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

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

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

May 27, 2024

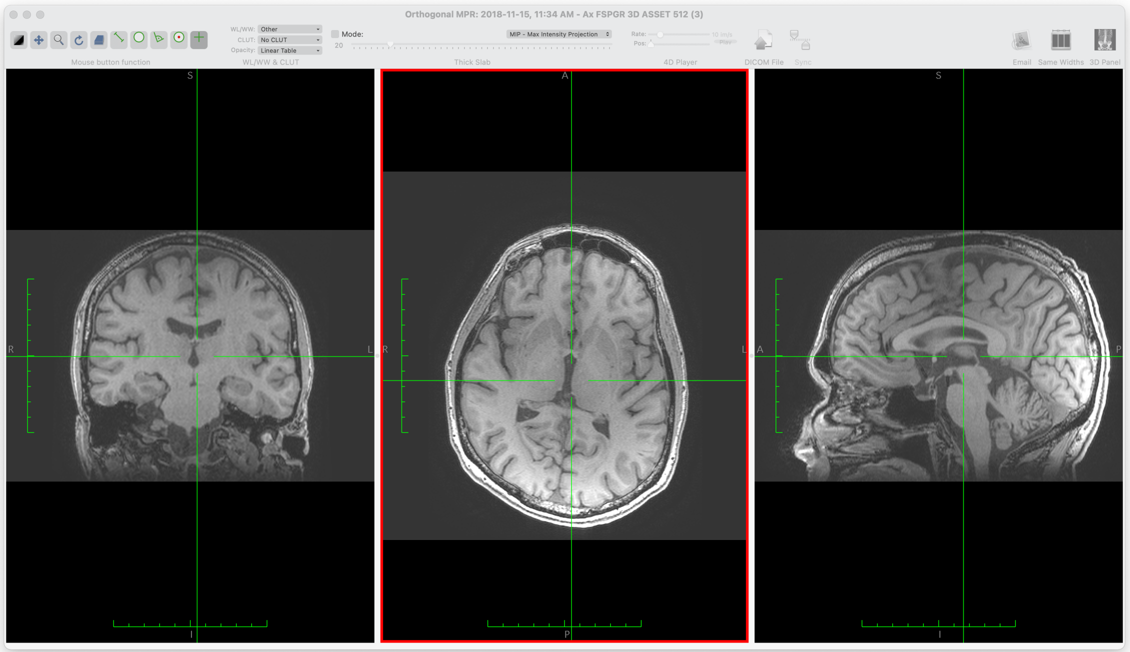
Abstract

We present the state of the practice for Medical Imaging (MI) software based on data available in open source repositories. We selected 29 medical imaging projects from 48 candidates and assessed 9 software qualities (installability, correctness/ verifiability, reliability, robustness, usability, maintainability, reusability, understandability, and visibility/transparency) by answering 108 questions for each software project. Based on the quantitative data, we ranked the MI software with the Analytic Hierarchy Process (AHP). The top four software products are *3D Slicer*, *ImageJ*, *Fiji*, and *OHIF Viewer*. Our ranking is mostly consistent with the community's ranking, with four of our top five projects also appearing in the top five of a list ordered by stars-per-year. We observed 88% of the documentation artifacts recommended by research software development guidelines. However, the current state of the practice deviates from the existing guidelines because of the rarity of some recommended artifacts (like test plans, requirements specification, code of conduct, code style guidelines, product roadmaps, and Application Program Interface (API) documentation).

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

1 Introduction

We aim to study the state of software development practice for Medical Imaging (MI) software using data available in open source repositories. 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, 2021b; Zhang et al., 2008). Figure 1, which shows an image of the brain, highlights the importance and value of MI. Through MI medical practitioners and researchers can noninvasively gain insights into the human body, including information on injuries and illnesses. Given the importance of MI software and the high number of competing software projects, we wish to understand the merits and drawbacks of the current development processes, tools, and methodologies. We aim to assess through a software engineering lens the quality of the existing software with the hope of highlighting standout examples, and providing guidelines and recommendations for future development.



Orthogonal MPR: 2018-11-15, 11:34 AM - Ax FSPGR 3D ASSET 512 (3)

WL/WW: Other
CLUT: No CLUT
Opacity: Linear Table
Mode: MIP - Max Intensity Projection
20

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

1.1 Research Questions

Not only do we wish to gain insight into the state of the practice for MI software, we also wish to understand the development of research software in general. We wish to understand the impact of the often cited gap, or chasm, between software engineering and research software (Kelly, 2007; Storer, 2017). Although scientists spend a substantial proportion of their working hours on software development (Hannay et al., 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 insight, we devised 10 research questions, which can be applied to MI, as well as to other domains, of research software (Smith and Michalski, 2022; Smith et al., 2021). We designed the questions to learn about the community's interest in, and experience with, software artifacts, tools, principles, processes, methodologies, and qualities. When we mention artifacts we mean the documents, scripts and code that constitutes a software development project. Example artifacts include requirements, specifications, user manuals, unit tests, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code. Once we have learned what MI developers do, we then put this information in context by contrasting MI software against the trends shown by developers in other research software communities.

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

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

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, 2021a) and Feature Extraction, Classification, and Interpretation (Kim et al., 2011). Software that provides MI support functions is also out of scope; therefore, we have not assessed the toolkit libraries VTK (Schroeder et al., 2006) and ITK (McCormick et al., 2014). Finally, Picture Archiving and Communication System (PACS), which helps users to economically store and conveniently access images (Choplin et al., 1992), are considered out of scope.

1.3 Methodology Overview

We designed a general methodology to assess the state of the practice for research software (Smith and Michalski, 2022; Smith et al., 2021). Details can be found in Section 3. Our methodology has been applied to MI software (Dong, 2021a) and Lattice Boltzmann Solvers (Michalski, 2021; Smith et al., 2024). This methodology builds off prior work to assess the state of the practice for such domains as Geographic Information Systems (Smith et al., 2018b), Mesh Generators (Smith et al., 2016b), Seismology software (Smith et al., 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. We consider a person month as 20 working days (4 weeks in a month, with 5 days of work per week) at 8 person hours per day, or $20 \times 8 = 160$ person hours.

