. (2021). The template provides measures of the qualities listed in Section 2.2, except for reproducibility, which is assessed through the developer interviews (Section 3.4). For each software package, we fill in the template questions. To stay within the target of 160 person hours to measure the domain, we allocated between one and four hours for each package. Project developers can be contacted for help regarding installation, if necessary, but we impose a cap of about two hours on the installation process, to keep the overall measurement time feasible. Figure 2 shows an excerpt of the spreadsheet. The spreadsheet includes a column for each measured software package.

Summary Information						
Software name?	3D Slicer	Ginkgo CADx	XMedCon	Weasis	ImageJ	DicomBrowser
Number of developers	100	3	2	8	18	3
Initial release date?	1998	2010	2000	2010	1997	2012
Last commit date?	02-08-2020	21-05-2019	03-08-2020	06-08-2020	16-08-2020	27-08-2020
Status?	alive	alive	alive	alive	alive	alive
License?	BSD	GNU LGPL	GNU LGPL	EPL 2.0	OSS	BSD
Software Category?	public	public	public	public	public	public
Development model?	open source	open source	open source	open source	open source	open source
Num pubs on the software?	22500	51	185	188	339000	unknown
Programming language(s)?	C++, Python, C	C++, C	C	Java	Java, Shell, Perl	Java, Shell
...	...	...	...	...	...	...
Installability						
Installation instructions?	yes	no	yes	no	yes	no
Instructions in one place?	no	n/a	no	n/a	yes	n/a
Linear instructions?	yes	n/a	yes	n/a	yes	n/a
Installation automated?	yes	yes	yes	yes	no	yes
messages?	n/a	n/a	n/a	n/a	n/a	n/a
Number of steps to install?	3	6	5	2	1	4
Numbe extra packages?	0	0	0	0	1	0
Package versions listed?	n/a	n/a	n/a	n/a	yes	n/a
Problems with uninstall?	no	no	no	no	no	no
...	...	...	...	...	...	...
Overall impression (1..10)?	10	8	8	7	6	7
...	...	...	...	...	...	...
Correctness/Verifiability						
...	...	...	...	...	...	...

Figure 2: Grading template example

The full template consists of 108 questions categorized under 9 qualities. We designed the questions to be unambiguous, quantifiable, and measurable with limited time and domain knowledge. We group the measures under headings for each quality, and one for summary information. The summary information (shown in Figure 2) is the first section of the template. This section summarizes general information, such as the software name, purpose, platform, programming language, publications about the software, the first release and the most recent change date, website, source code repository of the product, number of developers, etc. We follow the definitions given by Gewaltig and Cannon (2012) for the software categories. Public means software intended for public use. Private means software aimed only at a specific group, while the concept category is for

software written simply to demonstrate algorithms or concepts. The three categories of development models are (open source, free-ware and commercial) are discussed in Section 2.1. Information in the summary section sets the context for the project, but it does not directly affect the grading scores.

For measuring each quality, we ask several questions and the typical answers are among the collection of "yes", "no", "n/a", "unclear", a number, a string, a date, a set of strings, etc. The grader assigns each quality an overall score, between 1 and 10, based on all the previous questions. Several of the qualities use the word "surface". This is to highlight that, for these qualities in particular, the best that we can do is a shallow measure. For instance, we are not currently doing any experiments to measure usability. Instead, we are looking for an indication that the developers considered usability. We do this by looking for cues in the documentation, like a getting started manual, a user manual and a statement of expected user characteristics. Below is a summary of how we assess adoption of best practices by measuring each quality.

- **Installability** We assess the following: i) existence and quality of installation instructions; ii) the quality of the user experience via the ease of following instructions, number of steps, automation tools; and, iii) whether there is a means to verify the installation. If any problem interrupts the process of installation or uninstallation, we give a lower score. We also record the Operating System (OS) used for the installation test.

- **Correctness & Verifiability** We check each project to identify any techniques used to ensure this quality, such as literate programming, automated testing, symbolic execution, model checking, unit tests, etc. We also examine whether the projects use Continuous Integration and Continuous Delivery (CI/CD). For verifiability, we go through the documents of the projects to check for the presence of requirements specifications, theory manuals, and getting started tutorials. If a getting started tutorial exists and provides expected results, we follow it to check the correctness of the output.

- **Surface Reliability** We check the following: i) whether the software breaks during installation; ii) the operation of the software following the getting started tutorial (if present); iii) whether the error messages are descriptive; and, iv) whether we can recover the process after an error.

- **Surface Robustness** We check how the software handles unexpected/unanticipated input. For example, we prepare broken image files for MI software packages that load image files. We use a text file (.txt) with a modified extension name (.dcm) as an unexpected/unanticipated input. We load a few correct input files to ensure the function is working correctly before testing the unexpected/unanticipated ones.

- **Surface Usability** We examine the project's documentation, checking for the presence of getting started tutorials and/or a user manual. We also check whether users have channels to request support, such as an e-mail address, or issue tracker. Our impressions of usability are based on our interaction with the software during testing. In general, an easy-to-use graphical user interface will score high.

- **Maintainability** We believe that the artifacts of a project, including source code, documents, and building scripts, significantly influence its maintainability. Thus, we check each project for the presence of such artifacts as API documentation, bug tracker information, release notes, test cases, and build scripts. We also check for the use of tools supporting issue tracking and version control, the percentages of closed issues, and the proportion of comment lines in the code.

- **Reusability** We count the total number of code files for each project. Projects with numerous components potentially provide more choices for reuse. Furthermore, well-modularized code, which tends to have smaller parts in separate files, is typically easier to reuse. Thus, we assume that projects with more code files and fewer Lines of Code (LOC) per file are more reusable. We also consider projects with API documentation as delivering better reusability.

- **Surface Understandability** Given that time is a constraint, we cannot look at all code files for each project; therefore, we randomly examine 10 code files for their understandability. We check the code's style within each file, such as whether the identifiers, parameters, indentation, and formatting are consistent, whether the constants (other than 0 and 1) are not hardcoded, and whether the code is modularized. We also check the descriptive information for the code, such as documents mentioning the coding standard, the comments in the code, and the descriptions or links for details on algorithms in the code.

- **Visibility/Transparency** To measure this quality, we check the existing documents to find whether the software development process and current status of a project are visible and transparent. We examine the development process, current status, development environment, and release notes for each project.

As part of filling in the measurement template, we use freeware tools to collect repository related data. GitStats (Gieniusz, 2019) is used to measure the number of binary files as well as the number of added and deleted lines in a repository. We also use this tool to measure the number of commits over different intervals of time. Sloc Cloc and Code (scc) (Boyter, 2021) is used to measure the number of text based files as well as the number of total, code, comment, and blank lines in a repository.

Both tools measure the number of text-based files in a git repository and lines of text in these files. Based on our experience, most text-based files in a repository contain programming source code, and developers use them to compile and build software products. A minority of these files are instructions and other documents. So we roughly regard the lines of text in text-based files as lines of programming code. The two tools usually generate similar but not identical results. From our understanding, this minor difference is due to the different techniques to detect if a file is text-based or binary.

For projects on GitHub we manually collect additional information, such as the numbers of stars, forks, people watching this repository, open pull requests, closed pull requests, and the number of months a repository has been on GitHub. We need to take care with the project creation date, since a repository can have a creation date much earlier than the first day on GitHub. For example, the developers created the git repository for *3D Slicer* in 2002, but did not upload a copy of it to

GitHub until 2020. Some GitHub data can be found using its GitHub Application Program Interface (API) via the following url: *https://api.github.com/repos/[owner]/[repository]* where [owner] and [repository] are replaced by the repo specific values. The number of months a repository has been on GitHub helps us understand the average change of metrics over time, like the average new stars per month.

The repository measures help us in many ways. Firstly, they help us get a fast and accurate project overview. For example, the number of commits over the last 12 months shows how active a project has been, and the number of stars and forks may reveal its popularity (used to assess RQ3). Secondly, the results may affect our decisions regarding the grading scores for some software qualities. For example, if the percentage of comment lines is low, we double-check the understandability of the code; if the ratio of open versus closed pull requests is high, we pay more attention to maintainability.

As in Smith et al. (2016a), Virtual machines (VMs) were used to provide an optimal testing environment for each package. We used VMs because it is easier to start with a fresh environment, without having to worry about existing libraries and conflicts. Moreover, when the tests are complete the VM can be deleted, without any impact on the host operating system. The most significant advantage of using VMs is to level the playing field. Every software install starts from a clean slate, which removes "works-on-my-computer" errors. When filling in the measurement template, the grader notes the details for each VM, including hypervisor and operating system version.

When grading the software, we found 27 out of the 29 packages are compatible with two or three different OSes, such as Windows, macOS, and Linux, and 5 of them are browser-based, making them platform-independent. However, in the interest of time, we only performed the measurements for each project by installing it on one of the platforms. When it was an option, we selected Windows as the host OS.

3.4 Interview Methods

The repository-based measurements summarize the information we can collect from on-line resources. This information is incomplete because it doesn't generally capture the development process, the developer pain points, the perceived threats to software quality, and the developers' strategies to address these threats. Therefore, part of our methodology involves interviewing developers.

We based our interviews on a list of 20 questions, which can be found in Smith et al. (2021). Some questions are about the background of the software, the development teams, the interviewees, and how they organize their projects. We also ask about the developer's understanding of the users. Some questions focus on the current and past difficulties, and the solutions the team has found, or plan to try. We also discuss documentation, both with respect to how it is currently done, and how it is perceived. A few questions are about specific software qualities, such as maintainability, understandability, usability, and reproducibility. The interviews are semi-structured based on the question list; we ask follow-up questions when necessary. The interview process presented here was approved by the McMaster University Research Ethics Board under the application number MREB#: 5219.

We sent interview requests to all 29 projects using contact information from projects websites, code repository, publications, and from biographic pages at the teams' institutions. In the end nine developers from eight of the projects agreed to participate: *3D Slicer*, *INVESALIUS 3*, *dwv*, *BioImage Suite Web*, *ITK-SNAP*, *MRIcroGL*, *Weasis*, and *OHIF*. We spent about 90 minutes for each interview. One participant was too busy to have an interview, so they wrote down their answers. In one case two developers from the same project agreed to be interviewed. We held the meetings on-line using either Zoom or Teams, which facilitated recording and automatic transcription. The full interview answers can be found in Dong (2021).

4 Comparison of Tool Usage Between MI and Other Research Software

Developers use software tools to support the development, verification, maintenance, and evolution of software, software processes, and artifacts (Ghezzi et al., 2003, p. 501). MI software uses tools for CI/CD, user support, version control, documentation, and project management. To answer RQ5 we summarize the tool usage in these categories, and compare this to the usage by the research software community.

Table 2 summarizes the user support models by the number of projects using each model (projects may use more than one support model). We do not know whether the prevalent use of GitHub issues for user support is by design, or whether this just naturally happens as users seek help. The common use of GitHub by MI developers is not surprising, given that GitHub is the largest code host in the world, with over 128 million public repositories and over 23 million users (as of roughly February 26, 2020) (Kashyap, 2020).

User Support Model	Num. Projects
GitHub issue	24
Frequently Asked Questions (FAQ)	12
Forum	10
E-mail address	9
GitLab issue, SourceForge discussions	2
Troubleshooting	2
Contact form	1

Table 2: User support models by number of projects

From Section **??**, 27 of the 29 projects used git as the version control tool, one used Mercurial and one used Subversion. The hosting is on GitHub for 24 packages, SourceForge for three and BitBucket for two. Although teams may have a process for accepting new contributions, no one discussed this during their interviews. However, most teams (eight of nine) mentioned using GitHub and pull requests to manage contributions from the community. The interviewees generally gave very positive feedback on using GitHub. Some teams previously used a different approach to version control and eventually transferred to git and GitHub. The past approaches included contributions

from e-mail (three teams), contributions from forums (one team) and e-mailing the git repository back and forth between developers (one team).

