State of the Practice for Medical Imaging Software

Spencer Smith[a], Ao Dong[a], Jacques Carette[a], Mike Noseworthy[b]

[a]*McMaster University, Computing and Software Department, 1280 Main Street West, Hamilton, L8S 4K1, Ontario, Canada*
[b]*McMaster University, Electrical Engineering, 1280 Main Street West, Hamilton, L8S 4K1, Ontario, Canada*

Abstract

We present the state of the practice for Medical Imaging 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 top three software products were *3D Slicer*, *ImageJ*, and *OHIF Viewer*. By interviewing the developers, we identified three major pain points: i) lack of resources; ii) difficulty balancing between compatibility, maintainability, performance, and security; and, iii) lack of access to real-world datasets for testing. For future MI software projects, we propose adopting test-driven development, using continuous integration and continuous delivery (CI/CD), using git and GitHub, maintaining good documentation, supporting third-party plugins or extensions, considering web application solutions, and improving collaboration between different MI software projects. [Update after the paper has been revised. —SS]

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). An example medical image of the brain is provided in Figure 1. 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.

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 scientific programming (Kelly, 2007; Storer, 2017). Although scientists spend a substantial proportion of their working hours on software development (Hannay et al., 2009; Prabhu et al., 2011), many developers learn software engineering skills by themselves or from their peers, instead of from proper training (Hannay et al., 2009). Hannay et al. (2009) 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 et al., 2021). The questions are designed to learn about the community's interest in, and experience with, 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

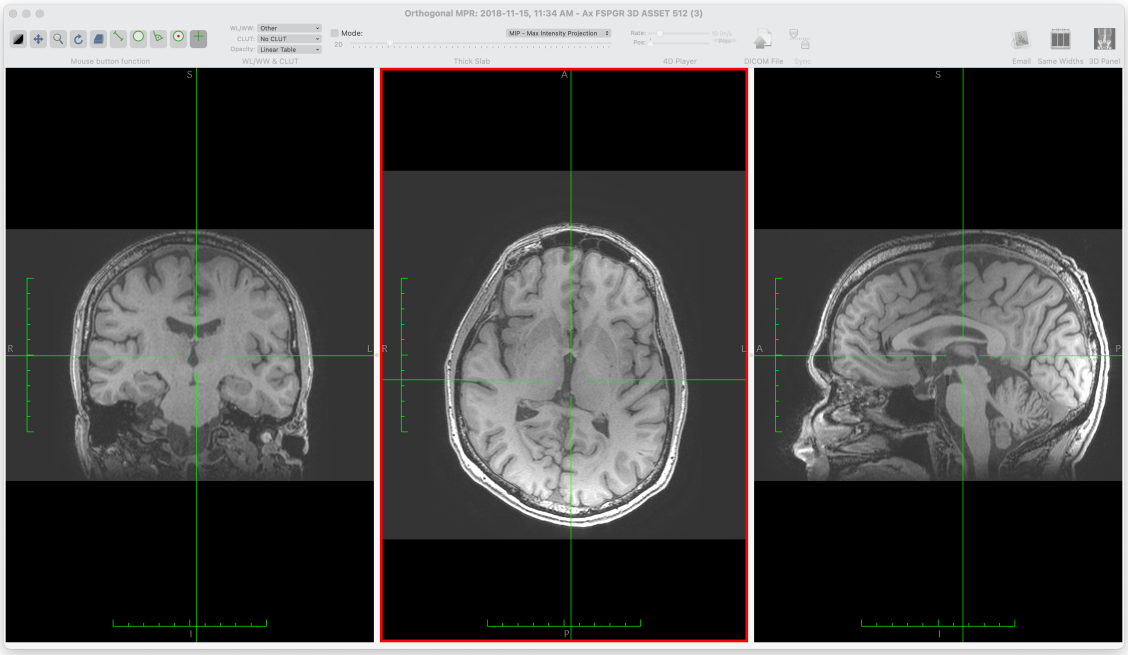

Figure 1: Example brain image showing a multi-planar reformat using Horos (free open-source medical imaging/DICOM viewer for OSX, based on OsiriX)

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.

The questions are used to structure the discussion in the paper, 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:

RQ1: What MI software projects exist, with the constraint that the source code must be available for all identified projects? (Section 4)

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 4)

RQ3: How similar is the list of top projects identified in RQ2 to the most popular projects, as viewed by the scientific community? (Section 5)

RQ4: How do MI projects compare to research software in general with respect to the artifacts present in their repositories? (Section 6)

RQ5: How do MI projects compare to research software in general with respect to the use of tools (Section 7) 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 8)

RQ7: What are the pain points for developers working on MI software projects? (Section 9)

RQ8: How do the pain points of developers from MI compare to the pain points for research software in general? (Section 9)

RQ9: For MI developers what specific best practices are taken to address the pain points and software quality concerns? (Section 10)

RQ10: What research software development practice could potentially address the pain point concerns identified in RQ7). (Section 12)

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 SC software (Smith et al., 2021). Details can be found in Section 3. Our methodology has been applied to MI software (Dong, 2021a) and Lattice Boltzmann Solvers (Michalski, 2021). 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., 2018d), 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. A person month is considered to be 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 an SC 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) (Details are found in Section 2). After this, as another addition to our previous methodology, we interview some of the 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 what license the software is distributed under. 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, it brings much more convenience to researchers analyzing the qualities of the 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, the users are required to pay to access all of the features of commercial software, 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 qualities are considered 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 the assumptions in its requirements specification are violated (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) (ISO/TR, 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 (IEEE, 1991), (Boehm, 2007).

- **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 of 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)

Thomas L. 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 factors in a hierarchical structure and uses pairwise comparisons between alternatives to calculate relative ratios (Saaty, 1990). We use AHP to generate a ranking for a set of software packages.

For a project with m criteria, we can use an $m \times m$ matrix A to record the relative importance between factors. When comparing criterion i and criterion 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 criterion i is not less important than criterion j. If that is not the case, we reverse i and j and determine A_{ji} first, then $A_{ij} = 1/A_{ji}$.

The priority vector w, which ranks the criteria by their importance, 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):

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

If there are n alternatives, for criterion $i = 1, 2, ..., m$, we can create an $n \times n$ matrix B_i to record the relative preferences between these choices for each of the m criterion. The way of generating B_i is similar to the one for A. However, rather than comparing the importance between criteria, we pairwise decide how much we favour one alternative over the other. We use the same method to calculate the local priority vector for each B_i. The local priority vector in this case ranks the n alternatives for criterion i.

In this project, the first nine software qualities mentioned in Section 2.2 are the criteria ($m = 9$), while 29 software packages ($n = 29$) are compared for each of the m criteria. The packages are evaluated with the grading template in Section 3.3, which includes a subjective score from 1 to 10 for each quality for each package. For each quality, for a pair of packages i and j, with $score_i >= score_j$, the difference between the two scores is $diff_{ij} = score_i - score_j$. The mapping between $diff_{ij}$ (which can vary between 0 and 9) and the values in A_{ij} (which can vary between 1 and 9) is as follows:

- $A_{ij} = 1$ and $diff_{ij} = 0$ when criterion i and criterion j are equally important;
- A_{ij} increases when $diff_{ij}$ increases;
- $A_{ij} = 9$ and $diff_{ij} = 9$ when criterion i is extremely more important than criterion j.

Thus, we approximate the pairwise comparison result of i versus j by the following equation:

$$A_{ij} = \min(score_i - score_j + 1, 9) \tag{3}$$

3. Methodology

We developed a methodology for evaluating the state of the practice of research software (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 to 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, the expert is 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), online articles (Emms, 2019; Hasan, 2020; Mu, 2019), and forum discussions (Samala, 2014). The full list of 48 packages is available in Dong (2021a). To reduce the length of the list to a manageable number (29 in this case, as given in Section 4), 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. 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 1 to 4 hours for each package. Project developers can be contacted for help regarding installation, if necessary, but a cap of about 2 hours is imposed on the installation process, to keep the overall measurement time feasible. An excerpt of the spreadsheet is shown in Figure 2. A column is included 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. The questions were designed to be unambiguous, quantifiable and measurable with limited time and domain knowledge. The measures are grouped 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 used 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. Each quality is assigned 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 usability was

considered by the developers. We do this by looking for cues in the documentation, like a getting started manual, a user manual and documentation 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 a large number of 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 less 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 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. This tool is also used 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 the 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 environments for each package. VMs were used 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 spreadsheet, the details for each VM are noted, 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

Our interviews were guided by 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 the projects. We also ask about the developer's understandings of the users. Some questions focus on the current and past difficulties, and the solutions the team has found, or will try. We also discuss the importance and current situations of documentation. 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. Based on our experience, the interviewees usually brought up unexpected and exciting ideas.

We sent out interview requests to all 29 projects, with 9 projects responding. In some cases multiple developers from the same project agreed to be interviewed. We found contact information from projects websites, code repository, publications, and from biographic pages at the teams's institutions. Meetings were held on-line using either Zoom or Teams, which facilitated recording and automatic transcription of the meetings. The interview process presented here was approved by the McMaster University Research Ethics Board under the application number MREB#: 5219.

4. Measurement Results

Table 1 shows the 29 software packages that we measured, along with summary data collected in the year 2020. We arrange the items in descending order of LOC. We found the initial release dates (Rlsd) for most projects and

marked the two unknown dates with "?". We used the date of the latest change to each code repository to decide the latest update. We found funding information (Fnd) for only eight projects. For the number of contributors (NOC) we considered anyone who made at least one accepted commit as a contributor. The NOC is not usually the same as the number of long-term project members, since many projects received change requests and code from the community. With respect to the OS, 25 packages work on all three OSs: Windows (W), macOS (M), and Linux (L). Although the usual approach to cross-platform compatibility was to work natively on multiple OSes, five projects achieved platform-independence via web applications. The full measurement data for all packages is available in Dong (2021b).

Software	Rlsd	Updated	Fnd	NOC	LOC	OS			Web
						W	M	L	
ParaView (Ahrens et al., 2005)	2002	2020-10	X	100	886326	X	X	X	X
Gwyddion (Nevcas and Klapetek, 2012)	2004	2020-11		38	643427	X	X	X	
Horos (horosproject.org, 2020)	?	2020-04		21	561617		X		
OsiriX Lite (SARL, 2019)	2004	2019-11		9	544304		X		
3D Slicer (Kikinis et al., 2014)	1998	2020-08	X	100	501451	X	X	X	
Drishti (Limaye, 2012)	2012	2020-08		1	268168	X	X	X	
Ginkgo CADx (Wollny, 2020)	2010	2019-05		3	257144	X	X	X	
GATE (Jan et al., 2004)	2011	2020-10		45	207122		X	X	
3DimViewer (TESCAN, 2020)	?	2020-03	X	3	178065	X	X		
medInria (Fillard et al., 2012)	2009	2020-11		21	148924	X	X	X	
BioImage Suite Web (Papademetris et al., 2005)	2018	2020-10	X	13	139699	X	X	X	X
Weasis (Roduit, 2021)	2010	2020-08		8	123272	X	X	X	
AMIDE (Loening, 2017)	2006	2017-01		4	102827	X	X	X	
XMedCon (Nolf et al., 2003)	2000	2020-08		2	96767	X	X	X	
ITK-SNAP (Yushkevich et al., 2006)	2006	2020-06	X	13	88530	X	X	X	
Papaya (Research Imaging Institute, 2019)	2012	2019-05		9	71831	X	X	X	
OHIF Viewer (Ziegler et al., 2020)	2015	2020-10		76	63951	X	X	X	X
SMILI (Chandra et al., 2018)	2014	2020-06		9	62626	X	X	X	
INVESALIUS 3 (Amorim et al., 2015)	2009	2020-09		10	48605	X	X	X	
dwv (Martelli, 2021)	2012	2020-09		22	47815	X	X	X	X
DICOM Viewer (Afsar, 2021)	2018	2020-04	X	5	30761	X	X	X	
MicroView (Innovations, 2020)	2015	2020-08		2	27470	X	X	X	
MatrixUser (Liu et al., 2016)	2013	2018-07		1	23121	X	X	X	
Slice:Drop (Haehn, 2013)	2012	2020-04		3	19020	X	X	X	X
dicompyler (Panchal and Keyes, 2010)	2009	2020-01		2	15941	X	X		
Fiji (Schindelin et al., 2012)	2011	2020-08	X	55	10833	X	X	X	
ImageJ (Rueden et al., 2017)	1997	2020-08	X	18	9681	X	X	X	
MRIcroGL (Lab, 2021)	2015	2020-08		2	8493	X	X	X	
DicomBrowser (Archie and Marcus, 2012)	2012	2020-08		3	5505	X	X	X	

Table 1: Final software list (sorted in descending order of the number of Lines Of Code (LOC))

Figure 3 shows the primary languages versus the number of projects using them. The primary language is the language used for the majority of the project's code; in most cases projects also use other languages. The most popular language is C++, with almost 40% of projects (11 of 29). The two least popular choices are Pascal and Matlab, with around 3% of projects each (1 of 29).