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

2 Background

To measure the existing MI software we need the definitions of the software qualities that we will be assessing (Section 2.1). 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 2.2).

2.1 Software Quality Definitions

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

- **Installability** The effort required for the installation and/or uninstallation of software in a specified environment (ISO/IEC, 2011; Lenhard et al., 2013).
- **Correctness & Verifiability** A program is correct if it matches its specification (Ghezzi et al., 2003, p. 17). The specification can either be explicitly or implicitly stated. The related quality of verifiability is the ease with which the software components or the integrated product can be checked to demonstrate its correctness.
- **Reliability** The probability of failure-free operation of a computer program in a specified environment for a specified time (Musa et al., 1987), (Ghezzi et al., 2003, p. 357).
- **Robustness** Software possesses the characteristic of robustness if it behaves "reasonably" in two situations: i) when it encounters circumstances not anticipated in the requirements specification, and ii) when users violate the assumptions in its requirements specification (Ghezzi et al., 2003, p. 19), (Boehm, 2007).

- **Usability** "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" (ISO/TR, 2002, 2018).
- **Maintainability** The effort with which a software system or component can be modified to i) correct faults; ii) improve performance or other attributes; iii) satisfy new requirements (Boehm, 2007; IEEE, 1991).
- **Reusability** "The extent to which a software component can be used with or without adaptation in a problem solution other than the one for which it was originally developed" (Kalagiakos, 2003).
- **Understandability** "The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use" (ISO, 2001).
- **Visibility/Transparency** The extent to which all the steps of a software development process and the current status of it are conveyed clearly (Ghezzi et al., 2003, p. 32).

2.2 Analytic Hierarchy Process (AHP)

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

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

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

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

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

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

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

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

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

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

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

Table 1: Sample AHP Calculations for the Quality of Usability

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

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

3 Methodology

We developed a methodology for evaluating the state of the practice of research software (Smith and Michalski, 2022; Smith et al., 2021). The methodology can be instantiated for a specific domain of scientific software, which in the current case is medical imaging software for visualization. Our methodology involves and engages a domain expert partner throughout, as discussed in Section 3.1. The three 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. Answer the research questions (as given in Section 1.1).

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

3.1 Interaction With Domain Expert

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

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

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

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

3.2 List of Representative Software

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

We initially identified 48 MI candidate software projects from the literature (Björn, 2017; Brühschwein et al., 2019; Haak et al., 2015), on-line articles (Emms, 2019; Hasan, 2020; Mu, 2019), and forum discussions (Samala, 2014). The full list of 48 packages is available in Dong (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:

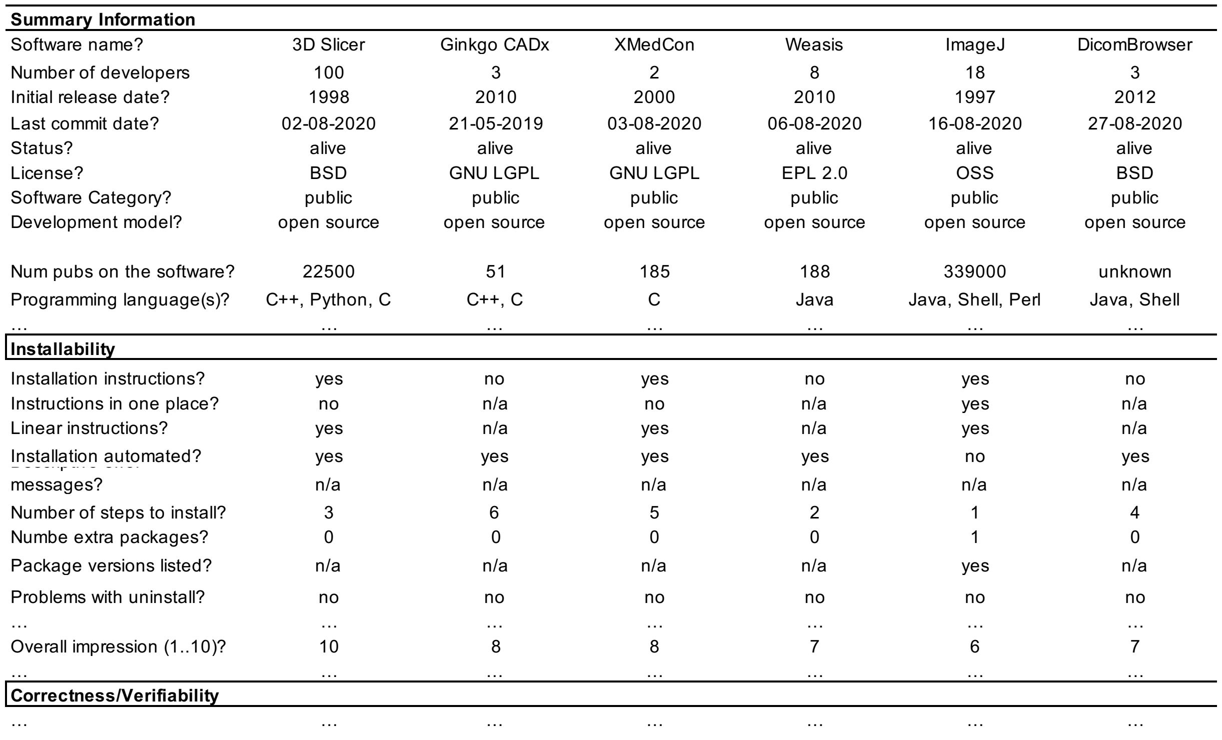
1. We removed the packages that did not have source code available, such as *MicroDicom*, *Aliza*, and *jivex*.
2. We focused on the MI software that provides visualization functions, as described in Section 1.2. Furthermore, we removed seven packages that were toolkits or libraries, such as *VTK*, *ITK*, and *dcm4che*. We removed another three that were for PACS.
3. We removed *Open Dicom Viewer*, since it has not received any updates in a long time (since 2011).