The common use of version control for MI software illustrates considerable improvement from the poor adoption of version control tools that Wilson lamented in 2006 (Wilson, 2006). The proliferation of version control tools for MI matches the increase in the broader research software community. A little over 10 years ago Nguyen-Hoan et al. (2010) estimated that only 50% of research software projects use version control, but even at that time Nguyen-Hoan et al. (2010) noted an increase from previous usage levels. A survey in 2018 shows 81% of developers use a version control system (AlNoamany and Borghi, 2018). Smith (2018) has similar results, showing version control usage for alive projects in mesh generation, geographic information systems and statistical software for psychiatry increasing from 75%, 89% and 17% (respectively) to 100%, 95% and 100% (respectively) over a four-year period ending in 2018. (For completeness the same study showed a small decrease in version control usage for seismology software over the same time period, from 41% down to 36%). A recent survey by Carver et al. (2022) shows version control use among practitioners at over 95%, with 83/87 survey respondents indicating that they use it. All but one of the software guides cited in Section **??** includes the advice to use version control. (The USGS guide (USGS, 2019) was the only set of recommendations to not mention version control.) The high usage of version control tools in MI software matches the trend in research software in general.

As mentioned in Section **??**, we identified five projects using CI/CD tools (about 17% of the assessed projects). We found which projects used CI/CD by examining the documentation and source code of all projects. The count of CI/CD usage may actually be higher, since traces of CI/CD usage may not always appear in a repository. This was the case for a study of LBM software, where interviews with developers showed that more projects used CI/CD than was evident from repository artifacts alone (Michalski, 2021). The 17% utilization for MI software contrasts with the high frequency with which research software development guidelines recommend continuous integration (Brett et al., 2021; Brown, 2015; Thiel, 2020; van Gompel et al., 2016; Zadka, 2018). Although there is currently little data available on CI/CD utilization for research software, our impression is that CI/CD is not yet common practice, despite its recommendation. This is certainly the case for LBM software, where usage numbers are similar to MI software, with only 12.5% of 24 LBM packages showing evidence of CI/CD in their repositories (Michalski, 2021). The survey of Carver et al. (2022) suggests higher use of CI/CD with 54% (54/100) of respondents indicating that they use it. However, that survey measures something different from the current one by surveying practitioners, rather than assessing projects. Additional information on CI/CD is given in the recommendations (Section 7.1).

For documentation tools and methods mentioned by the interviewees, the most popular (mentioned by about 30% of developers) were forum discussions and videos. The second most popular options (mentioned by about 20% of developers) were GitHub, wiki pages, workshops, and social media. The least frequently mentioned options (about 10% of developers) included writing books, and google forms. In contrasting MI software with LBM software, the most significant documentation tool difference is that LBM software often uses document generation tools, like doxygen and sphinx (Michalski, 2021), while MI does not appear to use these tools.

Some interviewees mentioned the project management tools they used. Generally speaking, the interviewees talked about two types of tools: i) trackers, including GitHub, issue trackers, bug

trackers and Jira; and, ii) documentation tools, including GitHub, Wiki page, Google Doc, and Confluence. Of the specifically named tools in the above lists, interviewees mentioned GitHub 3 times, and each of the other tools once each.

Based on information provided by Jung et al. (2022), tool utilization for MI software has much in common with tool utilization for ocean modelling software. Both use tools for editing, compiling, code management, testing, building, and project management. From the data available, ocean modelling differs from MI software in its use of Kanban boards for project management.

5 Comparison of Principles, Process, and Methodologies to Research Software in General

We answer research question RQ6 by comparing the principles, processes, and methodologies used for MI software to what can be gleaned from the literature on research software in general. In our interviews with developers the responses about development model were vague, with only two interviewees following a definite development model. In some cases the interviewees felt their process was similar to an existing development model. Three teams (about 38%) either followed agile, or something similar to agile. Two teams (25%) either followed a waterfall process, or something similar. Three teams (about 38%) explicitly stated that their process was undefined or self-directed.

Our observations of an informally defined process, with elements of agile methods, matches what has been observed for research software in general. Scientific developers naturally use an agile philosophy (Ackroyd et al., 2008; Carver et al., 2007; Easterbrook and Johns, 2009; Heaton and Carver, 2015; Segal, 2005), or an amethododical process (Kelly, 2013), or a knowledge acquisition driven process (Kelly, 2015). A waterfall-like process can work for research software (Smith, 2016), especially if the developers work iteratively and incrementally, but externally document their work as if they followed a rationale design process (Parnas and Clements, 1986).

No interviewee introduced any strictly defined project management process. The most common approach was following the issues, such as bugs and feature requests. Additionally, the *3D Slicer* team had weekly meetings to discuss the goals for the project; the *INVESALIUS 3* team relied on the GitHub process for their project management; the *ITK-SNAP* team had a fixed six-month release pace; only the interviewee from the *OHIF* team mentioned that the team has a project manager; the *3D Slicer* team and *BioImage Suite Web* team do nightly builds and tests. The *OHIF* developer believes that a better project management process can improve junior developer efficiency while also improving internal and external communication.

We identified the use of unit testing in less than half of the 29 projects. On the other hand, the interviewees believed that testing (including usability tests with users) was the top solution to improve correctness, usability, and reproducibility. This level of testing matches what was observed for LBM software (Michalski, 2021) and is apparently greater than the level of testing for ocean modelling software. Jung et al. (2022) reports that ocean modellers underemphasize testing.

As the observed artifacts in Table **??** show, none of the 29 projects emphasize documentation. None of them had theory manuals, although we did identify a road map in the *3D Slicer* project. We did not find requirements specifications. Table 3 summarizes interviewees' opinions on docu-

mentation. Interviewees from each of the eight projects thought that documentation was essential to their projects, and most of them said that it could save their time to answer questions from users and developers. Most of them saw the need to improve their documentation, and only three of them thought that their documentations conveyed information clearly enough. Nearly half of developers also believed that the lack of time prevented them from improving documentation.

Opinion on Documentation	Num. Ans.
Documentation is vital to the project	8
Documentation of the project needs improvements	7
Referring to documentation saves time to answer questions	6
Lack of time to maintain good documentation	4
Documentation of the project conveys information clearly	3
Coding is more fun than documentation	2
Users help each other by referring to documentation	1

Table 3: Opinions on documentation by the numbers of interviewees with the answers

As Table **??** suggests, an emphasis on documentation, especially for new developers, is echoed in research software guidelines. Multiple guidelines recommend a document explaining how to contribute to a project, often named CONTRIBUTING. Guidelines also recommend tutorials, user guides and quick start examples. Smith et al. (2018a) suggests including instructions specifically for on-boarding new developers. For open-source software in general (not just research software), Fogel (2005) recommends providing tutorial style examples, developer guidelines, demos, and screenshots.

6 Developer Pain Points

Based on interviews with nine developers (described in Section 3.4), we answer three research questions (first mentioned in Section 1.1): RQ7) What are the pain points for developers working on research software projects?; RQ8) How do the pain points of developers from MI compare to the pain points for research software in general?; and RQ9) For MI developers what specific best practices are taken to address the pain points and software quality concerns?

Our interviews identified pain points related to a lack of time and funding, technology hurdles, improving correctness, and improving usability. In this section, we go through each pain point and contrast the MI experience with observations from other domains. We also cover potential ways to address the pain points, as promoted by the community. (Later, in Section 7, we propose additional pain mitigation strategies based on our experience.) In addition to pain points, we summarize MI developer strategies for improving maintainability and reproducibility. Although the interviewees did not explicitly identify these two qualities as pain points, we did discuss threats to these qualities and ways to improve them as part of our interview process (Smith et al., 2021). The interviewee's practices for addressing pain points and improving quality can potentially be emulated by other MI developers. Moreover, these practices may provide examples that can be followed by other research software domains.

Pinto et al. (2018) lists some pain points that did not come up in our conversations with MI developers: interruptions while coding, scope bloat, lack of user feedback, hard to collaborate on software projects, and aloneness. Wiese et al. (2019) also mention two research software pain points that did not explicitly arise in our interviews: reproducibility, and software scope determination. To the list of pain points not discussed for MI, our study of LBM software (Smith et al., 2024) adds lack of software experience for the developers, technical debt, and documentation. We did not observe any pain points for MI that were not also observed for LBM. From the pain points mentioned above, although the topics of reproducibility and technical debt did not come up in our MI interviews, we covered these two topics as part of the discussion of software qualities, as summarized at the end of this section. Although previous studies show pain points that were not mentioned by MI developers, we cannot conclude that these pain points are not relevant for MI software development, since we only interviewed nine developers for about an hour each.

P1: **Lack of Development Time:** Many interviewees thought lack of time, along with lack of funding (discussed next), were their most significant obstacles. Other domains of research software also experience the lack of time pain point (Pinto et al., 2016, 2018; Wiese et al., 2019). Our study of LBM software (Smith et al., 2024) also highlighted lack of time as a significant pain point.

Potential and proven solutions suggested by the interviewees include:

- Shifting from development to maintenance when the team does not have enough developers for building new features and fixing bugs at the same time;
- Improving documentation to save time answering users' and developers' questions;
- Supporting third-party plugins and extensions; and,
- Using GitHub Actions for CI/CD (Continuous Integration and Continuous Delivery.)

P2: **Lack of Funding:** Developers felt the pain of having to attract funding to develop and maintain their software. For instance, the interviewees from *3D Slicer* and *OHIF* said getting funding for software maintenance is more challenging than finding funding for research. The interviewee from *ITK-SNAP* thought more funding was a way to solve the lack of time problem, because they could hire more dedicated developers. On the other hand, the interviewee from *Weasis* did not feel that funding could solve the same problem, since they would still need time to supervise the project.

Funding challenges have also been noted by others (Gewaltig and Cannon, 2012; Goble, 2014; Katerbow and Feulner, 2018; Smith et al., 2024). Researchers that devote time to software have the additional challenge that funding agencies do not always count software when they are judging the academic excellence of the applicant. Wiese et al. (2019) reported developer pains related to publicity, since publishing norms have historically made it difficult to get credit for creating software. As studied by Howison and Bullard (2016), research software (specifically biology software, but the trend likely applies to other research software domains) is infrequently cited. Pinto et al. (2018) also mentions the lack of formal reward system for research software.

An interviewee proposed an idea for increasing funding: Licensing the software to commercial companies to integrate it into their products.

P3: **Technology Hurdles:** The technology hurdles mentioned by MI developers include: hard to keep up with changes in OS and libraries, difficult to transfer to new technologies, hard to support multiple OSes, and hard to support lower-end computers. Developers expressed difficulty balancing between four factors: cross-platform compatibility, convenience to development and maintenance, performance, and security.

The pain point survey of Wiese et al. (2019) highlights that technology hurdles are an issue for research software in general. Some technical-related problems mentioned by Wiese et al. (2019) include dependency management, cross-platform compatibility (also mentioned by Pinto et al. (2018)), CI, hardware issues and operating system issues. From (Smith et al., 2024) technology pain points for LBM developers include setting up parallelization and CI.

The solutions proposed by the MI developers include the following:

- Adopting a web-based approach with backend servers, to better support lower-end computers;
- Using memory-mapped files to consume less computer memory, to better support lower-end computers;
- Using computing power from the computers GPU for web applications;
- Maintaining better documentations to ease the development and maintenance processes;
- Improving performance via more powerful computers, which one interviewee pointed out has already happened.

As the above list shows, developers perceive that web-based applications will address the technology hurdle. Table 4 shows the teams' choices between native application and web application. Most of the 29 teams (24 of 29, or 83%) chose to develop native applications. For the eight teams we interviewed, three of them were building web applications, and the *MRIcroGL* team was considering a web-based solution.