4.1. Installability

Figure 4 lists the installability scores. We found installation instructions for 16 projects. Among the ones without instructions, *BioImage Suite Web* and *Slice:Drop* do not need installation, since they are web applications. Installing

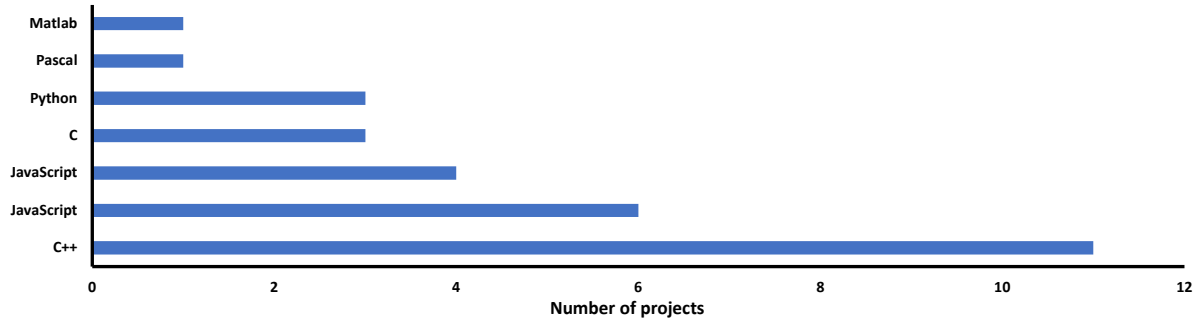

Figure 3: Primary languages versus number of projects

10 of the projects required extra dependencies. Five of them are web applications (as shown in Table 1) and depend on a browser; *dwv*, *OHIF Viewer*, and *GATE* needs extra dependencies to build; *ImageJ* and *Fiji* need an unzip tool; *MatrixUser* is based on Matlab; *DICOM Viewer* needs to work on a Nextcloud platform.

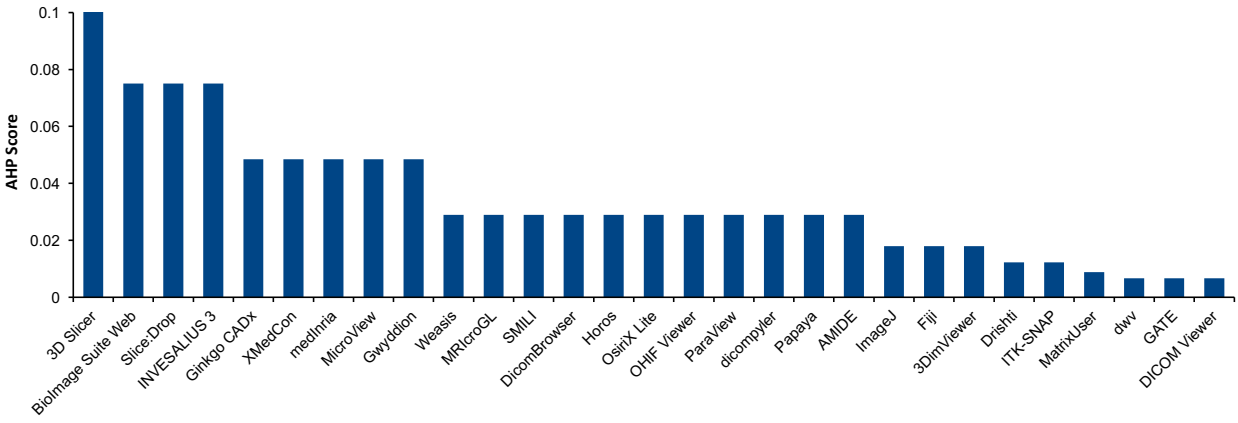

Figure 4: AHP installability scores

3D Slicer has the highest score because it had easy to follow installation instructions, and the installation processes were automated, fast, and frustration-free, with all dependencies automatically added. There were also no errors during the installation and uninstallation steps. Many other software packages also had installation instructions and automated installers, and we had no trouble installing them, such as *INVESALIUS 3*, *Gwyddion*, *XMedCon*, and *MicroView*. We determined their scores based on the understandability of the instructions, installation steps, and user experience. Since *BioImage Suite Web* and *Slice:Drop* needed no installation, we gave them high scores. *BioImage Suite Web* also provided an option to download cache for offline usage, which was easy to apply.

dwv, *GATE*, and *DICOM Viewer* showed severe installation problems. We were not able to install them, even after a reasonable amount of time (2 hours). For *dwv* and *GATE* we failed to build from the source code, but we were able to proceed with measuring other qualities using a deployed online version for *dwv*, and a VM version for *GATE*. For *DICOM Viewer* we could not install the NextCloud dependency, and we did not have another option for running the software. Therefore, for *DICOM Viewer* we could not measure reliability or robustness. The other seven qualities could be measured, since they do not require installation.

MatrixUser has a lower score because it depended on Matlab. We assessed the score from the point of view of a user that would have to install Matlab and acquire a license. Of course, for users that already work within Matlab, the installability score should be higher.

4.2. Correctness & Verifiability

The scores of correctness & verifiability are shown in Figure 5. Generally speaking, the packages with higher scores adopted more techniques to improve correctness, and had better documentation for us to verify against. For instance, we looked for evidence of unit testing, since it benefits most parts of the software's life cycle, such as designing, coding, debugging, and optimization (Hamill, 2004). We only found evidence of unit testing for about half of the projects. We identified five projects using CI/CD tools: *3D Slicer*, *ImageJ*, *Fiji*, *dwv*, and *OHIF Viewer*.

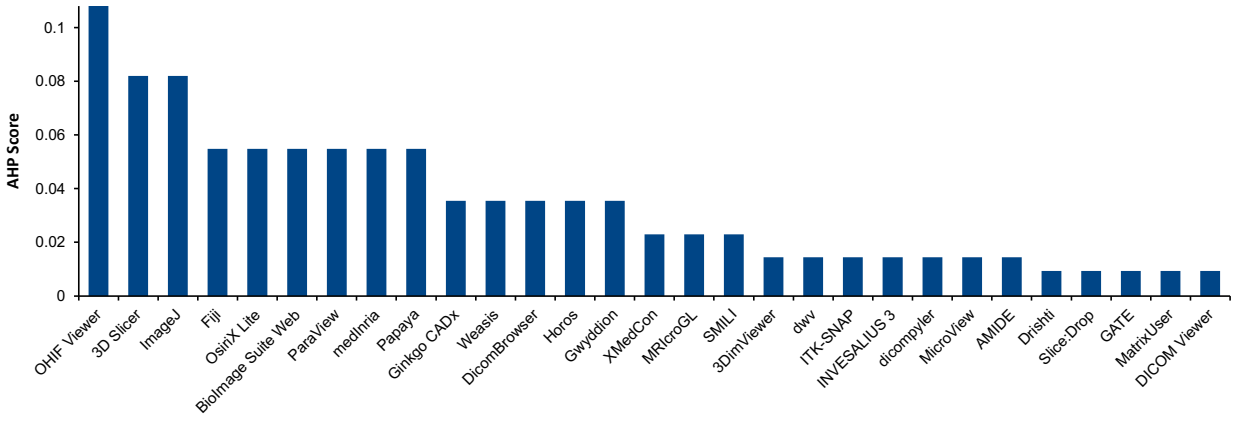

Figure 5: AHP correctness & verifiability scores

Even for some projects with well-organized documents, requirements specifications and theory manuals were still missing. We could not identify theory manuals for all projects and we did not find requirements specifications for most projects. The only document we found was a road map of *3D Slicer*, which contained design requirements for upcoming changes.

4.3. Surface Reliability

Figure 6 shows the AHP results. As shown in Section 4.1, most of the software products did not "break" during installation, or did not need installation; *dwv* and *GATE* broke in the building stage, and the processes were not recoverable; we could not install the dependency for *DICOM Viewer*. Of the seven software packages with a getting started tutorial and operation steps in the tutorial, most showed no error when we followed the steps. However, *GATE* could not open macro files and became unresponsive several times, without any descriptive error message. When assessing robustness (Section 4.4), we found that *Drishti* crashed when loading damaged image files, without showing any descriptive error message. On the other hand, we did not find any problems with the online version of *dwv*.

4.4. Surface Robustness

Figure 7 presents the scores for surface robustness. The packages with higher scores elegantly handled unexpected/unanticipated inputs, typically showing a clear error message. We may have underestimated the score of *OHIF Viewer*, since we needed further customization to load data.

Digital Imaging and Communications in Medicine (DICOM) "defines the formats for medical images that can be exchanged with the data and quality necessary for clinical use" (Association, 2021). According to their documentation, all 29 software packages should support the DICOM standard. To test robustness, we prepared two types of image files: correct and incorrect formats (with the incorrect format created by relabelled a text file to have the ".dcm" extension). All software packages loaded the correct format image, except for *GATE*, which failed for unknown reasons. For the broken format, *MatrixUser*, *dwv*, and *Slice:Drop* ignored the incorrect format of the file and loaded it regardless. They did not show any error message and displayed a blank image. *MRIcroGL* behaved similarly except that it showed a meaningless image. *Drishti* successfully detected the broken format of the file, but the software crashed as a result.