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

3.3 Grading Software

We grade the selected software using the measurement template summarized in Smith et al. (2021). The template provides measures of the qualities listed in Section 2.1. For each software package, we fill in the template questions. To stay within the target of 160 person hours to measure the domain, we allocated between one and four hours for each package. Project developers can be contacted for help regarding installation, if necessary, but we impose a cap of about two hours on the installation process, to keep the overall measurement time feasible. Figure 2 shows an excerpt of the spreadsheet. The spreadsheet includes a column for each measured software package.

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



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

public use. Private means software aimed only at a specific group, while the concept category is for software written simply to demonstrate algorithms or concepts. The three categories of development models are (open source, free-ware and commercial) are discussed in Section **??**. Information in the summary section sets the context for the project, but it does not directly affect the grading scores.

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

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

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

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

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

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

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

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

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

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

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

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

For projects on GitHub we manually collect additional information, such as the numbers of stars, forks, people watching this repository, open pull requests, closed pull requests, and the number of months a repository has been on GitHub. We need to take care with the project creation date, since a repository can have a creation date much earlier than the first day on GitHub. For example, the developers created the git repository for *3D Slicer* in 2002, but did not upload a copy of it to GitHub until 2020. Some GitHub data can be found using its GitHub Application Program Interface (API) via the following url: *https://api.github.com/repos/[owner]/[repository]* where [owner] and [repository] are replaced by the repo specific values. The number of months a repository has been on GitHub helps us understand the average change of metrics over time, like the average new stars per month.

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

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

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

4 Measurement Results

Table 2 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 "?". The date of the last update is the date of the latest update, at the time of measurement. 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).

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).

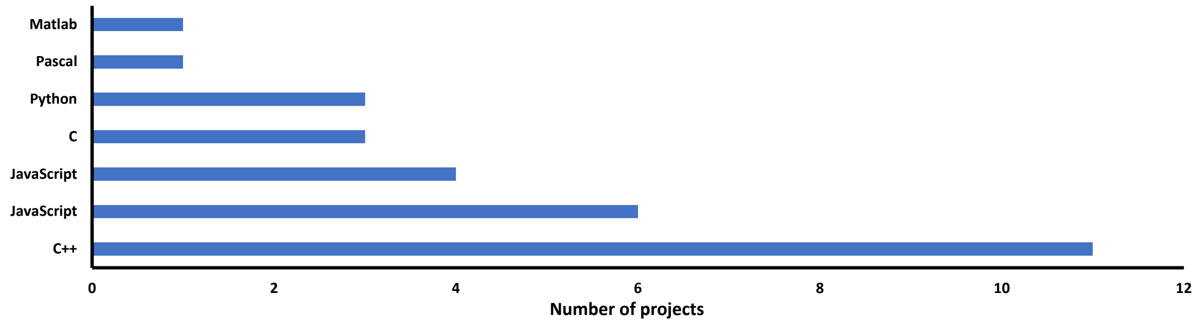


Figure 3: Primary languages versus number of projects

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

Table 2: Final software list (sorted in descending order of the number of Lines Of Code (LOC))

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 10 of the projects required extra dependencies. Five of them are web applications (as shown in Table 2) 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.

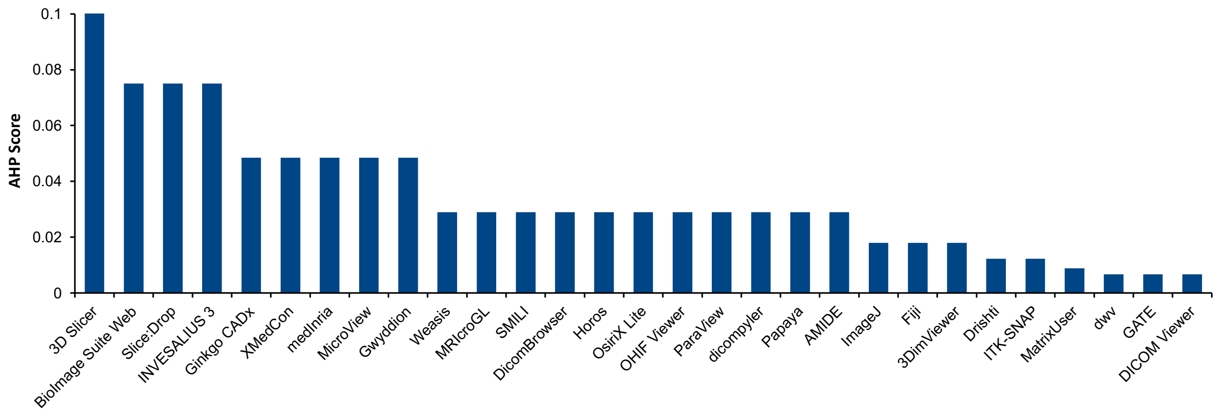


Figure 4: AHP installability scores

3D Slicer has the highest score because it had easy to follow installation instructions, and an automated, fast, frustration-free installation process. The installer added all dependencies automatically and no errors occurred during the installation and uninstallation steps. Many other software packages also had installation instructions and automated installers. We had no trouble installing the following packages: *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.

GATE, *dwv*, 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 on-line 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 depends 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

Figure 5 shows the scores of correctness and verifiability. 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*.