The advantage for native applications is higher performance, while web applications have the advantage of cross-platform compatibility and a simpler build process. These web advantages mirror the native disadvantages of difficulty with cross-platform compatibility and a complex build process. The lower performance disadvantage of web applications can be improved with a server backend, but in this case there are disadvantages for privacy protection and server costs. These issues are discussed further in the recommendations (Section 7.2).

P4: **Ensuring Correctness:** Interviewees identified multiple threats to correctness. The most frequently mentioned threat was complexity. Complexity enters the software by various means, including the large variety of data formats, complicated data standards, differing outputs between medical imaging machines, and the addition of (non-viewing related) functionality. Other threats to correctness identified include the following:

Software Team	Native Application	Web Application
3D Slicer	✓	
INVESALIUS 3	✓	
dwv		✓
BioImage Suite Web		✓
ITK-SNAP	✓	
MRIcroGL	✓	
Weasis	✓	
OHIF		✓
Total number among the eight teams	5	3
Total number among the 29 teams	24	5

Table 4: Teams' choices between native application and web application

- Lack of real world image data for testing, in part because of patient privacy concerns (Wiese et al. (2019) mentions that the pain point of privacy concerns also arises for research software in general);
- Tests are expensive and time-consuming because of the need for huge datasets;
- Software releases are difficult to manage;
- No systematic unit testing; and,
- No dedicated quality assurance team.

As implied by the above threats to correctness, testing was the most often mentioned strategy for MI developers for ensuring correctness. Seven teams mentioned test related activities, including test-driven development, component tests, integration tests, smoke tests, regression tests, self tests and automated tests. With the common emphasis on testing to improve correctness, MI software is ahead of some other scientific domains. For scientific software in general Pinto et al. (2018) mention the problem of insufficient testing and Hannay et al. (2009b) show that more developers think testing is important than the number that believe they have a sufficient understanding of testing concepts. Our study of LBM software suggests that this domain shares the challenges of insufficient testing and insufficient understanding of testing concepts (Smith et al., 2024). Automated testing is a specific challenge for LBM software since free testing services do not offer adequate facilities for large amounts of data (Smith et al., 2024). Although not specifically mentioned during our interviews, the large data sets for MI likely also cause a challenge for using free testing services, like GitHub Actions.

Research software in general often struggles with the oracle problem for testing because for many potential test cases the developer doesn't have a means to judge the correctness of their calculated solutions (Hannay et al., 2009b; Kanewala and Bieman, 2013; Kelly et al., 2011; Wiese et al., 2019). The MI developers did not allude to this challenge, likely because for a give image (test case) it is possible to determine, potentially by using other software, the expected analysis results.

A frequently cited strategy for building confidence in correctness (mentioned by 3 interviewees) is a two state development process with stable releases and nightly builds. Other strategies for ensuring correctness that came up during the interviews include CI/CD, using de-identified copies of medical images for debugging, sending beta versions to medical workers who can access the data to do the tests, and collecting/maintaining a dataset of problematic images. Some additional strategies used by MI developers include:

- Using open datasets.
- If (part of) the team belongs to a medical school or a hospital, using the datasets they can access;
- If the team has access to MRI scanners, self-building sample images for testing;
- If the team has connections with MI equipment manufacturers, asking for their help on data format problems;

The feedback from the interviewees makes it clear that increased connections between the development team and medical professionals/institutions could ease the pain of ensuring correctness via testing.

P5: **Usability:**

The discussion with the developers focused on usability issues for two classes of users: the end users and other developers. The threats to usability for end users include an unintuitive user interface, inadequate feedback from the interface (such as lack of a progress bar), users being unable to determine the purpose of the software, not all users knowing if the software includes certain features, not all users understanding how to use the command line tool, and not all users understanding that the software is a web application. For developers the threats to usability include not being able to find clear instructions on how to deploy the software, and the architecture being difficult for new developers to understand.

At least to some extent the problems for MI software users are due to holes in their background knowledge. The survey of Wiese et al. (2019) for research software in general also mentioned that users do not always have the expertise required to install or use the software. Smith et al. (2024) observes a similar pattern for LBM software, with several LBM developers noting that users sometimes try to use incorrect method combinations. Furthermore, some LBM users think that the packages will work out of the box to solve their cases, while in reality computational fluid dynamics knowledge needs to be applied to correctly modify the packages for a new endeavour.

To improve the usability of MI software, the most common strategies mentioned by developers are as follows:

- Use documentation (user manuals, mailing lists, forums) (mentioned by 4 developers)
- Usability tests and interviews with end users; and, (mentioned by 3 developers)
- Adjusting the software according to user feedback. (mentioned by 3 developers)

Other suggested and practiced strategies include a graphical user interface, testing every release with active users, making simple things simple and complicated things possible, focusing on limited number of functions, icons with clear visual expressions, designing the software to be intuitive, having a UX (User eXperience) designer, dialog windows for important notifications, providing an example for users to follow, downsampling images to consume less memory, and providing an option to load only part of the data to boost performance. The last two points recognize that an important component of usability is performance, since poor performance frustrates users.

Up to this point, we have covered the pain points that came up in interviews with MI developers, along with a summary of the techniques that are currently used to address these pain points. Although the developers did not explicitly identify the qualities of maintainability and reproducibility as pain points in our interviews, as part of our interview questions (Section 3.4) they did share their approaches for improving these qualities, as discussed below.

Q1: **Maintainability:** Nguyen-Hoan et al. (2010) rate maintainability as the third most important software quality for research software in general. The push for sustainable software (de Souza et al., 2019) is motivated by the pain that past developers have had with accumulating too much technical debt (Kruchten et al., 2012). For LBM software, Smith et al. (2024) identifies technical debt as one of the developer pain points.

To improve maintainability, the most popular (with five out of nine interviewees mentioning it) strategy is to use a modular approach, with often repeated functions in a library. Other strategies that were mentioned for improving maintainability include supporting third-party extensions, an easy-to-understand architecture, a dedicated architect, starting from simple solutions, and documentation. The *3D Slicer* team used a well-defined structure for the software, which they named as an "event-driven MVC pattern". Moreover, *3D Slicer* discovers and loads necessary modules at runtime, according to the configuration and installed extensions. The *BioImage Suite Web* team had designed and re-designed their software multiple times in the last 10+ years. They found that their modular approach effectively supports maintainability (Joshi et al., 2011).

Q2: **Reproducibility:** Although the MI developers did not mention reproducibility explicitly as a pain point, they did mention the need to improve documentation. Good documentation does not just address the pain points of lack of developer time (P1), technology hurdles (P3), usability P5, and maintainability. Documentation is also necessary for reproducibility. The challenges of inadequate documentation are a known problem for research software (Pinto et al., 2018; Wiese et al., 2019) and for non-research software (Lethbridge et al., 2003).

In our interviews, we discussed threats to reproducibility and strategies for improving it. The threats that were mentioned include closed-source software, no user interaction tests, no unit tests, the need to change versions of some common libraries, variability between CPUs, and misinterpretation of how manufacturers create medical images.

The most commonly cited (by 6 teams) strategy to improve reproducibility was testing (regression tests, unit tests, having good tests). The second most common strategy (mentioned

by 5 teams) is making code, data, and documentation available, possibly by creating open-source libraries. Other ideas that were mentioned include running the same tests on all platforms, a dockerized version of the software to insulate it from the OS environment, using standard libraries, monitoring the upgrades of the library dependencies, clearly documenting the version information, bringing along the exact versions of all the dependencies with the software, providing checksums of the data, and benchmarking the software against other software that overlaps in functionality. Specifically one interviewee suggested using *3D Slicer* as the benchmark to test their reproducibility.

7 Recommendations

In this section we provide recommendations to address the pain points from Section 6 to answer RQ10. Our recommendations are not lists of criticisms for what should have been done in the past, or what should be done now; they are suggestions for consideration in the future. We expand on some of the ideas that came out of our interviews with developers (Section 6), including continuous integration, moving to web applications, and enriching the test data sets. We also bring in new ideas from our experience like employing linters, peer review, design for change and assurance cases. Our aim is to mention ideas that are at least somewhat beyond conventional best practices. The ideas listed here have the potential to become best practices in the medium to long-term. We list the ideas roughly in the order of increasing implementation effort.

7.1 Use Continuous Integration

Continuous integration involves frequent pushes to a code repository. With every push the software is built and tested (Humble and Farley, 2010, p. 13), (Fowler, 2006; Shahin et al., 2017). CI can take significant time and effort to set up and integrate into a team's workflow, but the benefits are significant, as follows:

- Elimination of headaches associated with a separate integration phase (Fowler, 2006), (Humble and Farley, 2010, p. 20). If developers postpone integration, integration problems are inevitable. Continuous integration means that problems are immediately obvious and the source of the problem can be isolated to the small increment that was just committed.
- Detection and removal of bugs (Fowler, 2006) via automated testing. To improve productivity, defects are best discovered and fixed at the point where they are introduced (Humble and Farley, 2010, p. 23). Code is not the only source of errors; they are also found in the files and scripts related to configuration management (Humble and Farley, 2010, p. 18).
- Everyone is always working on a stable base, since the rejection of inadequate commits means that the main branch will always be working. A stable base will always pass all tests. If the CI system uses generators and linters, it will also have current documentation and standard compliant code. A stable base improves developer productivity, allowing them to focus on coding, testing, and documentation.

CI consists of the following elements:

- A version control system (Fowler, 2006). To be effective, all files should be under version control, not just code files. Anything that is needed to build, install and run the software should be under version control, including configuration files, build scripts, test harnesses, and operating system configuration files (Humble and Farley, 2010, p. 19). Fortunately for the MI, as shown in Section **??** all our measured projects use version control.

- A fully automated build system (Fowler, 2006). As Humble and Farley (2010, p. 5) point out, deploying software manually is an anti-pattern. For MI software, Table **??** shows 18 of 29 packages (62%) were observed to include build scripts. Projects without a build system will need to add one to pursue using CI.

- An automated test system (Fowler, 2006). Building quality software involves creating automated tests at the unit, component, and acceptance test level, and executing these tests whenever someone makes a change to the code, its configuration, the environment, or the software stack that it runs on (Humble and Farley, 2010, p. 83). As Table **??** shows, test cases are in the uncommon category for MI software artifacts, which means that some MI projects will need to increase their testing automation if they wish to pursue CI.

- An automated system for other tasks, such as code checking, documentation building and web-site updating. These other tasks are not essential to CI, but they can be incorporated to improve the quality of the code and the communication between developers and users. For instance, a static analysis (possibly via linters) of the code may find poor programming practice or lack of adherence to adopted coding standards.

- An integrated build system to pull everything together. Every time there is a check-in (for instance a pull request), the integration server automatically checks out the sources onto the integration machine, starts a build, runs tests, and informs the committer of the results.

To enable incorporation into a team's workflow, Humble and Farley (2010, p. 60) explain that the usual approach for CI is to keep the build and test process short. Since MI files are large, the tests run with every check-in may need to focus on simple code interface tests, saving large tests for less frequent execution. A more sophisticated option to address the bottleneck for merges is CIVET (Continuous Integration, Verification, Enhancement, and Testing), which solves this problem by intelligently pinning, cancelling, and if necessary, restarting jobs as merges occur (Slaughter et al., 2021). A more sophisticated process management system can also enforce rules for pull requests, like checking that a test specification includes the test's motivation, a test description, and a design description for all changes Slaughter et al. (2021).

Setting up a CI system has never been easier than it is today. A dedicated CI server (either physically or virtually) can be installed with tools such as Jenkins, Buildbot, Go, and Integrity. However, installation on your own server is often unnecessary since there are many hosted CI solutions, such as: Travis CI, GitHub Actions and CircleCI. All that is required to begin using a hosted CI is to select the service and then edit a few lines of a YAML configuration file in the project's root directory.