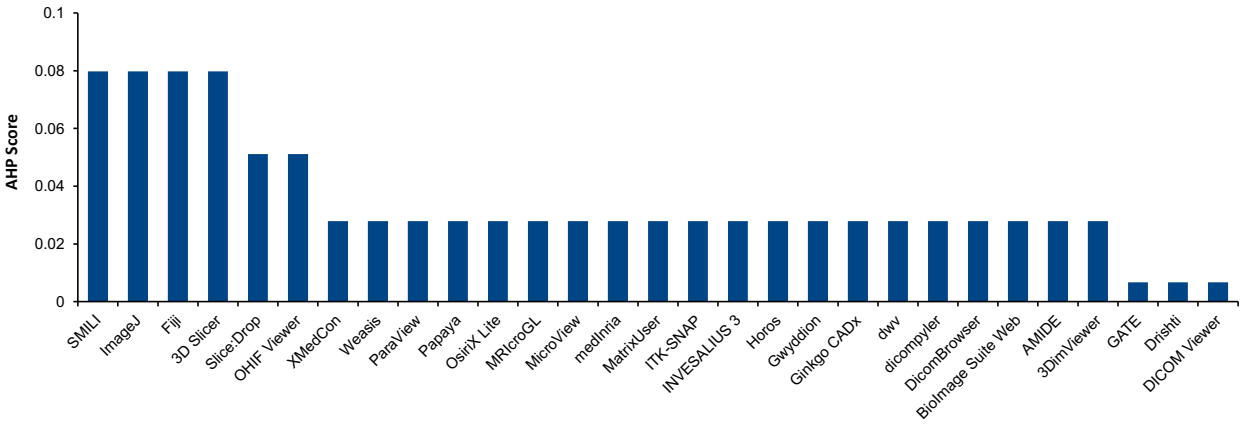

Figure 6: AHP surface reliability scores

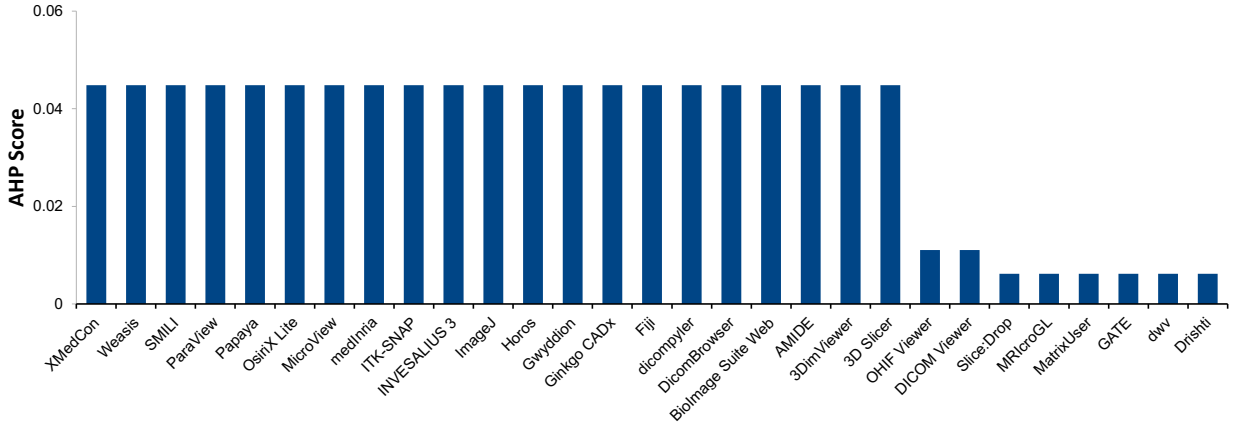

Figure 7: AHP surface robustness scores

4.5. Surface Usability

Figure 8 shows the AHP scores for surface usability. The software with higher scores usually provided both comprehensive documented guidance and a good user experience. *INVESALIUS 3* provided an excellent example of a detailed and precise user manual. *GATE* also provided a large number of documents, but unfortunately we had difficulty understanding and using them. We found getting started tutorials for only 11 projects, but a user manual for 22 projects. *MRIcroGL* was the only project that explicitly documented expected user characteristics.

4.6. Maintainability

Figure 9 shows the ranking results for maintainability. We marked *3D Slicer* with the highest score because we found it to have the most comprehensive artifacts. For example, as far as we could find, only a few of the 29 projects had a product, developer's manual, or API documentation, and only *3D Slicer*, *ImageJ*, *Fiji* included all three documents. Moreover, *3D Slicer* has a much higher percentage of closed issues (91.65%) compared to *ImageJ* (52.49%) and *Fiji* (63.79%). Table 2 shows which projects had these documents, in the descending order of their maintainability scores.

27 of the 29 projects used git as the version control tool, with 24 of these using GitHub. *AMIDE* used Mercurial and *Gwyddion* used Subversion. *XMedCon*, *AMIDE*, and *Gwyddion* used SourceForge. *DicomBrowser* and *3DimViewer* used BitBucket.

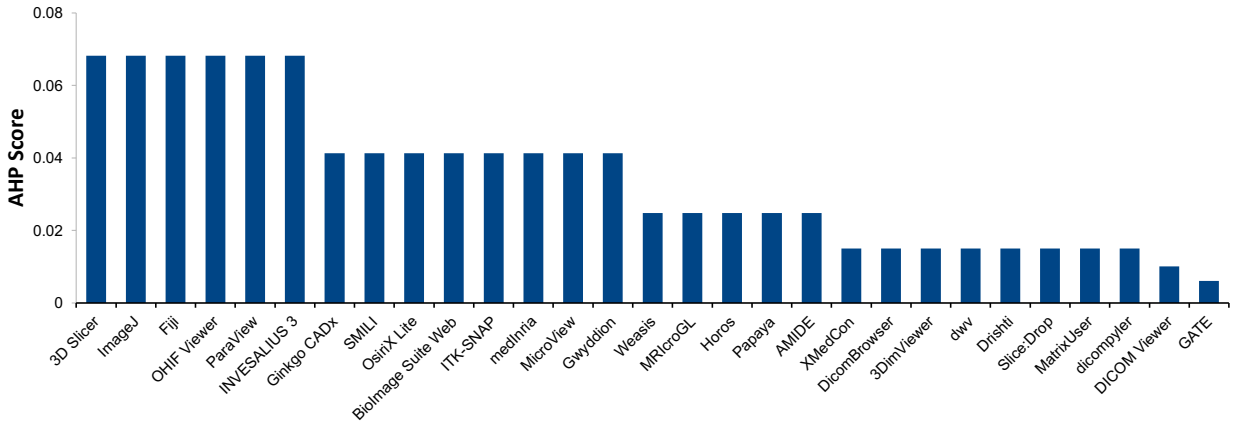

Figure 8: AHP surface usability scores

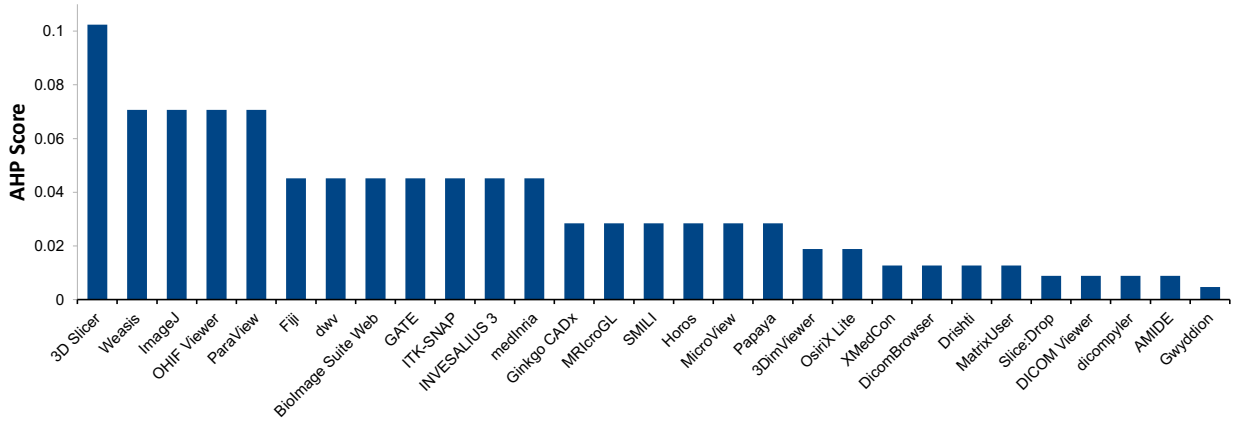

Figure 9: AHP maintainability scores

Software	Prod. roadmap	Dev. manual	API doc.
3D Slicer	X	X	X
ImageJ	X	X	X
Weasis		X	
OHIF Viewer		X	X
Fiji	X	X	X
ParaView	X		
SMILI			X
medInria		X	
INVESALIUS 3	X		
dwv			X
BioImage Suite Web		X	
Gwyddion		X	X

Table 2: Software with the maintainability documents (listed in descending order of maintainability score)

4.7. Reusability

Figure 10 shows the AHP results for reusability. As described in Section 3.3, we gave higher scores to the projects with API documentation. As shown in Table 2, seven projects had API documents. We also assumed that projects with more code files and less LOC per code file as more reusable. Table 3 shows the number of text-based files by projects, which we used to approximate the number of code files. The table also lists the total number of lines (including comments and blanks), LOC, and average LOC per file. We arranged the items in descending order of their reusability scores.

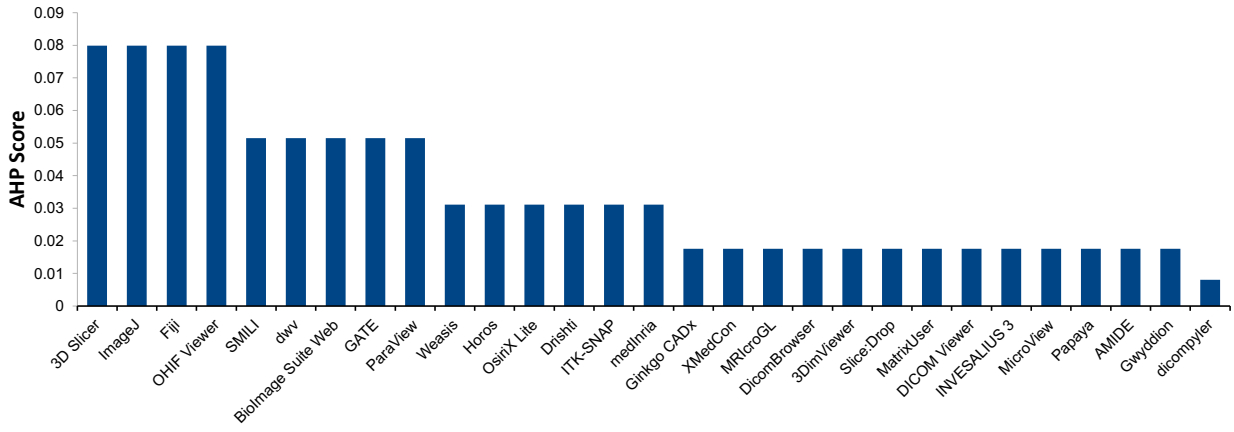

Figure 10: AHP reusability scores

4.8. Surface Understandability

Figure 11 shows the scores for surface understandability. All projects had a consistent coding style with parameters in the same order for all functions; the code was modularized; the comments were clear, indicating what is being done, not how. However, we only found explicit identification of a coding standard for 3 out of the 29: *3D Slicer*, *Weasis*, and *ImageJ*. We also found hard-coded constants (rather than symbolic constants) in *medInria*, *dicompyler*, *MicroView*, and *Papaya*. We did not find any reference to the algorithms used in projects *XMedCon*, *DicomBrowser*, *3DimViewer*, *BioImage Suite Web*, *Slice:Drop*, *MatrixUser*, *DICOM Viewer*, *dicompyler*, and *Papaya*.