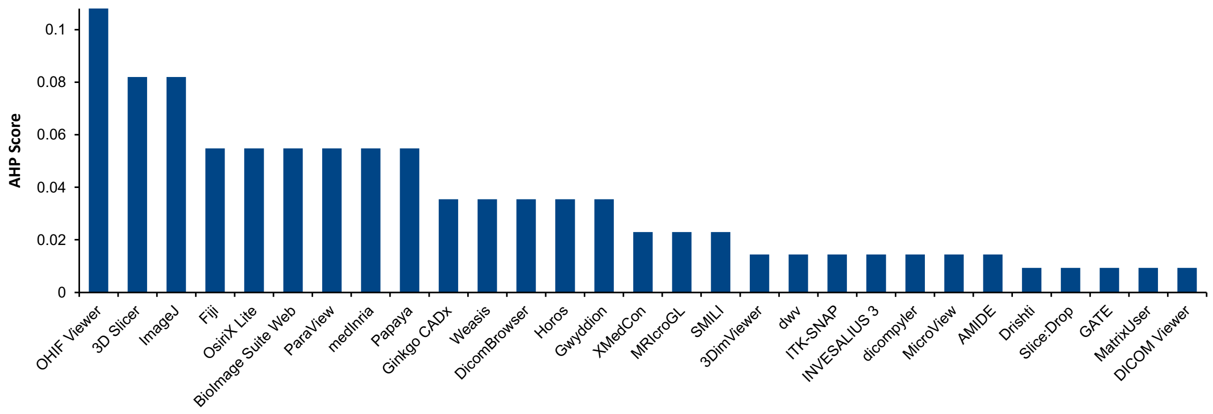


Figure 5: AHP correctness & verifiability scores

Even for some projects with well-organized documentation, 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 requirements-related 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. We did not find any problems with the on-line 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

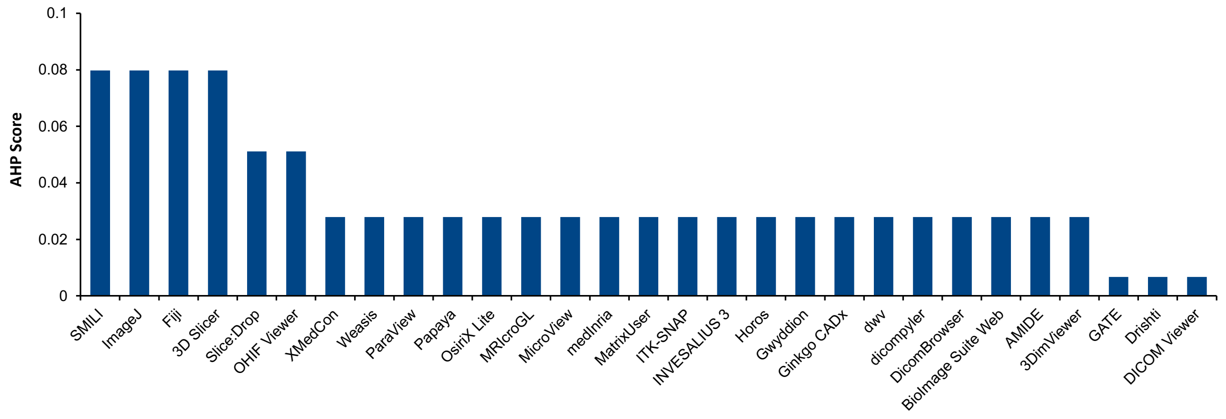


Figure 6: AHP surface reliability scores

underestimated the score of *OHIF Viewer*, since we needed further customization to load data.

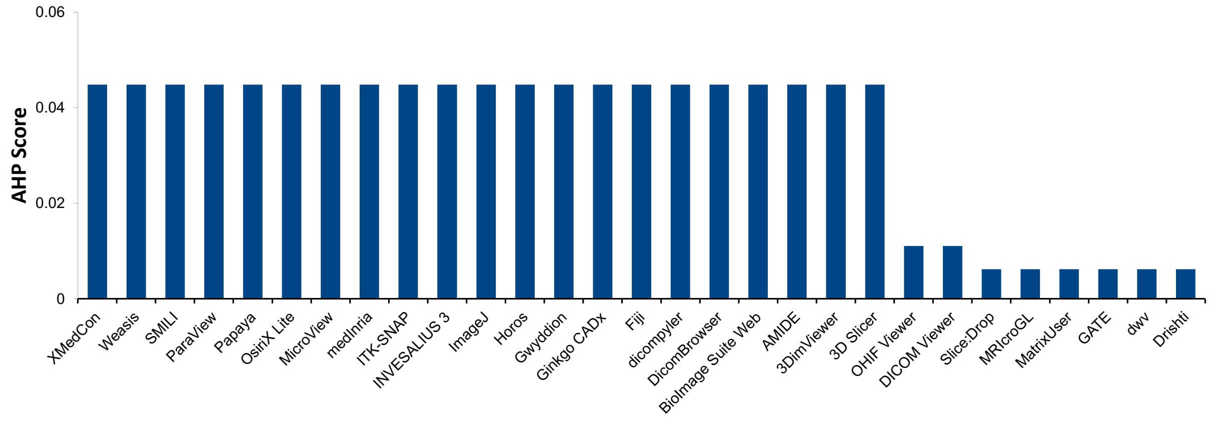


Figure 7: AHP surface robustness scores

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.

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 numerous 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.

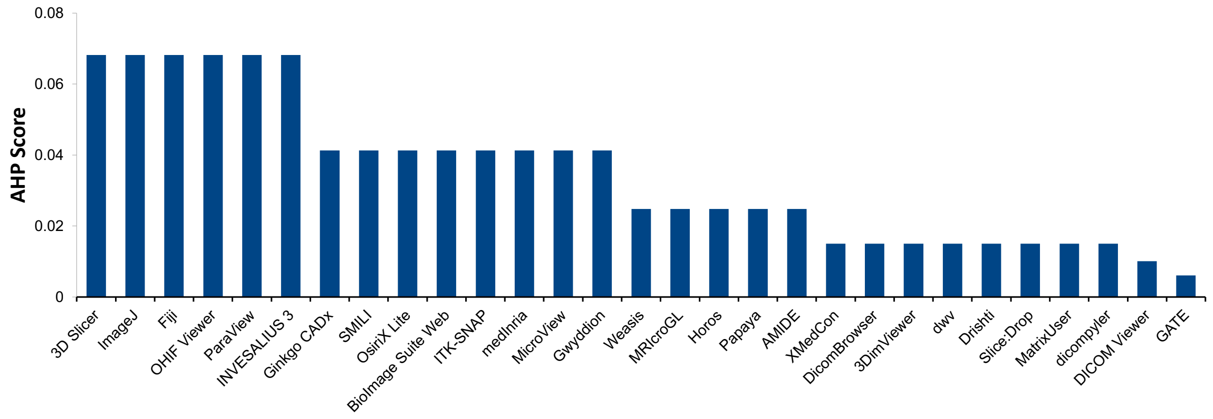


Figure 8: AHP surface usability scores

4.6 Maintainability

Figure 9 shows the ranking results for maintainability. We gave *3D Slicer* 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 (92%) compared to *ImageJ* (52%) and *Fiji* (64%). Table 3 shows which projects had these documents, in descending order of their maintainability scores.