Shahin et al. (2017) highlights the following challenges for adopting CI: lack of awareness and transparency, lack of expertise and skills, coordination and collaboration challenges, more pressure and workload for team members, general resistance to change, scepticism and distrust on continuous practices. The most common reason given for not adopting CI is that developers are not familiar enough with CI (Hilton et al., 2016). Shahin et al. (2017) observes that these problems can be mitigated via improving testing activities, planning and documentation, promoting a team mindset, adopting new rules and policies, and decomposing development into smaller units.

Continuous integration and delivery helps with addressing several pain points. For instance, CI/CD helps reduce development time (P1) by removing the need for a time-consuming integration stage and by automating regression testing. Automated regression tests also help with ensuring correctness (P4) and the quality of reproducibility (Q2).

7.2 Move To Web Applications

Section 6 describes the pain point of technology hurdles (P3), which motivates considering the use of web applications. Here we give further advice to help with deciding whether to adopt a web application. The decision will be based on whether, on balance, the web application improves the four factors identified by developers: compatibility, maintainability, performance, and security. To enable decision-making, a team will need to prioritize between these factors, based on their objectives and experience. The suggestions are intended to provide ideas and avenues for exploration; a web application will not be the right fit for all projects and all teams.

- **Modern technologies may improve frontend performance.** Web applications with only a frontend usually perform worse than native applications. However, new technologies may ease this difference. For example, some JavaScript libraries can help the frontend harness the power of the computer's GPU and accelerate graphical computing. In addition, there are new frameworks helping developers with cross-platform compatibility. For example, the Flutter project enables support for web, mobile, and desktop OS with one codebase. Other options include Vue, Angular and React, and Elm.

- **Backend servers can potentially deliver high performance.** Web applications with backend servers may perform even better than native applications. If a team needs to support lower-end computers, it is good to use back-end servers for heavy computing tasks. For backend servers where traffic and latency is not an issue, options include Django, Laravel and Node.js. The advantage of Django is that it provides access to Python libraries. For backend servers where traffic and latency is an issue, Gin is an option.

- **Backend servers can have low costs.** Serverless solutions from major cloud service providers (like Amazon Web Services (AWS) and Google Cloud Platform) may be worth exploring. Serverless solutions still use a server, but the server provider only charges the team when they use the server. The solution is event-driven, and costs the team by the number of requests processed. Thus, serverless can be very cost-effective for less intensively used functions.

- **Web transmission may diminish security.** Transferring sensitive data on-line can be a problem for projects requiring high security. Regulations for some MI applications may forbid doing web transmissions. In this case, a web application with a backend may not be an option.

7.3 Enrich the Testing Datasets

As described in Section 6, ensuring correctness (P4) via testing can be problematic because of limited access to real-world medical imaging datasets. We build on the suggestions we heard from our interviewees as follows:

- **Build and maintain good connections to datasets.** A team can build connections with professionals working in the medical domain, who may have access to private datasets and can perform tests for the team. If a team has such professionals as internal members, the process can be simplified.
- **Collect and maintain datasets over time.** A team may face problems caused by various unique inputs over the years of software development. This data should be collected and maintained over time to form a good, comprehensive, dataset for testing.
- **Search for open data sources.** In general, there are many open MI datasets. For instance, there are Chest X-ray Datasets by National Institute of Health (Wang et al., 2017), Cancer Imaging Archive (Prior et al., 2017), MedPix by National Library of Medicine (Smirniotopoulos, 2014), and datasets for liver (Bilic et al., 2019) and brain (B. H. Menze et al., 2015) tumor segmentation benchmarks. A team developing MI software should be able to find more open datasets according to their needs.
- **Create sample data for testing.** If a team can access tools creating sample data, they may also self-build datasets for testing. For example, an MI software development team can use an MRI scanner to create images of objects, animals, and volunteers. The team can build the images based on specific testing requirements.
- **Remove privacy from sensitive data.** For data with sensitive information, a team can ask the data owner to remove such information or add noise to protect privacy. One example is using de-identified copies of medical images for testing.
- **Establish community collaboration in the domain.** During our interviews with developers in the MI domain, we heard many stories of asking for supports from other professionals or equipment manufacturers. However, we believe that broader collaboration between development teams can address this problem better. Some datasets are too sensitive to share, but if the community has some kind of "group discussion", teams can better express their needs, and professionals can better offer voluntary support for testing. Ultimately, the community can establish a nonprofit organization as a third party, which maintains large datasets, tests Open Source Software (OSS) in the domain, and protects privacy.

7.4 Employ Linters

A linter is a tool that statically analyzes code to find programming errors, suspicious constructs, and stylistic inconsistencies (Wikipedia, 2022). Linters can be used as an ad hoc check for code files, but they really come into their own when used as part of a CI system, as discussed in Section 7.1. Almost none of the research software guidelines that we consulted, summarized in Section **??**, mention linters. The one exception is Thiel (2020). Despite the lack of mention in the guidelines, we believe that linters have the potential to improve code quality at a relatively low cost.

Linters have the following benefits: finding potential bugs, finding memory leaks, improving performance, standardizing code with respect to formatting, removing silly errors before code reviews, and catching potential security issues (SourceLevel, 2022). Most popular programming languages have an accompanying linter. For example, Python has the options of PyLint, flake8 and Black (Zadka, 2018).

We recommend the use of linters because they are relatively easy to incorporate into a developer's workflow, and they address several MI pain points (Section 6). For instance, linters address the lack of development time (P1) by increasing the developer's productive time via guarding against making frustrating, time-consuming, mundane mistakes. Moreover, since a linter can include rules that capture the wisdom of senior programmers, it can help newer developers avoid common mistakes. With respect to the technology hurdle pain point (P3), linters can assist with the move toward web applications (Section 7.2). For instance, ESLint in React is a pluggable linter that lets the developer know if they have imported something and not used it, if a function could be short-handed, if there are indentation inconsistencies, etc. (Whitehouse, 2018). By insisting on code standardization linters can reduce technical debt and thus improve maintainability (Q1). Although linters are tools for code analysis, the idea of statically checking for adherence to basic rules can be extended to check documentation. Smith et al. (2018d) shows how the use of tools to enforce documentation standards partially explains the relatively higher quality of statistical tools that are part of the Comprehensive R Archive Network (CRAN).

7.5 Conduct a Mix of Rigorous and Informal Peer Reviews

We advocate incorporating peer review into the development process, as frequently recommended for research software (Givler, 2020; Heroux et al., 2008; Orviz et al., 2017; USGS, 2019). In most cases a modern, lightweight review, should be adequate. Modern code review is informal, tool-based, asynchronous, and focused on reviewing code changes (Sadowski et al., 2018). Managing a project via GitHub pull requests is an example of a modern approach to reviewing code. Software development organizations have moved to this lightweight style of code review because of the inefficiencies of rigorous inspections (Rigby and Bird, 2013). However, for important parts of the code, developers may benefit from mixing in a more rigorous approach.

Fagan (1976) began work on rigorous review via code inspection. Elements of a typical inspection include reviewing the code against a checklist (checking the consistency of variable names, look for terminating loops, etc.), performing specific review tasks (such as summarizing the code's purpose, cross-referencing the code to the technical manual, creating a data dictionary for a given module, etc.) Rigorous inspection finds 60-65% of latent defects on average, and often tops 85%

in defect removal efficiency (Jones, 2008). The success rate of code inspection is generally higher than most forms of testing, which average between 30 — 35% for defect removal efficiency (Ebert and Jones, 2009; Jones, 2008). For research software, (Kelly and Shepard, 2000) show a task based inspection approach can be effective. Task based inspection is an ideal fit with an issue tracking system, like GitHub. The review tasks can be issues, so that they can be easily assigned, monitored and recorded. Potential issues include assigning junior developers to test getting-started tutorial and installation instructions.

As indicated in Section 5 some MI projects use modern code review, via issue tracking and the use of GitHub. Those MI projects not incorporating modern code review would likely benefit by adopting it. Although a rigorous code inspection is likely not worth the required resources, for critical parts of the code, developers may want to adopt a more rigorous approach. For instance, developers may drop the modern trend of asynchronous review and instead occasionally use synchronous review to help uncover errors and disseminate best practices throughout the team. For instance, teams could periodically meet, either in-person or virtually, and have junior members walk through their code. In-person reviews will likely help realize the benefits of modern code review noticed by Bird and Bacchelli (2013): defect detection, knowledge transfer, increased team awareness, and creation of alternative solutions to problems.

Due to improving code quality and increasing knowledge transfer, peer review addresses the same pain points and qualities as linters (Section 7.4): P1, P3, and Q1. Peer review can potentially find misunderstandings in how the code implements the required theory, which will improve the software's correctness (P4). The benefits of peer review for addressing pain points can be increased by extending the review from just code, to also reviewing all software artifacts, including documentation, build scripts, test cases and the development process itself.

7.6 Design For Change

In our "state of the practice" assessment exercise for LBM software (Smith et al., 2024), we noticed that LBM developers implicitly used modularization based on the principle of design for change to improve maintainability (Q1). We recommend that MI developers use the same principle for their modularizations. Although the advice to modularize research software to handle complexity is common (Stewart et al., 2017; Storer, 2017; Wilson et al., 2014), specific guidelines on how to divide the software into modules is less prevalent. Not every decomposition is a good design for supporting change, as shown by Parnas (1972). For instance, a design with low cohesion and high coupling (Ghezzi et al., 2003, p. 48) will make change difficult. Especially in research software, where change is inevitable, designers need to produce a modularization that supports change. Jung et al. (2022) points out that ocean modelling software is currently feeling the pain of not emphasizing modularization in legacy code.

Specific examples of design for change for LBM software (Smith et al., 2024) include the following:

- pyLBM has decoupled geometries and models of their system using abstraction and modularization of the source code, to make it easy to add new features. The pyLBM design allows for independent changes to the geometry and the model. pyLBM also redeveloped data structures to ease future change.

- TCLB (Rokicki and Laniewski-Wollk, 2016) is designed to allow for the addition of some LBM features, but changes to major aspects of the system would be difficult. For example, "implementing a new model will be an easy contribution", but changes to the "Cartesian mesh ... will be a nightmare" (Smith et al., 2024). The design of TCLB highlights that not every conceivable change needs to be supported, only the likely changes.

As the LBM examples above illustrate, developers can accomplish design for change by first identifying likely changes, either implicitly or explicitly, and second by hiding each likely change behind a well-defined module interface. This approach mirrors the recommendations from Parnas (1972). Section **??** lists ideas for how to document the design, including the likely changes, so that they are more visible to others.

7.7 Assurance Case

To ensure correctness (P4 and to achieve the quality of reproducibility (Q2), we recommend considering the use of assurance cases. Rinehart et al. (2015) defines an assurance case as "[a] reasoned and compelling argument, supported by a body of evidence, that a system, service, or organization will operate as intended for a defined application in a defined environment." An assurance cases provide an organized and explicit argument that the software and its documentation achieves desired qualities, such as correctness and reproducibility. Although assurance cases have been successfully employed for safety critical systems (Rinehart et al., 2015), this technique is relatively new for research software (Smith, 2018; Smith et al., 2020b).

One way to present an assurance case is through the Goal Structuring Notation (GSN) (Spriggs, 2012), which make arguments clear, easy to read and, hence, easy to challenge. GSN starts with a Top Goal (Claim), like "Program X delivers correct outputs when used for its intended use/purpose in its intended environment." We then decompose this top goal into Sub-Goals, which themselves may be further decomposed. The purpose of the decomposition is to take the abstract higher level goals and bring them down to something concrete that can be proven. The decomposition ends with the terminal Sub-Goals that are supported by Solutions (Evidence). Typical evidence will consist of documents, expert reviews, test case results, peer review, etc. Within the GSN framework, there are also strategy blocks, which describe the rationale for decomposing a Goal or Sub-Goal into more detailed Sub-Goals. A common tool for creating, editing, and presenting, a GSN argument is Astah.