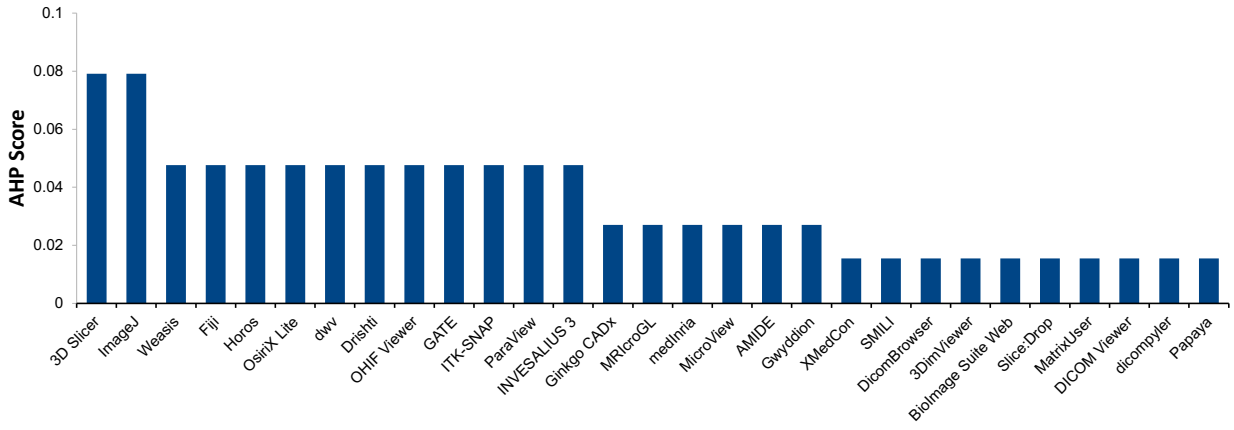

Figure 11: AHP surface understandability scores

Software	Text files	Total lines	LOC	LOC/file
OHIF Viewer	1162	86306	63951	55
3D Slicer	3386	709143	501451	148
Gwyddion	2060	787966	643427	312
ParaView	5556	1276863	886326	160
OsiriX Lite	2270	873025	544304	240
Horos	2346	912496	561617	239
medInria	1678	214607	148924	89
Weasis	1027	156551	123272	120
BioImage Suite Web	931	203810	139699	150
GATE	1720	311703	207122	120
Ginkgo CADx	974	361207	257144	264
SMILI	275	90146	62626	228
Fiji	136	13764	10833	80
Drishti	757	345225	268168	354
ITK-SNAP	677	139880	88530	131
3DimViewer	730	240627	178065	244
DICOM Viewer	302	34701	30761	102
ImageJ	40	10740	9681	242
dwv	188	71099	47815	254
MatrixUser	216	31336	23121	107
INVESALIUS 3	156	59328	48605	312
AMIDE	183	139658	102827	562
Papaya	110	95594	71831	653
MicroView	137	36173	27470	201
XMedCon	202	129991	96767	479
MRIcroGL	97	50445	8493	88
Slice:Drop	77	25720	19020	247
DicomBrowser	54	7375	5505	102
dicompyler	48	19201	15941	332

Table 3: Number of files and lines (entries are sorted in descending order of their reusability scores)

4.9. Visibility/Transparency

Figure 12 shows the AHP scores for visibility/transparency. Generally speaking, the teams that actively documented their development process and plans scored higher. Table 4 shows the projects that had documents for the development process, project status, development environment, and release notes, in descending order of their visibility/transparency scores.

4.10. Overall Scores

As described in Section 2.3, for our AHP measurements, we have nine criteria (qualities) and 29 alternatives (software packages). In the absence of a specific real world context, we assumed all nine qualities are equally important. Figure 13 shows the overall scores in descending order. Since we produced the scores from the AHP process, the total sum of the 29 scores is precisely 1.

The top three software products *3D Slicer*, *ImageJ*, and *OHIF Viewer* had higher scores in most criteria. *3D Slicer* ranked in the top two software products for all qualities except *surface robustness*; *ImageJ* ranked in the top three for correctness & verifiability, surface reliability, surface usability, maintainability, surface understandability, and visibility/transparency. *OHIF Viewer* ranked in the top five products for correctness & verifiability, surface reliability, surface usability, maintainability, and reusability. Given the installation problems, we may have underestimated the scores on reliability and robustness for *DICOM Viewer*, but we compared it equally for the other seven qualities.

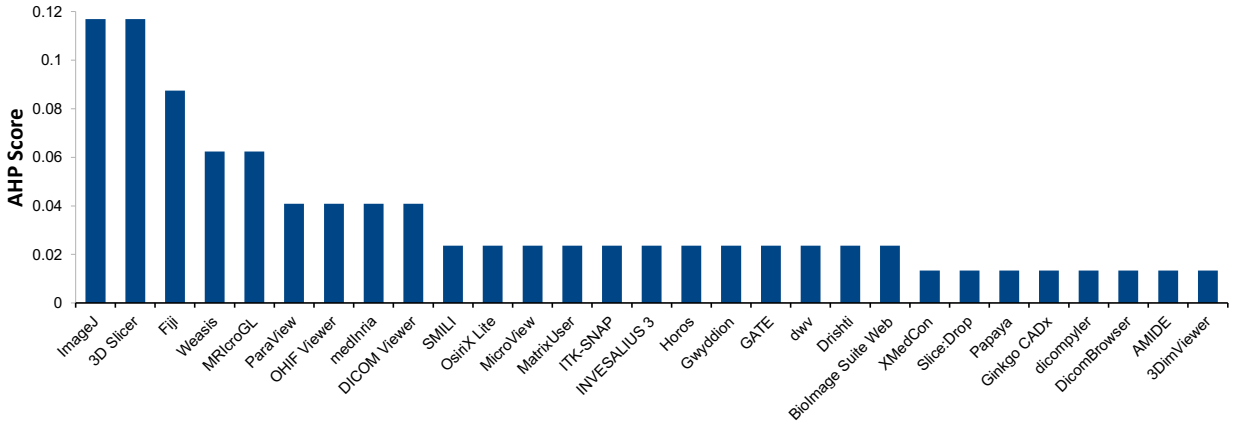

Figure 12: AHP visibility/transparency scores

Software	Dev process	Proj status	Dev env	Rls notes
3D Slicer	X	X	X	X
ImageJ	X	X	X	X
Fiji	X	X	X	
MRIcroGL				X
Weasis			X	X
ParaView		X		
OHIF Viewer			X	X
DICOM Viewer			X	X
medInria			X	X
SMILI				X
Drishti				X
INVESALIUS 3				X
OsiriX Lite				X
GATE				X
MicroView				X
MatrixUser				X
BioImage Suite Web			X	
ITK-SNAP				X
Horos				X
dwv				X
Gwyddion				X

Table 4: Software with the visibility/transparency documents (software is listed in descending order of visibility/transparency score)

5. Comparison to Community Ranking

To address RQ3 about how our ranking compares to the popularity of projects as judged by the scientific community, we make two comparisons:

- A comparison of our ranking (from Section 4) with the community ratings on GitHub, as shown by GitHub stars, number of forks, and number of people watching the projects; and,
- A comparison of top-rated software from our methodology with the top recommendations from our domain experts (as mentioned in Section 3.2).

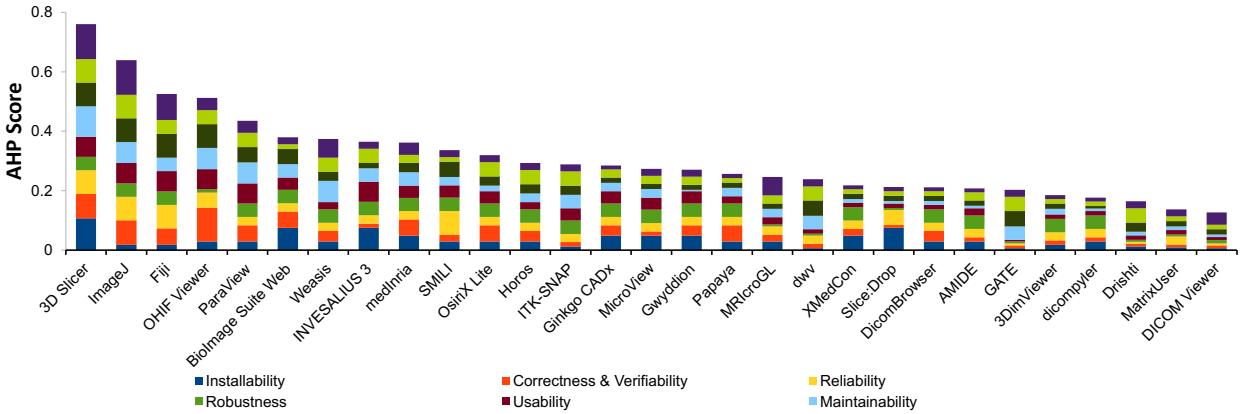

Figure 13: Overall AHP scores with an equal weighting for all 9 software qualities

Table 5 shows our ranking of the 29 MI projects, and their GitHub metrics, if applicable. As mentioned in Section 4.6, 24 projects used GitHub. Since GitHub repositories have different creation dates, we collect the number of months each stayed on GitHub, and calculate the average number of new stars, people watching, and forks per 12 months. The method of getting the creation date is described in Section 3.3. The items in Table 5 are listed in descending order of the average number of new stars per year. The non-GitHub items are listed in the order of our ranking. All GitHub statistics were collected in July, 2021.

Generally speaking, most of the top-ranking MI software projects also received greater attention and popularity on GitHub. Between our ranking and the GitHub stars-per-year ranking, four of the top five software projects appear in both lists.Our top 5 packages are scattered among the first 8 positions on the GitHub list. However, as discussed below there are discrepancies between the two lists.

In some cases projects are popular in the community, but were assigned a low rank by our methodology. This is the case for *dwv*. The reason for the low ranking is that, as mentioned in Section 4.1, we failed to build it locally, and used the test version on its websites for the measurements. We followed the instructions and tried to run the command "yarn run test" locally, which did not work. In addition, the test version did not detect a broken DICOM file and displayed a blank image as described in Section 4.4. We might underestimate the scores for *dwv* due to uncommon technical issues. We also ranked *DICOM Viewer* much lower than its popularity. As mentioned in Section 4.1 it depended on the NextCloud platform that we could not successfully install. Thus, we might underestimate the scores of its surface reliability and surface robustness. In addition, we weighted all qualities equally, which is not likely to be the case with all users. As a result, some projects with high community popularity may have scored lower with our method because of a relatively higher (compared to the scientific community's implicit ranking) weighting of the poor scores for some qualities. A further explanation for discrepancies between our measures and the star measures may also be due to inaccuracy with using stars to approximate popularity. Stars are not an ideal measure because stars represent the community's feeling in the past more than they measure current preferences (Szulik, 2017). The issue with stars is that they tend only to be added, not removed. A final reason for inconsistencies between our ranking and the community's ranking is that, as for consumer products, more factors influence popularity than just quality.

As shown in Section 3.2, our domain experts recommended a list of top software with 12 software products. All of the top 4 entries from the Domain Expert's list are among the top 12 ranked by our methodology. Three of the top four on both lists are the same: *3D Slicer*, *ImageJ*, and *Fiji*. *3D Slicer* is top project by both rankings (and by the GithHub stars measure as well). The Domain Expert ranked *Horos* as their second choice, while we ranked it twelfth. Our third ranked project, *OHIF Viewer* was not listed by the Domain Expert. Neither were the software packages that we ranked from fifth to eleventh (*ParaView*, *Weasis*, *medInria*, *BioImage Suite Web*, *OsiriX Lite*, *INVESALIUS*, and *Gwyddion*). The software mentioned by the Domain Expert that we did not rank were the six recommended packages that did not have visualization as the primary function (as discussed in Section 3.2). The differences between the list recommended by our methodology and the Domain Expert are not surprising. As mentioned above, the methodology weights all qualities equally, but that may not be the case for the Domain Expert's impressions. Moreover, although

Software	Comm. rank	Our rank	Stars/yr	Watches/yr	Forks/yr
3D Slicer	1	1	284	19	128
OHIF Viewer	2	3	277	19	224
dwv	3	23	124	12	51
ImageJ	4	2	84	9	30
ParaView	5	5	67	7	28
Horos	6	12	49	9	18
Papaya	7	17	45	5	20
Fiji	8	4	44	5	21
DICOM Viewer	9	29	43	6	9
INVESALIUS 3	10	10	40	4	17
Weasis	11	6	36	5	19
dicompyler	12	26	35	5	14
OsiriX Lite	13	9	34	9	24
MRIcroGL	14	18	24	3	3
GATE	15	25	19	6	26
Ginkgo CADx	16	14	19	4	6
BioImage Suite Web	17	8	18	5	7
Drishti	18	27	16	4	4
Slice:Drop	19	20	10	2	5
ITK-SNAP	20	15	9	1	4
medInria	21	7	7	3	6
SMILI	22	13	3	1	2
MatrixUser	23	28	2	0	0
MicroView	24	16	1	1	1
Gwyddion	25	11	n/a	n/a	n/a
XMedCon	26	19	n/a	n/a	n/a
DicomBrower	27	21	n/a	n/a	n/a
AMIDE	28	22	n/a	n/a	n/a
3DimViewer	29	24	n/a	n/a	n/a

Table 5: Software ranking by our methodology versus the community (Comm.) ranking using GitHub metrics (Sorted in descending order of community popularity, as estimated by the number of new stars per year)

the Domain Expert has significant experience with MI software, he has not used every one of the 29 packages that were measured.