Twenty-seven 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.

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 3, seven projects had API documents. We also assumed that projects with more code files and less LOC per code file are more reusable. Table 4 shows the number of text-based files by project, 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.

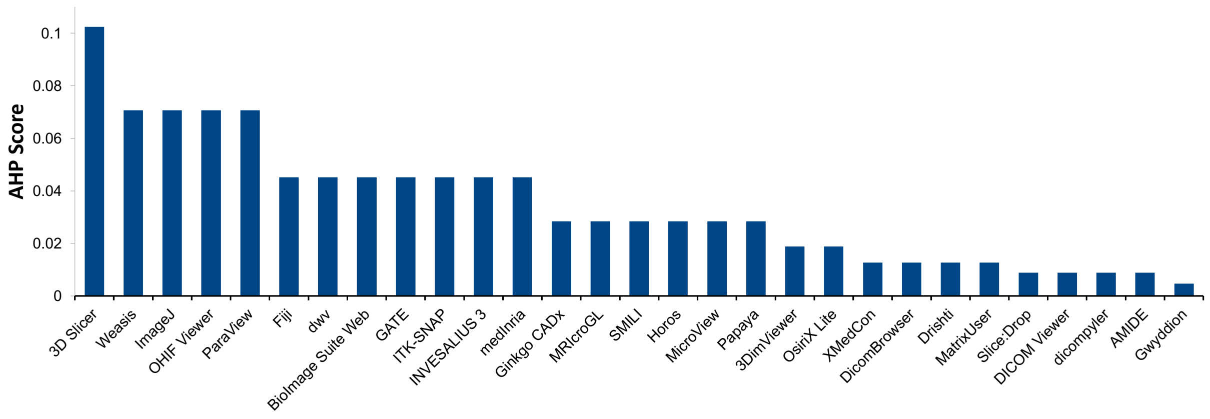


Figure 9: AHP maintainability scores

Software	Prod. Roadmap	Dev. Manual	API Doc.
3D Slicer	✓	✓	✓
ImageJ	✓	✓	✓
Weasis		✓	
OHIF Viewer		✓	✓
Fiji	✓	✓	✓
ParaView	✓		
SMILI			✓
medInria		✓	
INVESALIUS 3	✓		
dwv			✓
BioImage Suite Web		✓	
Gwyddion		✓	✓

Table 3: Software with the maintainability documents (listed in descending order of maintainability score)

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; modularized code; and, clear comments, indicating what is 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*.

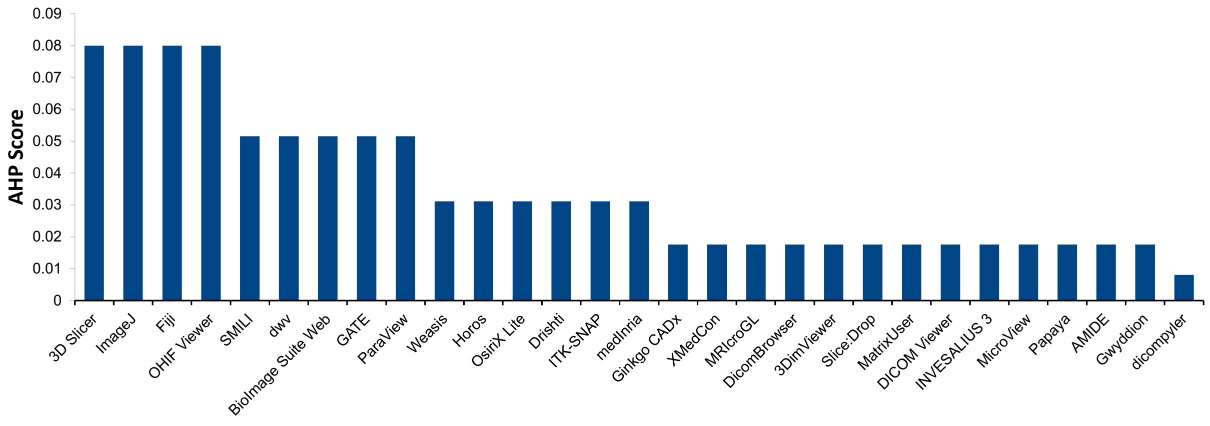


Figure 10: AHP reusability scores

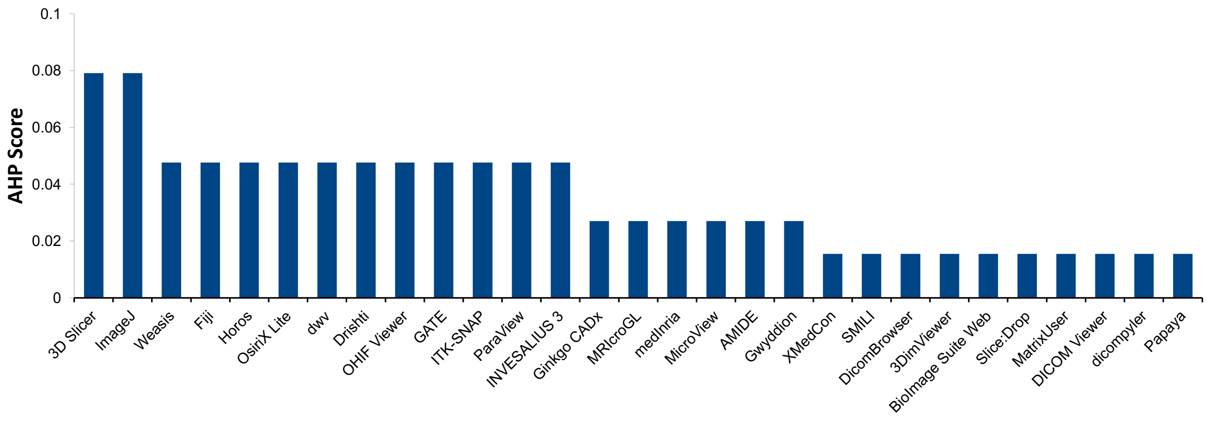


Figure 11: AHP surface understandability scores

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 5 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 2.2, 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 1.0.