Smith et al. (2020b) shows the example of arguing for the correctness of the Analysis of Functional NeuroImages (AFNI) package 3dfim+ (Ward, 2000). 3dfim+ analyzes the activity of the brain by computing the correlation between an ideal signal and the measured brain signal for each voxel. The assurance case for the correctness of 3dfim+ has the top level decomposed into four sub-goals, as shown in Figure 3. This example follows the same pattern as used for medical devices (Wassyng et al., 2015). The first sub-goal (GR) argues for the quality of the documentation of the requirements. The second sub-goal (GD) says that the design complies with the requirements and the third proposes that the implementation also complies with the requirements. The fourth sub-goal (GA) claims that the inputs to 3dfim+ will satisfy the operational assumptions, since we need valid input to make an argument for the correctness of the output.

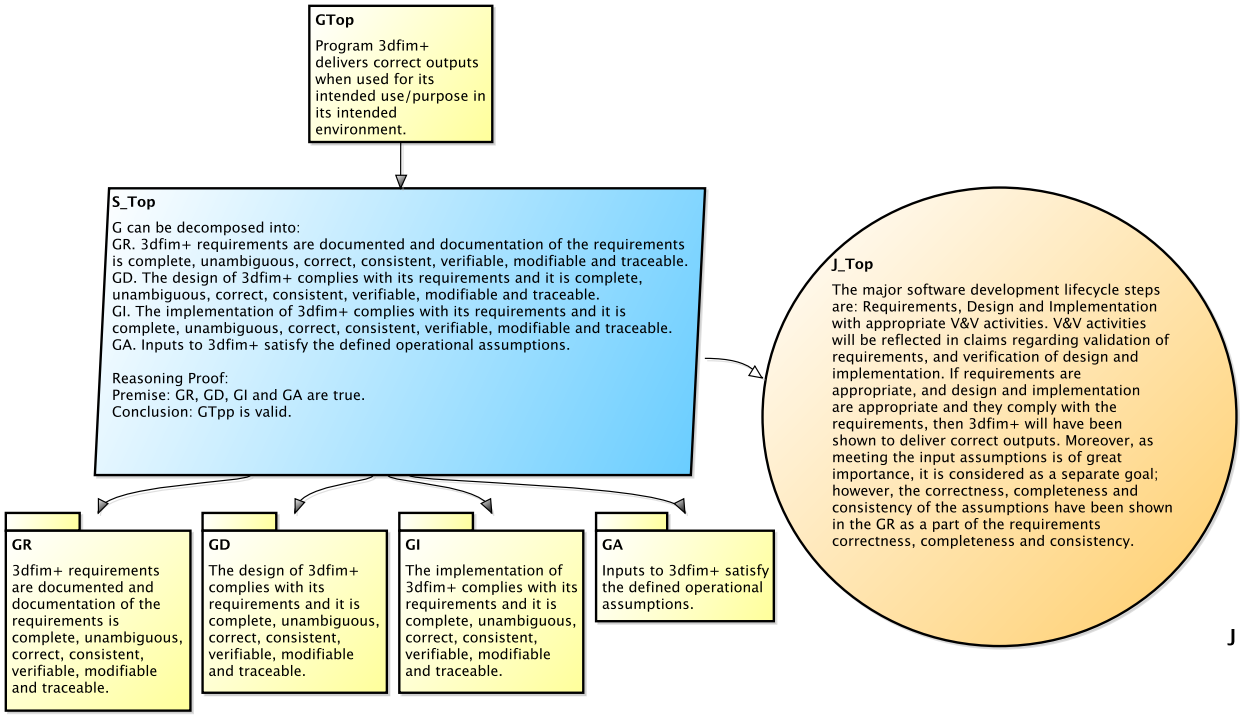

Figure 3: Top Goal of the assurance case and its sub-goals

Preparing an assurance case for the pre-existing 3dfim+ software shows the value of an assurance cases for research software. Although Smith et al. (2020a) found no errors in the output of the existing software, the rigour of the proposed approach did lead to finding ambiguities and omissions in the existing documentation, such as missing information on the coordinate system convention. In addition, a potential concern for the software itself was identified from the GA argument: running the software does not produce any warning about the obligation of the user to provide data that matches the parametric statistical model employed for the correlation calculations.

7.8 Generate All Things

To address developer pain points, we propose automatically generating MI code and its documentation via a "Generate All Things" (GAT) approach. A GAT approach uses models to capture knowledge in domains such as physics, computing, mathematics, documentation, and certification. Developers combine and transform knowledge via explicit "recipes", which weave it together to generate the desired code, documentation, test cases, inspection reports and build scripts. A recipe can even potentially be written to generate an assurance case (Section 7.7). Our definition of GAT implies generation of all software artifacts, not just the code. GAT moves development to a higher level of abstraction so that domain experts can work without concern for low-level implementation details. GAT allows developers to optimally generate code and documentation, reduce the likelihood of errors, eliminate redundancy, and automate maintenance. With a GAT approach

MI developers can experiment with different algorithm choices, different input formats, different outputs formats, etc.

Part of GAT is code generation. In the future, some believe that code generation will transform coding, documentation, design, and verification (Johanson and Hasselbring, 2018; Smith, 2018). GAT removes the distraction of writing software, allowing developers to focus on their science. A GAT approach removes the maintenance headaches of documentation duplicates and near duplicates (Luciv et al., 2018), since developers capture knowledge once and transform it as needed. Code generation has previously been applied to improve research software, such as linear algebra software packages like Blitz++ (Veldhuizen, 1998), and ATLAS (Automatically Tuned Linear Algebra Software) (Whaley et al., 2001). Software/hardware generation has been applied for digital signal processing in Spiral (Püschel et al., 2001). A generative approach has also been used for a family of efficient, type-safe Gaussian elimination algorithms Carette and Kiselyov (2011). (Logg et al., 2012) use code generation when solving partial differential equations in FEniCS (Finite Element and Computational Software). Matkerim et al. (2013) and Ober et al. (2018) use code generation for High Performance Computing (HPC), using UML (Unified Modelling Language) for their domain models. Szymczak et al. (2016) presents initial work on the GAT approach, Smith and Carette (2021) presents a motivating example, and (Carette et al., 2021) provides a prototype.

A GAT approach addresses multiple pain points. For example, a generative approach can decrease development time (P1) by automation, once the necessary infrastructure is in place. The GAT approach addresses the technology related pain point (P3) because technology information can be captured in the models and transformed as needed. To ensure correctness (P4), a GAT approach should be correct by construction. If there are mistakes, GAT has the advantage that they are propagated throughout the generated artifacts, which greatly increases the chance that someone will notice the mistake. Maintainability (Q1) is addressed because developers write the recipes used for generation at a high level making them relatively easy to change. Usability (P5) is addressed because of the emphasis on generating up-to-date documentation. GAT facilitates reproducibility (Q2) because at any time all the code and documentation can be regenerated. The generator can include explicit traceability to show the dependence of the software on specific versions of software libraries.

8 Threats to Validity

Below we categorize and list the threats to validity that we have identified. Our categories come from an analysis of software engineering secondary studies by Ampatzoglou et al. (2019), where a secondary study analyzes the data from a set of primary studies. Ampatzoglou et al. (2019) is appropriate because a common example of a secondary study is a systematic literature review. Our methodology is a systematic software review — the primary studies are the software packages, and our work collects and analyzes these primary studies. We identified similar threats to validity in our assessment of the state of the practice of Lattice Boltzmann Solvers (Smith et al., 2024).

8.1 Reliability

A study is reliable if repetition of the study by different researchers using the original study's methodology would lead to the same results (Runeson and Höst, 2009). Reliability means that data and analysis are independent of the specific researcher(s) doing the study. For the current study the identified reliability related threats are as follows:

- One individual does the manual measures for all packages. A different evaluator might find different results, due to differences in abilities, experiences, and biases.
- The manual measurements for the full set of packages took several months. Over this time the software repositories may have changed and the reviewer's judgement may have drifted.

In Smith et al. (2016a) we reduced concern over the reliability risk associated with the reviewer's judgement by demonstrating that the measurement process is reasonably reproducible. In Smith et al. (2016a) we graded five software products by two reviewers. Their rankings were almost identical. As long as each grader uses consistent definitions, the relative comparisons in the AHP results will be consistent between graders.

8.2 Construct Validity

Runeson and Höst (2009) defines construct validity as the adopted metrics representing what they are intended to measure. Our construct threats are often related to how we assume our measurements influences the various software qualities, as summarized in Section 3.3. Specifically, our construct validity related threats include the following:

- We make indirect measurement of software qualities since meaningful direct measures for qualities like maintainability, reusability and verifiability, are unavailable. We follow the usual assumption that developers achieve higher quality by following procedures and adhering to standards (van Vliet, 2000, p. 112).
- As mentioned in Section **??**, we could not install or build *dwv*, *GATE*, and *DICOM Viewer*. We used a deployed on-line version for *dwv*, a VM version for *GATE*, but no alternative for *DICOM Viewer*. We might underestimate their rank due to these technical issues.
- Measuring software robustness only involved two pieces of data. This is likely part of the reason for limited variation in the robustness scores (Figure **??**). We could add more robustness data by pushing the software to deal with more unexpected situations, like a broken Internet connection, but this would require a larger investment of measurement time.
- We may have inaccurately estimated maintainability by assuming a higher ratio of comments to source code improves maintainability. Moreover, we assumed that maintainability is improved if a high percentage of issues are closed, but a project may have a wealth of open issues, and still be maintainable.

- We assess reusability by the number of code files and LOC per file. This measure is indicative of modularity, but it does not necessarily mean a good modularization. The modules may not be general enough to be easily reused, or the formatting may be poor, or the understandability of the code may be low.
- The understandability measure relies on 10 random source code files, but the 10 files will not necessarily be representative.
- As discussed in Section **??**, our overall AHP ranking makes the unrealistic assumption of equal weighting.
- We approximated popularity by stars and watches (Section **??**), but this assumption may not be valid.
- As mentioned in Section 3.4, one interviewee was too busy to participate in a full interview, so they provided written answers instead. Since we did not have the chance to explain our questions or ask them follow-up questions, there is a possibility of misinterpretation of the questions or answers.
- In building Table **??** some judgement was necessary on our part, since not all guidelines use the same names for artifacts that contain essentially the same information.

8.3 Internal Validity

Internal validity means that discovered causal relations are trustworthy and cannot be explained by other factors (Runeson and Höst, 2009). In our methodology the internal validity threats include the following:

- In our search for software packages (Section 3.2), we may have missed a relevant package.
- Our methodology assumes that all relevant software development activities will leave a trace in the repositories, but this is not necessarily true. For instance, the possibility exists that CI usage was higher than what we observed through the artifacts (Section 4). As another example, although we saw little evidence of requirements (Section 4), maybe teams keep this kind of information outside their repos, possibly in journal papers or technical reports.
- We interviewed a relatively small sample of 8 teams. Their pain points (Section 6) may not be representative of the rest of their community.

8.4 External Validity

If the results of a study can be generalized (applied) to other situations/cases, then the study is externally valid (Runeson and Höst, 2009). We are confident that our search was exhaustive. We do not believe that we missed any highly popular examples. Therefore, the bulk of our validity concerns are internal (Section 8.3). However, our hope is that the trends observed, and the lessons learned for MI software can be applied to other research software. With that in mind we identified the following threat to external validity:

- We cannot generalize our results if the development of MI software is fundamentally different from other research software.

Although there are differences, like the importance of data privacy for MI data, we found the approach to developing LBM software (Smith et al., 2024) and MI software to be similar. Except for the domain specific aspects, we believe that the trends observed in the current study are externally valid for other research software.

9 Future Work

The following recommendations for future state of the practice measurement exercises, for MI or for other domains, could address some threats to validity mentioned above. Moreover, some ideas may make the data collection more efficient.