Although our ranking and the estimate of the community's ranking are not perfect measures, they do suggest a correlation between best practices and popularity. We do not know which comes first, the use of best practices or popularity, but we do know that the top ranked packages tend to incorporate best practices. The next sections will explore how the practices of the MI community compare to the broader research software community. We will also investigate the practices from the top projects that others within the MI community, and within the broader research software community, can potentially adopt.

6. Comparison Between MI and Research Software for Artifacts

As part of filling in the measurement template (from Section 3.3) we summarized the artifacts observed in each MI package. Table 6 groups the artifacts by frequency into categories of common (20 to 29 (>67%) packages), uncommon (10 to 19 (33-67%) packages), and rare (1 to 9 (<33%) packages). The full measurements are summarized in Dong (2021b). The details on which projects use which types of artifacts are summarized in Tables 2 and 4 for documents related to maintainability and visibility, respectively.

We answer RQ4 by comparing the artifacts that we observed in MI repositories to those observed and recommended for research software in general. Our comparison may point out areas where some MI software packages fall short of current best practices. This is not intended to be a criticism of any existing packages, especially since in practice not every project needs to achieve the highest possible quality. However, rather than delve into the nuances

Common	Uncommon	Rare
README (29)	Build scripts (18)	Getting Started (9)
Version control (29)	Tutorials (18)	Developer's manual (8)
License (28)	Installation guide (16)	Contributing (8)
Issue tracker (28)	Test cases (15)	API documentation (7)
User manual (22)	Authors (14)	Dependency list (7)
Release info. (22)	Frequently Asked Questions (FAQ) (14)	Troubleshooting guide (6)
	Acknowledgements (12)	Product roadmap (5)
	Changelog (12)	Design documentation (5)
	Citation (11)	Code style guide (3)
		Code of conduct (1)
		Requirements (1)

Table 6: Artifacts Present in MI Packages, Classified by Frequency (The number in brackets is the number of occurrences)

of which software can justify compromising which practices we will write our comparison under the ideal assumption that every project has sufficient resources to match best practices.

Table 7 (based on data from (Smith and Michalski, 2022)) shows that MI artifacts generally match the recommendations found in nine current research software development guidelines:

- United States Geological Survey Software Planning Checklist (USGS, 2019),
- DLR (German Aerospace Centre) Software Engineering Guidelines (Schlauch et al., 2018),
- Scottish Covid-19 Response Consortium Software Checklist (Brett et al., 2021),
- Good Enough Practices in Scientific Computing (Wilson et al., 2016),
- xSDK (Extreme-scale Scientific Software Development Kit) Community Package Policies (Smith et al., 2018a),
- Trilinos Developers Guide (Heroux et al., 2008),
- EURISE (European Research Infrastructure Software Engineers') Network Technical Reference (Thiel, 2020),
- CLARIAH (Common Lab Research Infrastructure for the Arts and Humanities) Guidelines for Software Quality (van Gompel et al., 2016), and
- A Set of Common Software Quality Assurance Baseline Criteria for Research Projects (Orviz et al., 2017).

In Table 7 each row corresponds to an artifact. For a given row, a checkmark in one of the columns means that the corresponding guideline recommends this artifact. The last column shows whether the artifact appears in the measured set of MI software, either not at all (blank), commonly (C), uncommonly (U) or rarely (R). We did our best to interpret the meaning of each artifact consistently between guidelines and specific MI software, but the terminology and the contents of artifacts are not standardized. The challenge even exists for the ubiquitous README file. As illustrated by Prana et al. (2018), the content of README files shows significant variation between projects. Although some content is reasonably consistent, with 97% of README files contain at least one section describing the 'What' of the repository and 89% offering some 'How' content, other categories are more variable. For instance, information on 'Contribution', 'Why'', and 'Who', appear in 28%, 26% and 53% of the analyzed files, respectively (Prana et al., 2018).

The frequency of checkmarks in Table 7 indicates the popularity of recommending a given artifact, but it does not imply that the most commonly recommended artifacts are the most important artifacts. Just because an artifact is not explicitly recommended in a given guidelines, does not mean that the artifact is not valued by the guideline authors. They may have exluded it because it is out of the scope of their recommendations, or outside of their experience. For

instance, an artifact related to uninstall is only explicitly mentioned by (van Gompel et al., 2016)), but other guideline authors would likely see its value. They may simply feel that uninstall is implied by install, or they may have never asked themselves whether separate uninstall instructions are needed.

	USGS (2019)	Schlauch et al. (2018)	Brett et al. (2021)	Wilson et al. (2016)	Smith et al. (2018a)	Heroux et al. (2008)	Thiel (2020)	van Gompel et al. (2016)	Orviz et al. (2017)	MI
LICENSE	✓	✓	✓	✓	✓		✓	✓	✓	C
README		✓	✓	✓	✓		✓	✓	✓	C
CONTRIBUTING		✓	✓	✓	✓		✓	✓	✓	R
CITATION				✓				✓	✓	U
CHANGELOG		✓		✓	✓		✓			U
INSTALL					✓		✓	✓	✓	U
Uninstall								✓		
Dependency List			✓		✓			✓		R
Authors							✓	✓	✓	U
Code of Conduct							✓			R
Acknowledgements							✓	✓	✓	U
Code Style Guide		✓					✓	✓	✓	R
Release Info.		✓				✓	✓			C
Prod. Roadmap						✓	✓	✓		R
Getting started					✓		✓	✓	✓	R
User manual			✓				✓			C
Tutorials							✓			U
FAQ							✓	✓	✓	U
Issue Track		✓	✓		✓	✓	✓		✓	C
Version Control		✓	✓	✓	✓	✓	✓	✓	✓	C
Build Scripts		✓		✓	✓	✓	✓		✓	U
Requirements		✓				✓			✓	R
Design Doc.		✓	✓		✓		✓	✓	✓	R
API Doc.					✓		✓	✓	✓	R
Test Plan		✓				✓				
Test Cases	✓	✓	✓		✓	✓	✓	✓	✓	U

Table 7: Comparison of Recommended Artifacts in Software Development Guidelines to Artifacts in MI Projects (C for Common, U for Uncommon and R for Rare)

Two of the items that appear in Table 6 do not appear in the software development guidelines shown in Table 7: Troubleshooting guide and Developer's manual. Although these two artifacts aren't specifically named in the guidelines that we found, the information contained within them overlaps with the recommended artifacts. A Troubleshooting guideline contains information that would typically be found in a User manual. A Developer's guide overlaps with information from the README, INSTALL, Uninstall, Dependency List, Release Information, API documentation and Design documentation. In our current analysis, we have identified artifacts by the names given by the software guidelines and MI examples. In the future, a more in-depth analysis would look at the knowledge fragments that are captured in the artifacts, rather than focusing on the names of the files that collect these fragments together.

Although the MI community shows examples of 88% (23 of 26) of the practices we found in research software guidelines (Table 7), three recommended artifacts were not observed: i) Unistall, ii) Test plans, and iii) Requirements.

Uninstall is likely an omission caused by the focus on installing software. Given the storage capacity of current hardware, developers and users do not generally concerned with uninstall. Moreover, as mentioned above, uninstall is not particularly emphasized in existing recommendations. Test plans were not observed for MI software, but that doesn't mean they weren't created; it means that the plans are not under version control. Test plans would have to at least be implicitly created, since test cases were observed with reasonable frequency for MI software (uncommon).

MI software is like other research software in its neglect of requirements documentation. Although requirements documentation is recommended by some (Schlauch et al., 2018; Heroux et al., 2008; Smith and Koothoor, 2016), in practice research software developers often do not produce a proper requirements specification (Heaton and Carver, 2015). Sanders and Kelly (2008) interviewed 16 scientists from 10 disciplines and found that none of the scientists created requirements specifications, unless regulations in their field mandated such a document. Nguyen-Hoan et al. (2010) showed requirements are the least commonly produced type of documentation for research software in general. When looking at the pain points for research software developers, Wiese et al. (2019) found that software requirements and management is the software engineering discipline that most hurts scientific developers, accounting for 23% of the technical problems reported by study participants. The lack of support for requirements is likely due to the perception that up-front requirements are impossible for research software (Carver et al., 2007; Segal and Morris, 2008), but when the instance on "up-front" requirements is dropped, allowing the requirements to be written iteratively and incrementally, requirements are feasible (Smith, 2016).

Table 7 shows several artifacts that are rarely observed in practice. A theme amount these rare artifacts is that many of them are related to developers more than users. For instance, the rare artifacts include Contributing, developer Code of conduct, Code Style Guide, Product roadmaps, Requirements, Design documentation, API documentation and a Test plan. The rare artifacts for MI software are similar to the rare artifacts for Lattice Boltzmann solvers (Michalski, 2021). However, MI software differs from ocean modelling software, since, according to (Jung et al., 2022), the later pays relatively less attention to testing.

To improve MI software in the future, an increased use of checklists could help. Checklists can be used in projects to ensure that best practices are followed by all developers. Some examples include checklists merging branches into master (Brown, 2015), checklists for saving and sharing changes to the project (Wilson et al., 2016), checklists for new and departing team members (Heroux and Bernholdt, 2018), checklists for processes related to commits and releases (Heroux et al., 2008) and checklists for overall software quality (Thiel, 2020; Institute, 2022). For instace, for Lattice Boltzmann solver software, ESPResSo has a checklist for managing releases (Michalski, 2021).

The above discussion shows that, taken together, MI projects fall somewhat short of recommended best practices for research software. However, MI software is not alone in this. Many, if not most, research projects fall short of best practices. A gap exists in scientific computing development practices and software engineering recommendations (Storer, 2017; Kelly, 2007; Owojaiye et al., 2021). Johanson and Hasselbring (2018) observe that the state-of-the-practice for SCS in industry and academia does not incorporate state-of-the-art SE tools and methods. This causes sustainability and reliability problems (Faulk et al., 2009). Rather than benefit from capturing and reusing previous knowledge, projects waste time and energy "reinventing the wheel" (de Souza et al., 2019).

7. Comparison of Tool Usage Between MI and Other Research Software

Software tools are used to support the development, verification, maintenance, and evolution of software, software processes, and artifacts (Ghezzi et al., 2003, p. 501). To tools used in MI software include tools for CI/CD, user support, version control, documentation and project management. We answer RQ5 by comparing aspects of tool usage in MI software packages to their utilization in the research software community in general.

CI/CD paragraph. As mentioned in Section 4.2, we identified five projects using CI/CD tools: *3D Slicer*, *ImageJ*, *Fiji*, *dwv*, and *OHIF Viewer*. We identified the above projects and tools by examining the documentation and source code of all projects. Potentially not all tools leave identifiable artifacts in the repositories; therefore, we may have missed the use of some tools.

user support paragraph. Table 8 summarizes the user support models by the number of projects using them (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 happened as users sought help.

version control paragraph. From Section 4.6, 27 of the 29 projects used git as the version control tool, with 24 of these using GitHub. *AMIDE* used Mercurial and *Gwyddion* used Subversion. *XMedCon*, *AMIDE*, and *Gwyddion* used

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 8: User support models by number of projects

SourceForge. *DicomBrowser* and *3DimViewer* used BitBucket. Although teams may have a process for accepting new contributions, no one discussed this during their interview. However, most teams (8 of 9) 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 (3 teams), contributions from forums (1 team) and sharing the git repository via e-mail (1 team). [I do not know what this last one means. Should ask Ao. —SS] [The last one happened when cloud-based repositories (Github, Gitlab, Bitbucket, etc.) were not widely used. The project git repository was stored locally on the project owner's local computer. The owner shared the whole git repo to team members and other individuals who wanted to contribute. The contributors made changes to the repo, and sent it back. They transferred the repo back and forth by email. —AD]

documentation paragraph. 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, google forms and state management.

project management paragraph. 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, GitHub was mentioned 3 times, while each of the other tools was only mentioned once.