The top four software products *3D Slicer*, *ImageJ*, *Fiji*, and *OHIF Viewer* have higher 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 4: Number of files and lines (sorted in descending order of reusability scores)

in most criteria. *3D Slicer* has a score in the top two for all qualities; *ImageJ* ranks near the top for all qualities, except for correctness & verifiability. *OHIF Viewer* and *Fiji* have similar overall scores, with *Fiji* doing better in installability and *OHIF Viewer* doing better in correctness & verifiability. 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.

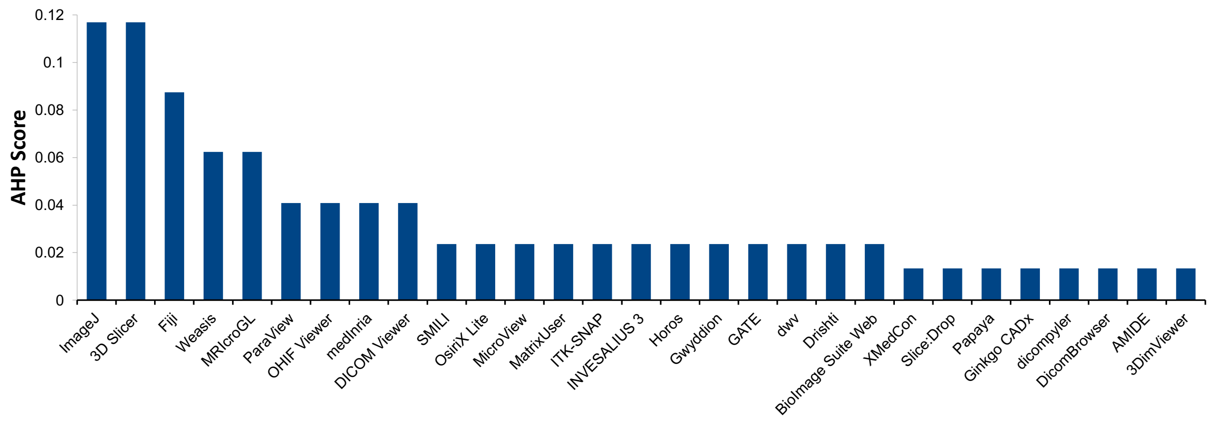


Figure 12: AHP visibility/transparency scores

Software	Dev. Process	Proj. Status	Dev. Env.	Rls. Notes
3D Slicer	✓	✓	✓	✓
ImageJ	✓	✓	✓	✓
Fiji	✓	✓	✓	
MRIcroGL				✓
Weasis			✓	✓
ParaView		✓		
OHIF Viewer			✓	✓
DICOM Viewer			✓	✓
medInria			✓	✓
SMILI				✓
Drishti				✓
INVESALIUS 3				✓
OsiriX Lite				✓
GATE				✓
MicroView				✓
MatrixUser				✓
BioImage Suite Web			✓	
ITK-SNAP				✓
Horos				✓
dwv				✓
Gwyddion				✓

Table 5: Software with visibility/transparency related documents (listed in descending order of visibility/transparency score)

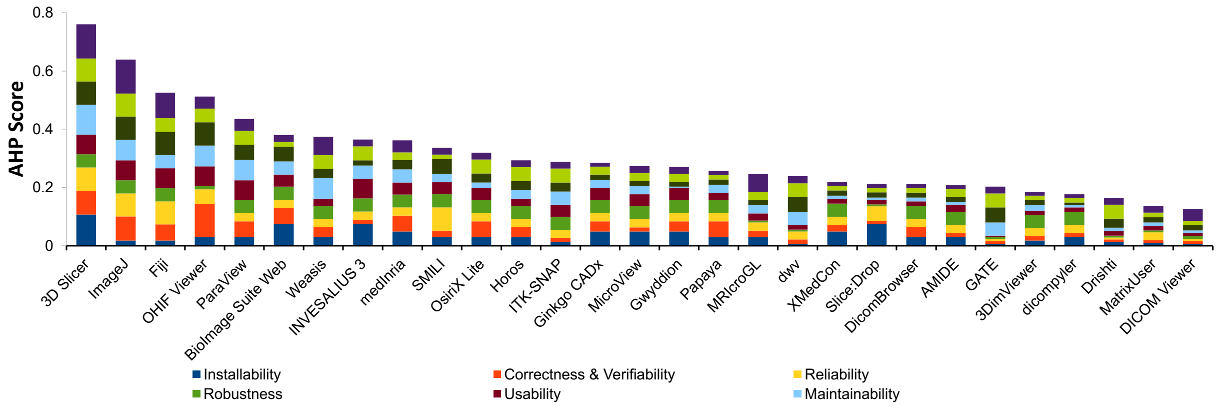


Figure 13: Overall AHP scores with an equal weighting for all 9 software qualities

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).

Table 6 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. Section 3.3 describes the method of getting the creation date. The items in Table 6 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. We collected all GitHub statistics 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 five packages are scattered among the first eight 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.

Further reason for discrepancies between our ranking and the community's ranking is that we weighted all qualities equally. This is not likely how users implicitly rank the different qualities. 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 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 GitHub 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, our methodology weights all qualities equally, but that may not be the case for the Domain Expert's impressions. Moreover, although the Domain Expert has significant experience with MI software, they have not used all 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 7 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). Dong (2021b) summarizes the full measurements. Tables 3 and 5 show the details on which projects use which types of artifacts for documents related to maintainability and visibility,

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

Table 6: 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)

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 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 8 (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),

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

- 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 8 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 8 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 a guideline does not explicitly recommend an artifact, does not mean the guideline authors do not value that artifact. They may have excluded it because it is out of the scope of their recommendations, or outside 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 they need separate uninstall instructions.