- We would like to make surface measurements less shallow. For example:
  - Surface reliability: our current measurement relies on the processes of installation and getting started tutorials. However, not all software needs installation or has a getting started tutorial. We could devise a list of operation steps (with the help of the Domain Expert), perform the same operations with each software, and record any errors.
  - Surface robustness: we used damaged images as inputs for this measuring MI software. This process is similar to fuzz testing (Wikipedia contributors, 2021b), which is one type of fault injection (Wikipedia contributors, 2021a). We may adopt more fault injection methods, and identify tools and libraries to automate this process.
  - Surface usability: we can design usability tests and test all software projects with end-users. The end-users can be volunteers and domain experts. Ideas for getting started are available in Smith et al. (2021).
  - Surface understandability: our current method does not require understanding the source code. As software engineers, perhaps we can select a small module of each project, read the source code and documentation, try to understand the logic, and score the ease of the process.
  - Maintainability: we can add a measure modifiability as part of the measurement of maintainability. An experiment could be conducted asking participants to make modifications, observing the study subjects during the modifications, testing the resulting software and surveying the participants (Smith et al., 2021).
- We can further automate the measurements on the grading template. For example, with automation scripts and the GitHub API, we may save significant time on retrieving the GitHub metrics through a GitHub Metric Collector. This Collector can take GitHub repository links as input, automatically collect metrics from the GitHub API, and record the results.
- We can improve some interview questions. Some examples are:

  - In one question we ask, "Do you think improving this process can tackle the current problem?" The problem is that this is a yes-or-no question, which is not informative. We could change the question to "By improving this process, what current problems can be tackled?";
  - We can ask for more details about the modular approach, such as "What principles did you use to divide code into modules? Can you describe an example of using your principles?".

- We can better organize the interview questions. Since we use audio conversion tools to transcribe the answers, we should make the transcription easier to read. For example, we can order them together for questions about the five software qualities and compose a similar structure for each.

- We can mark the follow-up interview questions with keywords. For example, say "this is a follow-up question" every time asking one. Thus, we record this sentence in the transcription, and it will be much easier to distinguish the follow-up questions from the 20 designed questions.

10 Conclusions

We analyzed the state of the practice for the MI domain with the goal of understanding current practice, answering our ten research questions (Section 1.1) and providing recommendations for current and future projects. Our methods in Section 3 form a general process to evaluate domain-specific software, that we apply to the specific domain of MI software. We identified 48 MI software candidates, then, with the help of the Domain Expert selected 29 of them to our final list.

Section **??** lists our measurement results for ranking the 29 projects for nine software qualities. Our ranking results appear credible since they are mostly consistent with the ranking from the scientific community implied by the GitHub stars-per-year metric. As discussed in Section **??**, four of the top five software projects appear in both our list and in the GitHub popularity list. Moreover, our top five packages appear among the first eight positions on the GitHub list. The noteworthy discrepancies between the two lists are for the packages that we were unable to install (*dwv* and *Dicom Viewer*).

Based on our grading scores *3D Slicer*, *ImageJ*, *Fiji* and *OHIF Viewer* are the top four software performers. However, the separation between the top performers and the others is not extreme. Almost all packages do well on at least a few qualities, as shown in Table 5, which summarizes the packages ranked first and second for each quality. Almost 70% (20 of 29) of the software packages appear in the top two for at least two qualities. The only packages that do not appear in Table 5, or only appear once, are *Papaya*, *MatrixUser*, *MRIcroGL*, *XMedCon*, *dicompyler*, *DicomBrowser*, *AMIDE*, *3DimViewer*, and *Drishti*. The shortness of this list suggests parity with respect to adoption of best practices for MI software overall.

For insight into devising future methods and tools, we interviewed nine developers (from eight teams) to learn about their pain points (Section 6). We also discussed qualities of potential concern. The identified pain points and qualities of concern include:

Quality	Ranked 1st or 2nd
Installability	3D Slicer, BioImage Suite Web, Slice:Drop, INVESALIUS
Correctness and Verifiability	OHIF Viewer, 3D Slicer, ImageJ
Reliability	SMILI, ImageJ, Fiji, 3D Slicer, Slice:Drop, OHIF Viewer
Robustness	XMedCon, Weasis, SMILI, ParaView, OsiriX Lite, MicroView, medInria, ITK-SNAP, INVESALIUS, ImageJ, Horos, Gwyddion, Fiji, dicompyler, Dicom-Browser, BioImage Suite Web, AMIDE, 3DimViewer, 3D Slicer, OHIF Viewer, DICOM Viewer
Usability	3D Slicer, ImageJ, Fiji, OHIF Viewer, ParaView, INVESALIUS, Ginkgo CADx, SMILI, OsiriX Lite, BioImage Suite Web, ITK-SNAP, medInria, MicroView, Gwyddion
Maintainability	3D Slicer, Weasis, ImageJ, OHIF Viewer, ParaView
Reusability	3D Slicer, ImageJ, Fiji, OHIF Viewer, SMILI, dwv, BioImage Suite Web, GATE, ParaView
Understandability	3D Slicer, ImageJ, Weasis, Fiji, Horos, OsiriX Lite, dwv, Drishti, OHIF Viewer, GATE, ITK-SNAP, ParaView, INVESALIUS
Visibility and Transparency	ImageJ, 3D Slicer, Fiji
Overall Quality	3D Slicer, ImageJ

Table 5: Top performers for each quality (sorted by order of quality measurement)

P1 Lack of development time,

P2 Lack of funding,

P3 Technology hurdles,

P4 Ensuring correctness,

P5 Usability,

Q1 Quality of maintainability, and

Q2 Quality of reproducibility.

Despite the pain points, overall MI software is in a healthy state for software development practices. In our survey of the selected projects we observed 88% of the documentation artifacts recommended by research software development guidelines (Section **??**). With respect to tools, MI is keeping pace with other research software with 100% of the projects using version control (with 93% specifically using git) (Section 4). We observed that the MI developers tend to follow the typical research software trend of using a quasi-agile software development process (Section 5).

Although the state of the practice for MI software is healthy, we did notice areas where practice seems to lag behind the research software development guidelines. For instance, the guidelines recommend three artifacts that were not observed: uninstall instructions, test plans, and requirements documentation. We observed the following recommended artifacts, but only rarely: contributing file, developer code of conduct, code style guidelines, product roadmap, design documentation, and API documentation (Section **??**). Although software development tool use seems healthy, we found the use of CI/CD behind typical usage rates (17% of the projects used CI/CD) (Section 4). With respect to the development process, developer identified areas for improvement included testing (only 50% of projects were identified to have unit testing) and documentation (only three out of nine developers felt their documentation was clear enough) (Section 5).

Our interviewees proposed strategies to improve the state of the practice, to address the identified pain points, and to improve software quality. To their list (Section 6) we added some of our own recommended strategies (Section 7). Below we summarize the proposed strategies, with traceability to where we discuss the strategy, and to the relevant pain points.

1. Increase documentation to address P1, P3, P5, Q1, Q2 (Section 6)
2. Increase testing by enriching datasets to address P4, Q2 (Section 6, 7.3)
3. Increase modularity to address Q1 (Section 6)
4. Use continuous integration to address P1, P4, Q2 (Section 6, 7.1)
5. Move to web applications to address P3 (Section 6, 7.2)
6. Employ linters to address P1, P3, Q1 (Section 7.4)
7. Peer reviews to address P1, P3, P4, Q1 (Section 7.5)
8. Design for change to address Q1 (Section 7.6)
9. Assurance case to address P4, Q2 (Section 7.7)
10. Generate all things to address P1, P3, P4, P5, Q1 and Q2. (Section 7.8)

Acknowledgements

We would like to thank Peter Michalski and Oluwaseun Owojaiye for fruitful discussions on topics relevant to this paper. We would also like to thank Jason Balaci for advice on web applications.

Conflict of Interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. 2008. Scientific Software Development at a Research Facility. *IEEE Software* 25, 4 (July/August 2008), 44–51.

U.S. Food & Drug Administration. 2021. Medical Imaging. https://www.fda.gov/radiation-emitting-products/radiation-emitting-products-and-procedures/medical-imaging. [Online; accessed 25-July-2021].

Yasmin AlNoamany and John A. Borghi. 2018. Towards computational reproducibility: researcher perspectives on the use and sharing of software. *PeerJ Computer Science* 4, e163 (September 2018), 1–25.

Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, Marijn Verbeek, and Alexander Chatzigeorgiou. 2019. Identifying, Categorizing and Mitigating Threats to Validity in Software Engineering Secondary Studies. *Information and Software Technology* 106 (02 2019). https://doi.org/10.1016/j.infsof.2018.10.006

S. Angenent, Eric Pichon, and Allen Tannenbaum. 2006. Mathematical methods in medical image processing. *Bulletin (new series) of the American Mathematical Society* 43 (07 2006), 365–396. https://doi.org/10.1090/S0273-0979-06-01104-9

B. H. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer, K. Farahani, J. Kirby, Y. Burren, N. Porz, J. Slotboom, R. Wiest, L. Lanczi, E. Gerstner, M. -A. Weber, T. Arbel, B. B. Avants, N. Ayache, P. Buendia, D. L. Collins, N. Cordier, J. J. Corso, A. Criminisi, T. Das, H. Delingette, Ç. Demiralp, C. R. Durst, M. Dojat, S. Doyle, J. Festa, F. Forbes, E. Geremia, B. Glocker, P. Golland, X. Guo, A. Hamamci, K. M. Iftekharuddin, R. Jena, N. M. John, E. Konukoglu, D. Lashkari, J. A. Mariz, R. Meier, S. Pereira, D. Precup, S. J. Price, T. R. Raviv, S. M. S. Reza, M. Ryan, D. Sarikaya, L. Schwartz, H. -C. Shin, J. Shotton, C. A. Silva, N. Sousa, N. K. Subbanna, G. Szekely, T. J. Taylor, O. M. Thomas, N. J. Tustison, G. Unal, F. Vasseur, M. Wintermark, D. H. Ye, L. Zhao, B. Zhao, D. Zikic, M. Prastawa, M. Reyes, and K. Van Leemput. 2015. The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS). *IEEE Transactions on Medical Imaging* 34, 10 (Oct. 2015), 1993–2024.

Isaac N. Bankman. 2000. Preface. In *Handbook of Medical Imaging*, Isaac N. Bankman (Ed.). Academic Press, San Diego, xi – xii. https://doi.org/10.1016/B978-012077790-7/50001-1

F. Benureau and N. Rougier. 2017. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *ArXiv e-prints* (Aug. 2017). arXiv:1708.08205 [cs.GL]

Patrick Bilic, Patrick Ferdinand Christ, Eugene Vorontsov, Grzegorz Chlebus, Hao Chen, Qi Dou, Chi-Wing Fu, Xiao Han, Pheng-Ann Heng, Jürgen Hesser, Samuel Kadoury, Tomasz K. Konopczynski, Miao Le, Chunming Li, Xiaomeng Li, Jana Lipková, John S. Lowengrub, Hans Meine, Jan Hendrik Moltz, Chris Pal, Marie Piraud, Xiaojuan Qi, Jin Qi, Markus Rempfler, Karsten Roth, Andrea Schenk, Anjany Sekuboyina, Ping Zhou, Christian Hülsemeyer, Marcel Beetz, Florian Ettlinger, Felix Grün, Georgios Kaissis, Fabian Lohöfer, Rickmer Braren, Julian Holch, Felix Hofmann, Wieland H. Sommer, Volker Heinemann, Colin Jacobs, Gabriel Efrain Humpire Mamani, Bram van Ginneken, Gabriel Chartrand, An Tang, Michal Drozdzal, Avi Ben-Cohen, Eyal Klang, Michal Marianne Amitai, Eli Konen, Hayit Greenspan, Johan Moreau, Alexandre Hostettler, Luc Soler, Refael Vivanti, Adi Szeskin, Naama Lev-Cohain, Jacob Sosna, Leo Joskowicz, and Bjoern H. Menze. 2019. The Liver Tumor Segmentation Benchmark (LiTS). *CoRR* abs/1901.04056 (2019). arXiv:1901.04056 http://arxiv.org/abs/1901.04056

Christian Bird and Alberto Bacchelli. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the International Conference on Software Engineering* (proceedings of the international conference on software engineering ed.). IEEE. https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/

Kari Björn. 2017. Evaluation of Open Source Medical Imaging Software: A Case Study on Health Technology Student Learning Experience. *Procedia Computer Science* 121 (01 2017), 724–731. https://doi.org/10.1016/j.procs.2017.11.094

Barry W Boehm. 2007. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*. Vol. 69. John Wiley & Sons.