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

This section answers 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 data collection for MI software, the software development process is not explicitly indicated in the artifacts for most of the packages. However, during our interviews one developer (ESPResSo) told us their non-rigorous development model is like a combination of agile and waterfall. Employing a loosely defined process makes sense for MI software, given that the teams are generally small and self-contained. Although eleven of the packages explicitly convey that they would accept outside contributors, generally the teams are centralized, often working at the same institution. Working at the same institution means that an informal process can show success, since informal conversations are relatively easy to have.

We answer this question by measuring the qualities and interviewing the developers. We previously summarized principles, processes, and methodologies in software testing (Section 8), documentation (Section 8.1, Table 2, Table 4, contribution management (Section 8.2), and project management (Section 8.2).

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. One pain point in the development process is the lack of access to real-world datasets for testing. The developers' strategies to address this are summarized in Section 10 [to be fixed —AD]. One threat to correctness is: with huge datasets for testing, the tests are expensive and time-consuming. Three interviewees endorsed self tests / automated tests, which may save time for testing.

Documentation is apparently not emphasized in any of the 29 projects. None of them had theory manuals, although we did identify a road map in the *3D Slicer* project. No requirements specifications were found. Eight of the nine interviewees thought that documentation was essential to their projects. However, they hold the common opinion that their documentation needed improvements. Nearly half of them also believed that the lack of time prevented them from improving the documentation.

8.1. Documents in the Projects

We tried to understand the interviewees' opinions on documentation and the quality of documentations with two questions:

Q11. How does documentation fit into your development process? Would improved documentation help with the obstacles you typically face?

Q19. Do you think the current documentation can clearly convey all necessary knowledge to the users? If yes, how did you successfully achieve it? If no, what improvements are needed?

Table 9 summarizes interviewees' opinions on documentation. Interviewees from each of the eight projects thought that documentation was important 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.

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 9: Opinions on documentation by the numbers of interviewees with the answers

8.2. Project Management

To investigate project management we asked three questions:

Q5. Do you have a defined process for accepting new contributions into your team?

Q13. What is your software development model? For example, waterfall, agile, etc.

Q14. What is your project management process? Do you think improving this process can tackle the current problem? Were any project management tools used?

For managing contributions, the *3D Slicer* team encouraged users to develop their extensions for specific use cases, while the *OHIF* team enabled the use of plug-ins. The interviewee from the *ITK-SNAP* team said one way of accepting new team members was through funded academic projects.

For the most part, 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.

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.

Most interviewees skipped the part of the last question asking "Do you think improving this process can tackle the current problem?". In retrospect, we should not have asked a yes-or-no question, since a binary answer is not very informative. The interviewee from the *OHIF* team gave a positive answer to this question. They believed that a better project management process can improve the efficiency of junior developers. They also improved the project management tools (from public Jira to public GitHub repository plus private Jira), so they could better communicate externally and internally.

9. Developer Pain Points

Based on interviews with four developers, this section aims to answer the research questions: i) What are the pain points for developers working on research software projects (RQ7)?; and, ii) How do the pain points of developers from MI compare to the pain points for research software in general (RQ8)? Below we go through each of the identified pain points and include citations that contrast the MI experience with observations from researchers in other domains. Potential ways to address the pain points are covered in Section 10. The full interview questions are found in Smith et al. (2021).

The measurement results in Section 4 are based on information we collected 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. To collect this information we interviewed nine developers from eight of the 29 MI software projects. The eight projects are *3D Slicer*, *INVESALIUS 3*, *dwv*, *BioImage Suite Web*, *ITK-SNAP*, *MRIcroGL*, *Weasis*, and *OHIF*. We spent about 90 minutes for each interview and asked 20 prepared questions (summarized in Smith et al. (2021)). We also asked following-up questions as the conversation dictated. One participant was too busy to have an interview, so they wrote down their answers. The interviewees may have provided multiple answers to each question. Thus, when we summarize the frequency of different responses, the total is sometimes larger than nine.

The full interview answers can be found in Dong (2021a). Selected questions and answers are highlighted and analyzed below. The question numbers refer to the numbering in Smith et al. (2021).

The pain points include current and past obstacles, as explored via the following questions:

Q9. Currently, what are the most significant obstacles in your development process?

Q10. How might you change your development process to remove or reduce these obstacles?

Q12. In the past, is there any major obstacle to your development process that has been solved? How did you solve it?

Table 10 shows the number of times the interviewees mentioned the current and past obstacles in their projects. We group these pain points into three major categories of obstacles: *resource*, *balance*, and *testing*. We put the less mentioned ones into the category *Others*.

Developers identified a resource related pain point caused by a lack of funding and time. 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;
- Licensing the software to commercial companies that integrate it into their products;
- Improving documentation to save time answering users' and developers' questions;
- Supporting third-party plugins and extensions; and,
- Using GitHub Actions for CI/CD to save time.

Category	Obstacle	Num ans.	
		current	past
Resource	Lack of fundings	3	
	Lack of time to devote to the project	2	1
Balance	Hard to keep up with changes in OS and libraries	1	
	Hard to support multiple OS	2	
	Hard to support lower-end computers	1	2
Testing	Lack of access to real-world datasets for testing	3	2
Others	Hard to have a high level roadmap from the start	1	
	Not enough participants for usability tests	1	
	Only a few people fully understand the large codebase	1	
	Hard to transfer to new technologies		2
	Hard to understand users' needs		1
	Hard to maintain good documentations		1

Table 10: Current and past obstacles by the numbers of interviewees with the answers

Many interviewees thought lack of fundings and lack of time were their most significant obstacles. The interviewees from *3D Slicer* team and *OHIF* team pointed out that it was more challenging to get fundings for software maintenance as opposed to research. The interviewee from the *ITK-SNAP* team thought more fundings was a way to solve the lack of time problem, because they could hire more dedicated developers. On the other hand, the interviewee from the *Weasis* team did not feel that fundings could solve the same problem, since he still would need a lot of time to supervise the project. No interviewee suggested any solution to bring extra funding to the project. However, they provided ideas to save time, such as better documentation, third-party plugins, and good CI/CD tools.

With respect to balance, developers expressed difficulty balancing between four factors: cross-platform compatibility, convenience to development & maintenance, performance, and security. The potential and proven solutions are:

- 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;
- Increasing funding;
- Maintaining better documentations to ease the development & maintenance processes;
- Improving performance via more powerful computers, which one interviewee pointed out has already happened to reduce the balance problem.

As the above list shows, web-based applications are perceived to help the balance problem. Table 11 shows the teams' choices between native application and web application. In all the 29 teams on our list, most of them chose to develop native applications. For the eight teams we interviewed, three of them were building web applications, and the *MRIcroGL* team was considering web-based solutions. So we had a good chance to discuss the differences between the two choices with the interviewees.

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 are mirrored by the native application 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 the cost of the servers.

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

Table 11: Teams' choices between native application and web application

For the pain point category of testing developers identified the the lack of access to real-world datasets. Some proposed and proven solutions are as follows:

- Using open datasets
- Asking the users to provide de-identified copies of medical images if they have problems loading the images;
- Sending the beta versions of software to medical workers who can access the data and complete the tests;
- 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;
- Storing all images that cause special problems, and maintaining this special dataset over time.

No interviewee provided a perfect solution to the testing problem. However, increased connections between the development team and medical professionals/institutions could ease the pain.

9.1. Discussions on Software Qualities

Questions 15–18, and 20 cover the software qualities of correctness, maintainability, understandability, usability, and reproducibility, respectively. We discuss each quality below.

Q15. Was it hard to ensure the correctness of the software? If there were any obstacles, what methods have been considered or practiced to improve the situation? If practiced, did it work?

The interviewees identified multiple threats to correctness. The most frequently mentioned threat was complexity. Complexity enters the software by various means, including a 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 by at least one time include the following:

- Lack of real world image data for testing;
- The team cannot use private data for debugging even when the data causes problems; [Why not? Should ask Ao —SS] [For example, when a physician faces some problems to use the software to display a medical image, he/she might reach to the team for help. However, the image is private, so the physician can only describe the problems, but not share the image with the team to reproduce the problems for debugging. —AD]
- Expense and time consumed by tests because of the huge datasets;

- Difficulty managing releases;
- No unit tests; and,
- No dedicated quality assurance team.

Testing was the most often mentioned strategy 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. Another approach frequently adopted (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.

Q16. When designing the software, did you consider the ease of future changes? For example, will it be hard to change the system's structure, modules, or code blocks? What measures have been taken to ensure the ease of future changes and maintains?

The most popular, with five out of nine interviewees mentioning it, strategy for ensuring maintainability was 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).

Q17. Provide instances where users have misunderstood the software. What, if any, actions were taken to address understandability issues?

The discussion with the developers focused on understandability issues for two classes of users: the end users and other developers. The threats to understandability for end users include users not understanding how to use features ([Because the users don't have sufficient background? Ask Ao. —SS]) [Did you mean medical or medical imaging background? No, it was majorly caused by the software usability. Based on the context of our interview conversations, it means that users don't understand how to complete certain operations or what functions a feature provides. It's due to unclear description, lack of documentation, poorly designed UI or nonintuitive process —AD], the team not having a dedicated user experience (UX) designer, some important indicators are not noticeable (e.g. a progress bar), not all users understand the purpose of the software, not all users know if the software includes certain features, not all users understand how to use the command line tool, not all users understand that the software is a web application. For developers the threats to understandability include developers not understanding how to deploy the software ([This sounds like a consequence of a lack of understandability, rather than threat to it. Or is the point that developers lack sufficient background? Ask Ao —SS]) [Yes, this should be a consequence of a lack of understandability. I think the direct cause is that the project doesn't clearly explain how to deploy. The background of the developers wasn't discussed at that point. —AD], and the architecture is difficult for new developers to understand.

The most common strategy, cited by 4 developers, for ensuring understandability was to use documentation (user manuals, mailing lists, forums). Other suggested and practiced strategies include a graphical user interface, testing every release with active users, making simple things simple and complicated things possible, icons with clear visual expressions, designing the software to be intuitive, having a UX designer with the right experience, dialog windows for important notifications, and providing an example for users to follow.

Q18. What, if any, actions were taken to address usability issues?

The two most frequently mentioned strategies (mentioned 3 times each) for improving usability are: i) usability tests and interviews with end users; and, ii) adjusting the software according to user feedback. Other ideas that came up include a straightforward and intuitively designed interface / professional UX designer, providing step-by-step

processes, making the basic functions easy to use without reading the documentation, focusing on limited number of functions, making the software more streamlined, downsampling images to consume less memory, and an option to load only part of the data to boost performance.

Q20. Do you have any concern that your computational results won't be reproducible in the future? Have you taken any steps to ensure reproducibility?

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, using different versions of some common libraries [because the interface and/or behaviour of the library could have changed? Ask Ao —SS] [Yes. Behaviour change could be caused by a lib changing to use different algorithms or different dependencies —AD], 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.

10. Lessons from MI Developers

The best practices from MI developers are taken to address the pain points mentioned in Section 9. We found the practices as part of the qualitative data from developer interviews (Section 3.4). The practices we summarize can potentially be emulated by the MI software packages that do not currently follow them. Moreover, these practices may also provide examples that can be followed by other research software domains.

11. Threats to Validity

Below we categorize and list the threats to validity that we have identified. The categories come from Ampatzoglou et al. (2019) and Zhou et al. (2016).

[observed artifacts (Section 6) - human judgement, not always given the expected name, but the content is there. —SS]

Construct Validity: "Defines how effectively a test or experiment measures up to its claims. This aspect deals with whether or not the researcher measures what is intended to be measured" (Ampatzoglou et al., 2019).