	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 8: 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 7 do not appear in the software development guidelines shown in Table 8: Troubleshooting guide and Developer's manual. Although the guidelines do not name these two artifacts, 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 8), we did not observe three recommended artifacts: i) Uninstall, 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 are not generally concerned with uninstall. Moreover, as mentioned above, uninstall is not particularly emphasized in existing recommendations.

Test plans describe the scope, approach, resources, and the schedule of planned test activities (van Vliet, 2000). The plan should cover details such as the makeup of the testing team, automated testing tools and technology to employ, the testing process, system tests, integration tests and unit tests. We did not observe test plans for MI software, but that doesn't mean plans weren't created; it means that the plans are not under version control. Test plans would have to at least be implicitly created, since we observed test cases with reasonable frequency for MI software (test cases are categorized as uncommon).

The other apparently neglected document is the requirements specification, which records the functionalities, expected performance, goals, context, design constraints, external interfaces and other quality attributes of the software (IEEE, 1998). If developers write requirements they typically are based on a template, which provide documentation structure, guidelines, and rules. Although there is no universally accepted template, examples include ESA (1991), IEEE (1998), NASA (1989), and Robertson and Robertson (1999).

MI software is like other research software in its neglect of requirements documentation. Although requirements documentation is recommended by some (Heroux et al., 2008; Schlauch et al., 2018; 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 if we drop the insistence on "up-front" requirements, allowing instead for the requirements to be written iteratively and incrementally, requirements are feasible (Smith, 2016). Smith et al. (2007)

provides a requirements template tailored to research software.

Table 8 shows several recommended artifacts that are rarely observed in practice. A theme among these rare artifacts is that, except for the user-focused getting started manual, they are developer-focused. The dependency list, which is a list of software library dependencies, was rarely observed, but this information is still likely present, just embedded in build scripts. The other developer-focused and rare artifacts are as follows:

- **A Contributing file** provides new contributors with the information that they need to start adding/modifying the repository's files. Abdalla (2016) provides a simple template for creating an open-source contributor guideline.
- **A Developer Code of Conduct** explicitly states the expectations for developers on how they should treat one another (Tourani et al., 2017). The code outlines rules for communication and establishes enforcement mechanisms for violations. As Tourani et al. (2017) states, the developer code documents the spirit of a community so that anyone can comfortably contribute regardless of ethnicity, gender, or sexual orientation. Three popular codes of conduct are (Tourani et al., 2017): Contributor Covenant, Ubuntu Code of Conduct, and Django Code of Conduct. A code of conduct can improve inclusivity, which brings the benefit of a wider pool of contributors. For example, a code of conduct can improve the participation of women (Singh et al., 2021). A standard of ethical behaviour can be captured in the code, for projects that are looking to abide by a code of ethics, such as the IEEE Code of Ethics (IEEE-CS/ACM and Practices, 1999), or the Professional Engineers of Ontario code of ethics (Professional Engineers Act, 2021, p. 23–24).
- **Code Style Guidelines** present standards for writing code. Style guides codify such elements as formatting, commenting, naming identifiers, best practices and dangers to avoid (Carty, 2020). For instance, most coding style guides will specify using ALLCAPS when naming symbolic constants. Understandability improves under standardization, since developers spend more time on the content of the code, and less time distracted by its style. Three sample style guides are: Google Java Style Guide, PEP8 Style Guide for Python, and Google C++Style Guide. Linting tools, like flake8 for Python, can be used to enforced coding styles, like the PEP8 standard.
- **A Product Roadmap** explains the vision and direction for a product offering (Münch et al., 2019). Although they have different forms, all roadmaps cover the following: i) Where are we now?, ii) Where are we going?, and iii) How can we get there? (Phaal et al., 2005). As stated by Pichler (2012), a product roadmap provides the following benefits: continuity of purpose, facilitation of collaboration, and assistance with prioritization. Creating a roadmap involves the following steps: i) define and outline a strategic mission and product vision, ii) scan the environment, iii) revise and distill the product vision to write the product roadmap, and iv) estimate the product life cycle and evaluate the mix of planned development efforts (Vähäniitty et al., 2002).
- **Design Documentation** explicitly states the design goals and priorities, records the likely changes, decomposes the complex system into separate modules, and specifies the relation-

ship between the modules. Design documentation shows the syntax of the interface for the modules, and in some cases also documents the semantics. Some potential elements of design documentation include the following:

  - Representation of the system design and class design using Unified Modelling Language class diagrams. This approach is suited to object-oriented design and designs that use patterns (Gamma et al., 1995).
  - Rigorous documentation of the system design following the template for a Module Guide (MG) (Parnas et al., 1984). An MG organizes the modules in a hierarchy by their secrets.
  - An explanation of the design using data flow diagrams to show typical use cases for input transformation.
  - A table or graph showing the traceability between the requirements and the modules (or classes)
  - The syntax of the modules (or classes) by providing lists of the state variables, exported constants and all exported access programs for each module (or class). This shows the interface that can used to access each module's services.
  - A formal specification of the semantics of input/output relationships and state transitions for each module using a Module Interface Specification (MIS) (Hoffman and Strooper, 1995). An MIS is an abstract model that formally shows each module's access programs and the associated transitions and outputs based on their state, environment, and input variables. ElSheikh et al. (2004); Smith and Yu (2009) show the example of an MIS for a mesh generator.

- **API Documentation** shows developers the services or data provided by the software application (or library) through such resources as its methods or objects (Meng et al., 2018). Understandability is improved by API documentation (Meng et al., 2018). API documentation can be generated using tools like Doxygen, pydoc, and javadoc.

The rare artifacts for MI software are similar to the rare artifacts for Lattice Boltzmann solvers (Michalski, 2021), except LBM software is more likely to have developer related artifacts, like Contributing, Dependency list, and Design documentation.