Ben Boyter. 2021. Sloc Cloc and Code. https://github.com/boyter/scc. [Online; accessed 27-May-2021].

Alys Brett, James Cook, Peter Fox, Ian Hinder, John Nonweiler, Richard Reeve, and Robert Turner. 2021. Scottish Covid-19 Response Consortium. https://github.com/ScottishCovidResponse/modelling-software-checklist/blob/main/software-checklist.md.

Titus Brown. 2015. Notes from"How to grow a sustainable software development process (for scientific software)". http://ivory.idyll.org/blog/2015-growing-sustainable-software-development-process.html.

Andreas Brühschwein, Julius Klever, Anne-Sophie Hoffmann, Denise Huber, Elisabeth Kaufmann, Sven Reese, and Andrea Meyer-Lindenberg. 2019. Free DICOM-Viewers for Veterinary Medicine: Survey and Comparison of Functionality and User-Friendliness of Medical Imaging PACS-DICOM-Viewer Freeware for Specific Use in Veterinary Medicine Practices. *Journal of Digital Imaging* (03 2019). https://doi.org/10.1007/s10278-019-00194-3

Jacques Carette and Oleg Kiselyov. 2011. Multi-stage programming with Functors and Monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76, 5 (2011), 349–375.

Jacques Carette, Spencer Smith, Jason Balaci, Anthony Hunt, Ting-Yu Wu, Samuel Crawford, Dong Chen, Dan Szymczak, Brooks MacLachlan, Dan Scime, and Maryyam Niazi. 2021. *Drasil*. https://github.com/JacquesCarette/Drasil/tree/v0.1-alpha

Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 2007. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 550–559. https://doi.org/10.1109/ICSE.2007.77

Jeffrey C. Carver, Nicholas Weber, Karthik Ram, Sandra Gesing, and Daniel S. Katz. 2022. A Survey of the State of the Practice for Research Software in the United States. *PeerJ Computer Science* 8:e963 (2022). https://doi.org/10.7717/peerj-cs.963

Robert Choplin, J Boehme, and C Maynard. 1992. Picture archiving and communication systems: an overview. *Radiographics : a review publication of the Radiological Society of North America, Inc* 12 (02 1992), 127–9. https://doi.org/10.1148/radiographics.12.1.1734458

James Edward Corbly. 2014. The Free Software Alternative: Freeware, Open Source Software, and Libraries. *Information Technology and Libraries* 33, 3 (Sep. 2014), 65–75. https://doi.org/10.6017/ital.v33i3.5105

Mario Rosado de Souza, Robert Haines, Markel Vigo, and Caroline Jay. 2019. What Makes Research Software Sustainable? An Interview Study With Research Software Engineers. *CoRR* abs/1903.06039 (2019). arXiv:1903.06039 http://arxiv.org/abs/1903.06039

Ao Dong. 2021. *Assessing the State of the Practice for Medical Imaging Software*. Master's thesis. McMaster University, Hamilton, ON, Canada.

Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the Software for Understanding Climate Change. *Comuting in Science & Engineering* 11, 6 (November/December 2009), 65–74. https://doi.org/10.1109/MCSE.2009.193

C. Ebert and C. Jones. 2009. Embedded Software: Facts, Figures, and Future. *Computer* 42, 4 (April 2009), 42–52. https://doi.org/10.1109/MC.2009.118

Steve Emms. 2019. 16 Best Free Linux Medical Imaging Software. https://www.linuxlinks.com/medicalimaging/. [Online; accessed 02-February-2020].

M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211. https://doi.org/10.1147/sj.153.0182

Karl Fogel. 2005. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc.

Martin Fowler. 2006. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html.

Marc-Oliver Gewaltig and Robert Cannon. 2012. Quality and sustainability of software tools in neuroscience. *Cornell University Library* (May 2012), 20 pp.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2003. *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall, Upper Saddle River, NJ, USA.

Tomasz Gieniusz. 2019. GitStats. https://github.com/tomgi/git_stats. [Online; accessed 27-May-2021].

Ray Givler. 2020. A Checklist of Basic Software Engineering Practices for Data Analysts and Data Scientists. https://www.linkedin.com/pulse/checklist-basic-software-engineering-practices-data-analysts-givler/?articleId=6681691007303630849.

GNU. 2019. Categories of free and nonfree software. https://www.gnu.org/philosophy/categories.html. [Online; accessed 20-May-2021].

Carole Goble. 2014. Better Software, Better Research. *IEEE Internet Computing* 18, 5 (2014), 4–8. https://doi.org/10.1109/MIC.2014.88

Daniel Haak, Charles-E Page, and Thomas Deserno. 2015. A Survey of DICOM Viewer Software to Integrate Clinical Research and Medical Imaging. *Journal of digital imaging* 29 (10 2015). https://doi.org/10.1007/s10278-015-9833-1

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009a. How do scientists develop and use scientific software?. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. 1–8. https://doi.org/10.1109/SECSE.2009.5069155

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009b. How Do Scientists Develop and Use Scientific Software?. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE '09)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/SECSE.2009.5069155

Mehedi Hasan. 2020. Top 25 Best Free Medical Imaging Software for Linux System. https://www.ubuntupit.com/top-25-best-free-medical-imaging-software-for-linux-system/. [Online; accessed 30-January-2020].

Dustin Heaton and Jeffrey C. Carver. 2015. Claims About the Use of Software Engineering Practices in Science. *Inf. Softw. Technol.* 67, C (Nov. 2015), 207–219. https://doi.org/10.1016/j.infsof.2015.07.011

Michael A. Heroux, James M. Bieman, and Robert T. Heaphy. 2008. Trilinos Developers Guide Part II: ASC Softwar Quality Engineering Practices Version 2.0. https://faculty.csbsju.edu/mheroux/fall2012_csci330/TrilinosDevGuide2.pdf.

Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 426–437.

James Howison and Julia Bullard. 2016. Software in the Scientific Literature: Problems with Seeing, Finding, and Using Software Mentioned in the Biology Literature. *J. Assoc. Inf. Sci. Technol.* 67, 9 (sep 2016), 2137–2155. https://doi.org/10.1002/asi.23538

Jez Humble and David G. Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, Upper Saddle River, NJ. http://my.safaribooksonline.com/9780321601919

IEEE. 1991. *IEEE Standard Glossary of Software Engineering Terminology*. Standard. IEEE.

Alessio Ishizaka and Markus Lusti. 2006. How to derive priorities in AHP: A comparative study. *Central European Journal of Operations Research* 14 (12 2006), 387–400. https://doi.org/10.1007/s10100-006-0012-9

ISO. 2001. Iec 9126-1: Software engineering-product quality-part 1: Quality model. *Geneva, Switzerland: International Organization for Standardization* 21 (2001).

ISO/IEC. 2011. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Standard. International Organization for Standardization.

ISO/TR. 2002. *Ergonomics of human-system interaction — Usability methods supporting human-centred design*. Standard. International Organization for Standardization.

ISO/TR. 2018. *Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts*. Standard. International Organization for Standardization.

Arne N. Johanson and Wilhelm Hasselbring. 2018. Software Engineering for Computational Science: Past, Present, Future. *Computing in Science & Engineering* Accepted (2018), 1–31.

Capers Jones. 2008. Measuring Defect Potentials and Defect Removal Efficiency. *Crosstalk, The Journal of Defense Software Engineering* 21, 6 (June 2008), 11–13.

Alark Joshi, Dustin Scheinost, Hirohito Okuda, Dominique Belhachemi, Isabella Murphy, Lawrence Staib, and Xenophon Papademetris. 2011. Unified Framework for Development, Deployment and Robust Testing of Neuroimaging Algorithms. *Neuroinformatics* 9 (03 2011), 69–84. https://doi.org/10.1007/s12021-010-9092-8

Reiner Jung, Sven Gundlach, and Wilhelm Hasselbring. 2022. Thematic Domain Analysis for Ocean Modeling. *Environmental Modelling & Software* (Jan 2022), 105323. https://doi.org/10.1016/j.envsoft.2022.105323

Panagiotis Kalagiakos. 2003. The Non-Technical Factors of Reusability. In *Proceedings of the 29th Conference on EUROMICRO*. IEEE Computer Society, 124.

U. Kanewala and J. M. Bieman. 2013. Techniques for testing scientific programs without an oracle. In *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*. 48–57. https://doi.org/10.1109/SECSE.2013.6615099

Neeraj Kashyap. 2020. GitHub's Path to 128M Public Repositories. https://towardsdatascience.com/githubs-path-to-128m-public-repositories-f6f656ab56b1.

Matthias Katerbow and Georg Feulner. 2018. Recommendations on the development, use and provision of Research Software. https://doi.org/10.5281/zenodo.1172988

Diane Kelly. 2013. Industrial Scientific Software: A Set of Interviews on Software Development. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research* (Ontario, Canada) *(CASCON '13)*. IBM Corp., Riverton, NJ, USA, 299–310. http://dl.acm.org/citation.cfm?id=2555523.2555555

Diane Kelly. 2015. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software* 109 (2015), 50–61. https://doi.org/10.1016/j.jss.2015.07.027

Diane Kelly and Terry Shepard. 2000. Task-directed software inspection technique: an experiment and case study. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research* (Mississauga, Ontario, Canada). IBM Press, 6. http://portal.acm.org/citation.cfm?id=782040#

Diane F. Kelly. 2007. A Software Chasm: Software Engineering and Scientific Computing. *IEEE Software* 24, 6 (2007), 120–119. https://doi.org/10.1109/MS.2007.155

Diane F. Kelly, W. Spencer Smith, and Nicholas Meng. 2011. Software Engineering for Scientists. *Computing in Science & Engineering* 13, 5 (Oct. 2011), 7–11.

Tae-Yun Kim, Jaebum Son, and Kwanggi Kim. 2011. The Recent Progress in Quantitative Medical Image Analysis for Computer Aided Diagnosis Systems. *Healthcare informatics research* 17 (09 2011), 143–9. https://doi.org/10.4258/hir.2011.17.3.143

Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 29, 6 (2012), 18–21.

Jörg Lenhard, Simon Harrer, and Guido Wirtz. 2013. Measuring the installability of service orchestrations using the square method. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE, 118–125.

T.C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. *IEEE Software* 20, 6 (2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

A. Logg, K.-A. Mardal, and G. N. Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer. https://doi.org/10.1007/978-3-642-23099-8

D. V. Luciv, D. V. Koznov, G. A. Chernishev, A. N. Terekhov, K. Yu. Romanovsky, and D. A. Grigoriev. 2018. Detecting Near Duplicates in Software Documentation. *Programming and Computer Software* 44, 5 (01 Sep 2018), 335–343. https://doi.org/10.1134/S0361768818050079

Bazargul Matkerim, Darhan Akhmed-Zaki, and Manuel Barata. 2013. Development High Performance Scientific Computing Application Using Model-Driven Architecture. *Applied Mathematical Sciences* 7, 100 (2013), 4961–4974.