- For practical considerations the time spent measuring each package had to be limited. The time limit may have caused an assessment to have missed something something relevant.
- Our ranking is partly based on surface (shallow) measurement, which may not fully reveal the underlying qualities.
- The questions in the measurement template may not actually measure the qualities they are associated with. For instance, we have assumed that maintainability is improved if a high percentage of identified issues are closed, but it is possible for a project with a wealth of ideas to have many open issues, and still be maintainable.
- With the exception of the interview data for 8 projects, we collected all of the information for each project from the artifacts available on the Internet. In some cases this source of data may mean we did not find evidence of something, like unit testing, not because the project didn't do it, but because no artifacts of this activity remained in the publicly available repository.

- As mentioned in Section 9, one interviewee was too busy to participate in a full interview, so they provided a version of written answers to us. 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.
- As mentioned in Section 4.1, we could not install or build *dwv*, *GATE*, and *DICOM Viewer*. We used a deployed online version for *dwv*, a VM version for *GATE*, but no alternative for *DICOM Viewer*. We might underestimate their rank due to an uncommon technical issue.

Internal Validity: "This aspect relates to the examination of causal relations. Internal validity examines whether an experimental treatment/condition makes a difference or not, and whether there is evidence to support the claim" (Ampatzoglou et al., 2019).

- Many of the measurement template questions look for the presence of certain artifacts, like a user manual, or a getting started tutorial. We have implicitly assumed that the presence of these artifacts is an indication of a given quality, but this is an indirect measure. It may be possible to achieve qualities without the artifacts we sought. A direct measure of quality, like an usability experiment, would not have this problem.
- We compared our rankings to the rankings by the community and the Domain Expert, but we have assumed all qualities are equally weighted. The community and the Domain Expert likely have a more complex weighting between qualities.
- We assumed there was a casual relationship between the number of files and reusability, since we assumed multiple files implies modularity. Of course the code can be modular, and not divided into separate files. Moreover, having many files does not immediately in itself imply a decomposition based on sound design principles, like information hiding.

External Validity: "Define the domain to which a study's findings can be generalized" (Zhou et al., 2016).

- We interviewed eight teams, which is a good proportion of the 29. However, there is still a risk that the subset of the entire group does not represent the whole MI software community.
- We identified qualities that we believe the community will be interested in, and we weighted these qualities equally, but the qualities we chose, and their weighting, may not match the external reality.
- The number of GitHub stars, watches, and forks are not ideal measures of popularity. [Can we find a reference for this? —SS]

Conclusion Validity: "Demonstrate that the operations of a study such as the data collection procedure can be repeated, with the same results" (Zhou et al., 2016).

- The grading template included an entry for the reviewer's impression. We aimed for objectivity, but there is a risk that some scores may be subjective and biased.
- The measurements are made at an instant in time, and the instant differs by a few weeks between projects, due to the time needed to measure them. For the most part, the projects are living and continually changing, which means if we measured at a different time, our results may change.

12. Recommendations

This section presents our recommendations on MI software development. Although our focus is on MI software, unless noted otherwise, our recommendations apply to any scientific computing software. Section 12.1 discusses the actions that can potentially improve the ten software qualities. Sections 12.2, 12.3, and 12.4 are based on the primary pain points (from Section 9) collected from developers in the MI domain.

12.1. Recommendations on Improving Software Qualities

Based on our quality measurements in Section 4 and discussions with the developers in Section 9, we collected key points for improving software qualities. These points should be considered for new and existing projects, but we are not saying that every project should do everything on our list. All projects will have finite resources, so strategies will have to be selected that provide the greatest return on investment. Our specific recommendations for each quality are as follows:

- **Installability** (Section 4.1)
  - providing clear instructions;
  - automating installation;
  - including all dependencies in the installer;
  - avoiding heavily depending on other commercial products (e.g. Matlab);
  - potentially building a web application that needs no installation.
- **Correctness & Verifiability** (Section 4.2 and Section 9.1)
  - test-driven development with unit tests, integration tests, and nightly tests;
  - two stage development process with stable release & nightly builds;
  - CI/CD;
  - requirements specifications and theory manuals (Smith, 2016; Smith and Lai, 2005).
  - static code analysis tools (e.g. Lint and SonarQube)
- **Reliability** (Section 4.3)
  - test-driven development with unit tests, integration tests, and nightly tests.
  - two stage development process with stable release & nightly builds;
  - descriptive error messages.
- **Robustness** (Section 4.4)
  - designing with exception handling so the software can fail gracefully;
  - descriptive error messages.
- **Usability** (Section 4.5 and Section 9.1)
  - usability tests and interviews with end users;
  - adjusting according to users' feedbacks;
  - getting started tutorials;
  - user manuals;
  - professional UX (User eXperience) designs;
  - graphical user interface;
  - active supports to users.
- **Maintainability** (Section 4.6 and Section 9.1)
  - use GitHub (or equivalent version control and project management environment);
  - use a modular approach to design (we advocate basing the design on Parnas's principle of information hiding: "system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change." (Parnas et al., 2000))

  - documentation for developers: project plan, developer's manual, and API documentation.

- **Reusability** (Section 4.7)
  - modular approach;
  - API documentation;
  - tools that generate software documentation for developers (e.g. Doxygen, Javadoc, and Sphinx).

- **Understandability** (Section 4.8 and Section 9.1)
  - modular approach;
  - good coding style: consistent indentation and formatting style; consistent, distinctive, and meaningful code identifiers; keeping parameters in the same order for all functions; avoiding hard-coded constants (other than 0 and 1);
  - clear comments, indicating what is being done, not how;
  - description of algorithms used;
  - documentation of explicit requirements on coding standards;
  - communication between developers and users via GitHub issues, mailing lists, and forums.

- **Visibility/Transparency** (Section 4.9)
  - documents for the development process, contributor's guide, project status, development environment, and release notes.

- **Reproducibility** (Section 9.1)
  - test-driven development with unit tests, integration tests, and nightly tests.
  - open-source;
  - making data and documentation available;
  - using open-source libraries.

[Could "reverse" the list and list the recommendations and beside each the qualities that that recommendation helps with - this would reduce repetition. —SS]

12.2. Recommendations on Dealing With Limited Resources

The limitation of resources has many faces, with the key manifestations being lack of fundings, time, and developers. We summarized our discussion with the MI software developers in Section 9. Many of our recommendations involve investing more effort into processes, tools and methods, so that the limited resources can be used more effectively. Our recommendations are as follows:

[Talk about increasing the number of developers. CONTRIBUTING is rare (Table 6), but it doesn't have to be. Related to this information for new developers is rare: Code Style Guide, Requirements, Design Doc, API. —SS]

- **Identify the root cause.** More fundings or developers may not solve problems called by a lack of time. It is beneficial to identify the underlying obstacles to the team. [Is there something from the mythical man month that should be cited here? —SS]

- **Maintain good documentation.** Creating and updating documentation consumes time, but can save much more time in the long term. If the users and developers can find answers to their questions themselves, they are less likely to abuse the team's issue tracker. [If it is an option, we could cite one of our productivity related papers. —SS]

- **Adopt time-saving tools.** A good CI/CD tool (e.g. GitHub Actions) saves time for building and deploying the product, and automated tests can work in the background while developers are focusing on other tasks.

- **Use test-driven development process.** Many people think writing test cases is less rewarding than writing code, but without testing identifying and fixing bugs can consume substantial resources. Setting up the test cases costs time, but generates more benefits in the long run.
- **Consider supporting third-party plugins or extensions.** Why not let users share the burden? No software product can deliver every user's needs, and the large quantity of features leads to more bugs and maintenance problems. So it may be a good idea to shift some development and maintenance responsibilities to the users. The users may also be happy with the extra flexibility.
- **Consider "hibernating" for a while.** When there aren't enough developers, the team can shift from development mode toward maintenance mode. The team can stop building new features for a while and instead fix bugs and design problems from the past. If the development team can repay some of its technical debt (Kruchten et al., 2012), the software qualities will likely improve as a result.
- **Commercialization is not always toxic.** Licensing the software to commercial companies to use as internal modules of their products may bring financial support to the team. Meanwhile, the project can stay open-source for the community.

12.3. Recommendations on Choosing A Tech Stack

A tech stack refers to a set of technologies used by a team to build software and manage the project. Section 9 lists the advantages and disadvantages between native and web applications. We give further suggestions on the choice of a tech stack to improve the four priority factors identified by developers: compatibility, maintainability, performance and security. The suggestions are intended to provide ideas and avenues for exploration; not all of the suggestions will be the right fit for all projects and all teams.

- **Identify the priorities between the factors.** Simultaneously achieving high levels for all four factors (compatibility, maintainability, performance and security) is difficult. A team needs to prioritize its objectives according to its resource and experience.
- **Be open-minded about new technologies.** Web applications with only a frontend are known for worse performance 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.
- **Use git and GitHub.** As mentioned in Section 4.6, almost all of the 29 MI software projects used git, and the majority of them used GitHub. We found from the projects' websites and our interviews with developers that, some projects moved from other version control tools to git and GitHub. GitHub provides convenient repository and project management, and OSS projects tend to receive more attention and contributions on GitHub.
- **Web applications can also 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.
- **Backend servers can have low costs.** Serverless solutions from major cloud service providers may be worth exploring. Serverless still uses a server, but the team is only charged when they use it. The solution is event-driven, and costs the team by the number of requests it processes. Thus, serverless can be very cost-effective for less intensively used functions.
- **Web transmission may diminish security.** Transferring sensitive data online 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.
- **Maintain good documentation.** No matter what tech stack a team uses, a well-maintained project plan, developer's manual, and API documentation generally help team members to contribute more and make fewer mistakes.

12.4. Recommendations on Enriching the Testing Datasets

As described in Section 9, it is difficult for software development teams to access real-world medical imaging datasets. This problem restricts their capability and flexibility for testing. We provide some suggestions 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), and MedPix by National Library of Medicine (Smirniotopoulos, 2014). 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 OSS in the domain, and protects privacy.

13. Conclusions

We analyzed the state of the practice for the MI domain with the goal of understanding current practice, answering our six 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 4 lists our measurements to nine software qualities for the 29 projects, and Section 9 contains our interviews with eight of the 29 teams, discussing their development process and five software qualities. We answered our research questions. In addition, Section 12 presents our recommendations on SC software development.

13.1. Key Findings

With the measurement results in Section 4, we summarized the current status of MI software development. We ranked the 29 software projects in nine qualities. Based on on the grading scores *3D Slicer*, *ImageJ*, and *OHIF Viewer* are the top three software packages.

The interview results in Section 9 show some merits, drawbacks, and pain points within the development process. The three primary categories of pain points are:

- the lack of fundings and time;
- the difficulty to balance between four factors: cross-platform compatibility, convenience to development & maintenance, performance, and security;

- the lack of access to real-world datasets for testing.

We summarized the solutions from the developers to address these problems, including developing a web-based approach with backend servers and maintaining better documentation. We also collected the status of documentation. We found that for all 8 interviewed teams that documentation is felt to be vital to a project, with the most popular form of documentation being forum discussions and videos. With respect to project management almost all teams used GitHub and pull requests to manage contributions. Very few teams used a specific development model. It appears that the development process is more ad hoc than planned for the majority of projects.

Our answers to the research questions are based on the above findings. We identified the existing artifacts, tools, principles, processes, and methodologies in the 29 projects. By comparisons with the implied popularity of existing projects we found: 1) four of the top five software projects in our ranking were also among the top five ones receiving the most GitHub stars per year (Table 5); 2) three of the top four in our ranking were among the top four provided by the domain experts.