Matthew McCormick, Xiaoxiao Liu, Julien Jomier, Charles Marion, and Luis Ibanez. 2014. ITK: Enabling Reproducible Research and Open Science. *Frontiers in neuroinformatics* 8 (02 2014), 13. https://doi.org/10.3389/fninf.2014.00013

Peter Michalski. 2021. *State of The Practice for Lattice Boltzmann Method Software.* Master's thesis. McMaster University, Hamilton, Ontario, Canada.

Hamza Mu. 2019. 20 Free & open source DICOM viewers for Windows. https://medevel.com/free-dicom-viewers-for-windows/. [Online; accessed 31-January-2020].

JD Musa, Anthony Iannino, and Kazuhira Okumoto. 1987. Software reliability: prediction and application.

Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. A Survey of Scientific Software Development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy) *(ESEM '10)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1852786.1852802

Ileana Ober, Marc Palyart, Jean-Michel Bruel, and David Lugato. 2018. On the use of models for high-performance scientific computing applications: an experience report. *Software & Systems Modeling* 17, 1 (01 Feb 2018), 319–342. https://doi.org/10.1007/s10270-016-0518-0

Pablo Orviz, Álvaro López García, Doina Cristina Duma, Giacinto Donvito, Mario David, and Jorge Gomes. 2017. A set of common software quality assurance baseline criteria for research projects. https://doi.org/10.20350/digitalCSIC/12543

David L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM* 15, 2 (Dec. 1972), 1053–1058.

David Lorge Parnas and Paul C Clements. 1986. A rational design process: How and why to fake it. *IEEE transactions on software engineering* 2 (1986), 251–257.

Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. 2016. More Common Than You Think: An In-depth Study of Casual Contributors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 112–123. https://doi.org/10.1109/SANER.2016.68

Gustavo Pinto, Igor Wiese, and Luis Felipe Dias. 2018. How Do Scientists Develop and Use Scientific Software? An External Replication. In *Proceedings of 25th IEEE International Conference on*

Software Analysis, Evolution and Reengineering. 582–591. https://doi.org/10.1109/SANER.2018.8330263

Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. 2011. A Survey of the Practice of Computational Science *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 19, 12 pages. https://doi.org/10.1145/2063348.2063374

F. Prior, Kirk Smith, Ashish Sharma, Justin Kirby, Lawrence Tarbox, Ken Clark, William Bennett, Tracy Nolan, and John Freymann. 2017. The public cancer radiology imaging collections of The Cancer Imaging Archive. *Scientific Data* 4 (09 2017), sdata2017124. https://doi.org/10.1038/sdata.2017.124

The Linux Information Project. 2006. Freeware Definition. http://www.linfo.org/freeware.html. [Online; accessed 20-May-2021].

Markus Püschel, Bryan Singer, Manuela Veloso, and José M. F. Moura. 2001. Fast Automatic Generation of DSP Algorithms. In *International Conference on Computational Science (ICCS) (Lecture Notes In Computer Science, Vol. 2073)*. Springer, 97–106.

Peter C. Rigby and Christian Bird. 2013. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 202–212. https://doi.org/10.1145/2491411.2491444

David J. Rinehart, John C. Knight, and Jonathan Rowanhill. 2015. *Current Practices in Constructing and Evaluating Assurance Cases with Applications to Aviation*. Technical Report CR-2014-218678. National Aeronautics and Space Administration (NASA), Langley Research Centre, Hampton, Virginia.

J. Rokicki and L. Laniewski-Wollk. 2016. Adjoint lattice Boltzmann for topology optimization on multi-GPU architecture. *Computers & Mathematics with Applications* 71, 3 (2016), 833–848.

Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (19 Dec 2009), 131–164. https://doi.org/10.1007/s10664-008-9102-8

Thomas L. Saaty. 1990. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research* 48, 1 (1990), 9–26. https://doi.org/10.1016/0377-2217(90)90057-I Desicion making by the analytic hierarchy process: Theory and applications.

Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 181–190.

Ravi Samala. 2014. Can anyone suggest free software for medical images segmentation and volume? https://www.researchgate.net/post/Can_anyone_suggest_free_software_for_medical_images_segmentation_and_volume. [Online; accessed 31-January-2020].

Will Schroeder, Bill Lorensen, and Ken Martin. 2006. *The visualization toolkit*. Kitware.

Judith Segal. 2005. When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering* 10, 4 (Oct. 2005), 517–536. https://doi.org/10.1007/s10664-005-3865-y

Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

Andrew Slaughter, Cody Permann, Jason Miller, Brian Alger, and Stephen Novascone. 2021. Continuous Integration, In-Code Documentation, and Automation for Nuclear Quality Assurance Conformance. *Nuclear Technology* 207 (01 2021), 1–8. https://doi.org/10.1080/00295450.2020.1826804

James Smirniotopoulos. 2014. MedPix Medical Image Database. https://doi.org/10.13140/2.1.3403.3608

Barry Smith, Roscoe Bartlett, and xSDK Developers. 2018a. xSDK Community Package Policies. https://doi.org/10.6084/m9.figshare.4495136.v6

Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. 2020a. Quality Definitions of Qualities. (2020). Manuscript in preparation.

Spencer Smith and Peter Michalski. 2022. Digging Deeper Into the State of the Practice for Domain Specific Research Software. In *Proceedings of the International Conference on Computational Science, ICCS*. 1–15.

Spencer Smith, Peter Michalski, Jacques Carette, and Zahra Keshavarz-Motamed. 2024. State of the Practice for Lattice Boltzmann Method Software. *Archives of Computational Methods in Engineering* 31, 1 (Jan 2024), 313–350. https://doi.org/10.1007/s11831-023-09981-2

Spencer Smith, Yue Sun, and Jacques Carette. 2018c. Statistical Software for Psychology: Comparing Development Practices Between CRAN and Other Communities. arXiv:1802.07362 [cs.SE]

Spencer Smith, Zheng Zeng, and Jacques Carette. 2018e. Seismology software: state of the practice. *Journal of Seismology* 22 (05 2018). https://doi.org/10.1007/s10950-018-9731-3

W. Spencer Smith. 2016. A Rational Document Driven Design Process for Scientific Computing Software. In *Software Engineering for Science*, Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal (Eds.). Taylor & Francis, Chapter Section I – Examples of the Application of Traditional Software Engineering Practices to Science, 33–63.

W. Spencer Smith. 2018. Beyond Software Carpentry. In *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*. 1–8.

W. Spencer Smith and Jacques Carette. 2021. Sustainable Software via Generation. In *Proceedings of the 1st Annual Booth Resource and Innovation Cluster (BRIC) Symposium*. 21.

W. Spencer Smith, Jacques Carette, Peter Michalski, Ao Dong, and Oluwaseun Owojaiye. 2021. Methodology for Assessing the State of the Practice for Domain X. https://arxiv.org/abs/2110.11575.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. 2016a. State of Practice for Mesh Generation Software. *Advances in Engineering Software* 100 (Oct. 2016), 53–71.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. 2018b. State of the Practice for GIS Software. arXiv:1802.03422 [cs.SE]

W Spencer Smith, D Adam Lazzarato, and Jacques Carette. 2016b. State of the practice for mesh generation and mesh processing software. *Advances in Engineering Software* 100 (2016), 53–71.

W. Spencer Smith, Mojdeh Sayari Nejad, and Alan Wassyng. 2020b. Raising the Bar: Assurance Cases for Scientific Computing Software. *Computing in Science and Engineering* 23, 1 (Feb. 2020), 47–57.

W. Spencer Smith, Yue Sun, and Jacques Carette. 2018d. Statistical Software for Psychology: Comparing Development Practices Between CRAN and Other Communities. https://arxiv.org/abs/1802.07362. 33 pp..

SourceLevel. 2022. What is a linter and why your team should use it? https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it.

John Spriggs. 2012. *GSN - The Goal Structuring Notation*. Springer-Verlag, London.

Graeme Stewart et al. 2017. A Roadmap for HEP Software and Computing R&D for the 2020s. *arXiv* (2017). arXiv:1712.06982 [physics.comp-ph]

Tim Storer. 2017. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Comput. Surv.* 50, 4, Article 47 (Aug. 2017), 32 pages. https://doi.org/10.1145/3084225

Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE)*. In conjunction with ICSE 2016, Austin, Texas, United States. 4 pp..

Carsten Thiel. 2020. EURISE Network Technical Reference. https://technical-reference.readthedocs.io/en/latest/.

USGS. 2019. USGS (United States Geological Survey) Software Plannning Checklist. https://www.usgs.gov/media/files/usgs-software-planning-checklist.

Omkarprasad S. Vaidya and Sushil Kumar. 2006. Analytic hierarchy process: An overview of applications. *European Journal of Operational Research* 169, 1 (2006), 1–29. https://doi.org/10.1016/j.ejor.2004.04.028

Maarten van Gompel, Jauco Noordzij, Reinier de Valk, and Andrea Scharnhorst. 2016. Guidelines for Software Quality, CLARIAH Task Force 54.100. https://github.com/CLARIAH/software-quality-guidelines/blob/master/softwareguidelines.pdf.

Hans van Vliet. 2000. *Software Engineering (2nd ed.): Principles and Practice*. John Wiley & Sons, Inc., New York, NY, USA.

Todd. L. Veldhuizen. 1998. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Lecture Notes in Computer Science*. Springer-Verlag.

Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald Summers. 2017. ChestX-ray8: Hospital-scale Chest X-ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases. *arXiv:1705.02315* (05 2017).

B. Douglas Ward. 2000. *Program 3dfim+*. Biophysics Research Institute, Medical College of Wisconsin. https://afni.nimh.nih.gov/afni/doc/manual/3dfim+.pdf

Alan Wassyng, Neeraj Kumar Singh, Mischa Geven, Nicholas Proscia, Hao Wang, Mark Lawford, and Tom Maibaum. 2015. Can Product-Specific Assurance Case Templates Be Used as Medical Device Standards? *IEEE Design & Test* 32, 5 (2015), 45–55. https://doi.org/10.1109/MDAT.2015.2462720

R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* 27, 1–2 (2001), 3–35.

Ross Whitehouse. 2018. Setting up ESLint in React. https://medium.com/@RossWhitehouse/setting-up-eslint-in-react-c20015ef35f7.

I. S. Wiese, I. Polato, and G. Pinto. 2019. Naming the Pain in Developing Scientific Software. *IEEE Software* (2019), 1–1. https://doi.org/10.1109/MS.2019.2899838

Wikipedia. 2022. Lint (software). https://en.wikipedia.org/wiki/Lint_(software).

Wikipedia contributors. 2021a. Fault injection — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Fault_injection&oldid=1039005082 [Online; accessed 28-August-2021].

Wikipedia contributors. 2021b. Fuzzing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=1039424308 [Online; accessed 28-August-2021].

Wikipedia contributors. 2021c. Medical image computing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Medical_image_computing&oldid=1034877594 [Online; accessed 25-July-2021].

Wikipedia contributors. 2021d. Medical imaging — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Medical_imaging&oldid=1034887445 [Online; accessed 25-July-2021].

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 2014. Best Practices for Scientific Computing. *PLoS Biol* 12, 1 (Jan. 2014), e1001745. https://doi.org/10.1371/journal.pbio.1001745

Gregory V. Wilson. 2006. Where's the Real Bottleneck in Scientific Computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist* 94, 1 (2006). http://www.americanscientist.org/issues/pub/wheres-the-real-bottleneck-in-scientific-computing

Moshe Zadka. 2018. How to open source your Python library. https://opensource.com/article/18/12/tips-open-sourcing-python-libraries.

Xiaofeng Zhang, Nadine Smith, and Andrew Webb. 2008. 1 - Medical Imaging. In *Biomedical Information Technology*, David Dagan Feng (Ed.). Academic Press, Burlington, 3–27. https://doi.org/10.1016/B978-012373583-6.50005-0