Section 12 presents our recommendations on improving software qualities and easing pain points during development. Some highlighted recommendations are as follows:

- adopting test-driven development with unit tests, integration tests, and nightly tests;
- maintaining good documentation (e.g., installation instructions, requirements specifications, theory manuals, getting started tutorials, user manuals, project plan, developer's manual, API documentation, requirements on coding standards, development process, project status, development environment, and release notes);
- using CI/CD;
- using git and GitHub;
- modular approach with the design principle proposed by Parnas et al. (2000);
- considering newer technologies (e.g. web application and serverless solution);
- various ways of enriching the testing datasets, such as using existing open data sources and establishing greater community collaboration in the MI domain (Section 12.4).

13.2. Future Works

With learnings from this project, we summarized recommendations for the future state of the practice assessments:

- we can make the 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 can design a list of operation steps, 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.
  - 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. Ideas for getting started are available in Smith et al. (2021).
  - 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.
- the rubric for the grading standard can be made more explicit.
- we can improve some interview questions. Some examples are:
  - in Q14, "Do you think improving this process can tackle the current problem?" is a yes-or-no question, which is not informative enough. As mentioned in Section 8.2, most interviewees ignored it. We can change it to "By improving this process, what current problems can be tackled?";
  - in Q16, 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?";
  - Q17 and Q18 should respectively ask understandability to developers and usability to end-users, since there was confusion during the interviews as to which group was being discussed.
- 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.

References

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].

Aysel Afsar. 2021. DICOM Viewer. https://github.com/ayselafsar/dicomviewer. [Online; accessed 27-May-2021].

J. Ahrens, Berk Geveci, and Charles Law. 2005. ParaView: An End-User Tool for Large Data Visualization. *Visualization Handbook* (01 2005).

Paulo Amorim, Thiago Franco de Moraes, Helio Pedrini, and Jorge Silva. 2015. InVesalius: An Interactive Rendering Framework for Health Care Support. 10. https://doi.org/10.1007/978-3-319-27857-5_5

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

Kevin Archie and Daniel Marcus. 2012. DicomBrowser: Software for Viewing and Modifying DICOM Metadata. *Journal of digital imaging : the official journal of the Society for Computer Applications in Radiology* 25 (02 2012), 635–45. https://doi.org/10.1007/s10278-012-9462-x

Medical Imaging Technology Association. 2021. About DICOM: Overview. https://www.dicomstandard.org/about-home. [Online; accessed 11-August-2021].

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]

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

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

Shekhar Chandra, Jason Dowling, Craig Engstrom, Ying Xia, Anthony Paproki, Ales Neubert, David Rivest-Hénault, Olivier Salvado, Stuart Crozier, and Jurgen Fripp. 2018. A lightweight rapid application development framework for biomedical image analysis. *Computer Methods and Programs in Biomedicine* 164 (07 2018). https://doi.org/10.1016/j.cmpb.2018.07.011

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

Ao Dong. 2021b. Software Quality Grades for MI Software. Mendeley Data, V1, doi: 10.17632/k3pcdvdzj2.1. https://doi.org/10.17632/k3pcdvdzj2.1

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

S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta. 2009. Scientific Computing's Productivity Gridlock: How Software Engineering Can Help. *Computing in Science Engineering* 11, 6 (Nov 2009), 30–39. https://doi.org/10.1109/MCSE.2009.205

Pierre Fillard, Nicolas Toussaint, and Xavier Pennec. 2012. Medinria: DT-MRI processing and visualization software. (04 2012).

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].

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

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

Daniel Haehn. 2013. Slice:drop: collaborative medical imaging in the browser. 1–1. https://doi.org/10.1145/2503541.2503645

Paul Hamill. 2004. *Unit test frameworks: Tools for high-quality software development*. O'Reilly Media.

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. 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

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 and David E. Bernholdt. 2018. Better (Small) Scientific Software Teams, tutorial in Argonne Training Program on Extreme-Scale Computing (ATPESC). https://press3.mcs.anl.gov//atpesc/files/2018/08/ATPESC_2018_Track-6_3_8-8_1030am_Bernholdt-Better_Scientific_Software_Teams.pdf. https://doi.org/articles/journal_contribution/ATPESC_Software_Productivity_03_Better_Small_Scientific_Software_Teams/6941438

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.

horosproject.org. 2020. Horos. https://github.com/horosproject/horos. [Online; accessed 27-May-2021].

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

Parallax Innovations. 2020. Microview. https://github.com/parallaxinnovations/MicroView/. [Online; accessed 27-May-2021].

Software Sustainability Institute. 2022. Online sustainability evaluation. https://www.software.ac.uk/resources/online-sustainability-evaluation.

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.

Sama Jan, Giovanni Santin, Daniel Strul, S Staelens, K Assié, Damien Autret, Stéphane Avner, Remi Barbier, Manuel Bardiès, Peter Bloomfield, David Brasse, Vincent Breton, Peter Bruyndonckx, Irene Buvat, AF Chatziioannou, Yunsung Choi, YH Chung, Claude Comtat, Denise Donnarieix, and Christian Morel. 2004. GATE: a simulation toolkit for PET and SPECT. *Physics in medicine and biology* 49 (11 2004), 4543–61. https://doi.org/10.1088/0031-9155/49/19/007

Arne N. Johanson and Wilhelm Hasselbring. 2018. Software Engineering for Computational Science: Past, Present, Future. *Computing in Science*

& Engineering Accepted (2018), 1–31.

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.

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

Ron Kikinis, Steve Pieper, and Kirby Vosburgh. 2014. *3D Slicer: A Platform for Subject-Specific Image Analysis, Visualization, and Clinical Support*. Vol. 3. 277–289. https://doi.org/10.1007/978-1-4614-7657-3_19

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.

Chris Rorden's Lab. 2021. MRIcroGL. https://github.com/rordenlab/MRIcroGL. [Online; accessed 27-May-2021].

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.

Ajay Limaye. 2012. Drishti, A Volume Exploration and Presentation Tool. *Proc SPIE* 8506, 85060X. https://doi.org/10.1117/12.935640

Fang Liu, Julia Velikina, Walter Block, Richard Kijowski, and Alexey Samsonov. 2016. Fast Realistic MRI Simulations Based on Generalized Multi-Pool Exchange Tissue Model. *IEEE Transactions on Medical Imaging* PP (10 2016), 1–1. https://doi.org/10.1109/TMI.2016.2620961

Andy Loening. 2017. AMIDE. https://sourceforge.net/p/amide/code/ci/default/tree/amide-current/. [Online; accessed 27-May-2021].

Yves Martelli. 2021. dwv. https://github.com/ivmartel/dwv. [Online; accessed 27-May-2021].

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.

D Nevcas and P Klapetek. 2012. Gwyddion: an open-source software for spm data analysis. *Cent Eur J Phys* 10 (01 2012).

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

E Nolf, Tony Voet, Filip Jacobs, R Dierckx, and Ignace Lemahieu. 2003. (X)MedCon * An OpenSource Medical Image Conversion Toolkit. *European Journal of Nuclear Medicine and Molecular Imaging* 30 (08 2003), S246. https://doi.org/10.1007/s00259-003-1284-0

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

Oluwaseun Owojaiye, W. Spencer Smith, Jacques Carette, Peter Michalski, and Ao Dong. 2021. State of Sustainability for Research Software (poster). In *SIAM-CSE 2021 Conference on Computational Science and Engineering, Minisymposterium: Software Productivity and Sustainability for CSE*. https://doi.org/10.6084/m9.figshare.14039888.v2

A. Panchal and R. Keyes. 2010. SU-GG-T-260: Dicompyler: An Open Source Radiation Therapy Research Platform with a Plugin Architecture. *Medical Physics - MED PHYS* 37 (06 2010). https://doi.org/10.1118/1.3468652

Xenophon Papademetris, Marcel Jackowski, Nallakkandi Rajeevan, Robert Constable, and Lawrence Staib. 2005. BioImage Suite: An integrated medical image analysis suite. 1 (01 2005).

David Parnas, Systems Branch, Washington C, P. Clements, and David Weiss. 2000. The Modular Structure of Complex Systems. (09 2000).

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

Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2018. Categorizing the Content of GitHub README Files. arXiv:1802.06997 [cs.SE]

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].

UTHSCSA Research Imaging Institute. 2019. Papaya. https://github.com/rii-mango/Papaya. [Online; accessed 27-May-2021].

Nicolas Roduit. 2021. Weasis. https://github.com/nroduit/nroduit.github.io. [Online; accessed 27-May-2021].

Curtis Rueden, Johannes Schindelin, Mark Hiner, Barry Dezonia, Alison Walter, and Kevin Eliceiri. 2017. ImageJ2: ImageJ for the next generation of scientific image data. *BMC Bioinformatics* 18 (11 2017). https://doi.org/10.1186/s12859-017-1934-z

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.

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].
Rebecca Sanders and Diane Kelly. 2008. Dealing with Risk in Scientific Software Development. *IEEE Software* 4 (July/August 2008), 21–28.
Pixmeo SARL. 2019. OsiriX Lite. https://github.com/pixmeo/osirix. [Online; accessed 27-May-2021].
Johannes Schindelin, Ignacio Arganda-Carreras, Erwin Frise, Verena Kaynig, Mark Longair, Tobias Pietzsch, Stephan Preibisch, Curtis Rueden, Stephan Saalfeld, Benjamin Schmid, Jean-Yves Tinevez, Daniel White, Volker Hartenstein, Kevin Eliceiri, Pavel Tomancak, and Albert Cardona. 2012. Fiji: An Open-Source Platform for Biological-Image Analysis. *Nature methods* 9 (06 2012), 676–82. https://doi.org/10.1038/nmeth.2019
Tobias Schlauch, Michael Meinel, and Carina Haupt. 2018. DLR Software Engineering Guidelines. https://doi.org/10.5281/zenodo.1344612
Will Schroeder, Bill Lorensen, and Ken Martin. 2006. *The visualization toolkit*. Kitware.
Judith Segal and Chris Morris. 2008. Developing Scientific Software. *IEEE Software* 25, 4 (July/August 2008), 18–20.
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 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, 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. 2018d. 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, 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 and Nirmitha Koothoor. 2016. A Document-Driven Method for Certifying Scientific Computing Software for Use in Nuclear Safety Analysis. *Nuclear Engineering and Technology* 48, 2 (April 2016), 404–418. https://doi.org/10.1016/j.net.2015.11.008
W. Spencer Smith and Lei Lai. 2005. A New Requirements Template for Scientific Computing. In *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, J. Ralyté, P. Ågerfalk, and N. Kraiem (Eds.). In conjunction with 13th IEEE International Requirements Engineering Conference, Paris, France, 107–121.
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.
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
Keenan Szulik. 2017. Don't judge a project by its GitHub stars alone. https://blog.tidelift.com/dont-judge-a-project-by-its-github-stars-alone.
TESCAN. 2020. 3DimViewer. https://bitbucket.org/3dimlab/3dimviewer/src/master/. [Online; accessed 27-May-2021].
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.
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).
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 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, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2016. Good Enough Practices in Scientific Computing. *CoRR* abs/1609.00037 (2016). http://arxiv.org/abs/1609.00037
Gert Wollny. 2020. Ginkgo CADx. https://github.com/gerddie/ginkgocadx. [Online; accessed 27-May-2021].
Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. 2006. User-Guided 3D Active Contour Segmentation of Anatomical Structures: Significantly Improved Efficiency and Reliability. *Neuroimage* 31, 3 (2006),

1116–1128.

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

Xin Zhou, Yuqin Jin, He Zhang, Shanshan Li, and Xin Huang. 2016. A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering. 153–160. https://doi.org/10.1109/APSEC.2016.031

Erik Ziegler, Trinity Urban, Danny Brown, James Petts, Steve D. Pieper, Rob Lewis, Chris Hafey, and Gordon J. Harris. 2020. Open Health Imaging Foundation Viewer: An Extensible Open-Source Framework for Building Web-Based Imaging Applications to Support Cancer Research. *JCO Clinical Cancer Informatics* 4 (2020), 336–345. https://doi.org/10.1200/CCI.19.00131 arXiv:https://doi.org/10.1200/CCI.19.00131 PMID: 32324447.