To improve MI software in the future, an increased use of checklists could help. Developers can use checklists to ensure they follow best practices. 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 (Institute, 2022; Thiel, 2020). For instance, 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 research software development practices and software engineering recommendations (Kelly, 2007; Owojaiye et al., 2021; Storer, 2017). Johanson and Hasselbring (2018) observe that the state-of-the-practice for research software 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 Threats to Validity

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

7.1 Reliability

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

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

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

7.2 Construct Validity

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

- We make indirect measurement of software qualities since meaningful direct measures for qualities like maintainability, reusability and verifiability, are unavailable. We follow the usual assumption that developers achieve higher quality by following procedures and adhering to standards (van Vliet, 2000, p. 112).

- As mentioned in Section 4.1, we could not install or build *dwv*, *GATE*, and *DICOM Viewer*. We used a deployed on-line version for *dwv*, a VM version for *GATE*, but no alternative for *DICOM Viewer*. We might underestimate their rank due to these technical issues.
- Measuring software robustness only involved two pieces of data. This is likely part of the reason for limited variation in the robustness scores (Figure 7). We could add more robustness data by pushing the software to deal with more unexpected situations, like a broken Internet connection, but this would require a larger investment of measurement time.
- We may have inaccurately estimated maintainability by assuming a higher ratio of comments to source code improves maintainability. Moreover, we assumed that maintainability is improved if a high percentage of issues are closed, but a project may have a wealth of open issues, and still be maintainable.
- We assess reusability by the number of code files and LOC per file. This measure is indicative of modularity, but it does not necessarily mean a good modularization. The modules may not be general enough to be easily reused, or the formatting may be poor, or the understandability of the code may be low.
- The understandability measure relies on 10 random source code files, but the 10 files will not necessarily be representative.
- As discussed in Section 4.10, our overall AHP ranking makes the unrealistic assumption of equal weighting.
- We approximated popularity by stars and watches (Section 5), but this assumption may not be valid.
- In building Table 8 some judgement was necessary on our part, since not all guidelines use the same names for artifacts that contain essentially the same information.

7.3 Internal Validity

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

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

7.4 External Validity

If the results of a study can be generalized (applied) to other situations/cases, then the study is externally valid (Runeson and Höst, 2009). We are confident that our search was exhaustive. We do not believe that we missed any highly popular examples. Therefore, the bulk of our validity concerns are internal (Section 7.3). However, our hope is that the trends observed, and the lessons learned for MI software can be applied to other research software. With that in mind we identified the following threat to external validity:

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

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

8 Conclusions

We analyzed the state of the practice for the MI domain with the goal of understanding current practice by answering our four research questions (Section 1.1). 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 measurement results for ranking the 29 projects for nine software qualities. Our ranking results appear credible since they are mostly consistent with the ranking from the scientific community implied by the GitHub stars-per-year metric. As discussed in Section 5, four of the top five software projects appear in both our list and in the GitHub popularity list. Moreover, our top five packages appear among the first eight positions on the GitHub list. The noteworthy discrepancies between the two lists are for the packages that we were unable to install (*dwv* and *Dicom Viewer*).

Based on our grading scores *3D Slicer*, *ImageJ*, *Fiji* and *OHIF Viewer* are the top four software performers. However, the separation between the top performers and the others is not extreme. Almost all packages do well on at least a few qualities, as shown in Table 9, which summarizes the packages ranked first and second for each quality. Almost 70% (20 of 29) of the software packages appear in the top two for at least two qualities. The only packages that do not appear in Table 9, or only appear once, are *Papaya*, *MatrixUser*, *MRIcroGL*, *XMedCon*, *dicompyler*, *DicomBrowser*, *AMIDE*, *3DimViewer*, and *Drishti*. The shortness of this list suggests parity with respect to adoption of best practices for MI software overall.

When it comes to recommended software artifacts, the state of open source MI projects is healthy, with our surveyed examples showing 88% of the documentation artifacts recommended by research software development guidelines (Section 6). However, we did notice areas where practice seems to lag behind the research software development guidelines. For instance, the guidelines recommend three artifacts that were not observed: uninstall instructions, test plans, and requirements

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

Table 9: Top performers for each quality (sorted by order of quality measurement)

documentation. We observed the following recommended artifacts, but only rarely: contributing file, developer code of conduct, code style guidelines, product roadmap, design documentation, and API documentation (Section 6). Future MI projects may wish to consider incorporating more artifacts to potentially improve software quality.

Next steps. Mention threats to validity? Future work on pain points. Point to LBM methodology in that SOP paper?

Acknowledgements

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

Conflict of Interest

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

References

Safia Abdalla. 2016. A template for creating open source contributor guidelines. https://opensource.com/life/16/3/contributor-guidelines-template-and-tips.

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

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

David Carty. 2020. Follow Google's lead with programming style guides. https://www.techtarget.com/searchsoftwarequality/feature/Follow-Googles-lead-with-programming-style-guides.

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

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

Ahmed H. ElSheikh, W. Spencer Smith, and Samir E. Chidiac. 2004. Semi-formal design of reliable mesh generation systems. *Advances in Engineering Software* 35, 12 (2004), 827–841.

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

ESA. February 1991. *ESA Software Engineering Standards, PSS-05-0 Issue 2.* Technical Report. European Space Agency.

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).

E. Gamma, R. Helm, J. Vlissides, and I R Johnson. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional.

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].

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.

Daniel M. Hoffman and Paul A. Strooper. 1995. *Software Design, Automated Testing, and Maintenance: A Practical Approach.* International Thomson Computer Press, New York, NY, USA.

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.

IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998* (Oct. 1998), 1–40. https://doi.org/10.1109/IEEESTD.1998.88286

Joint Task Force on Software Engineering Ethics IEEE-CS/ACM and Professional Practices. 1999. Code of Ethics, IEEE Computer Society. https://www.computer.org/education/code-of-ethics.

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.

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

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

Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2018. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication* 48, 3 (2018), 295–330. https://doi.org/10.1177/0047281617721853 arXiv:https://doi.org/10.1177/0047281617721853

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].

Jürgen Münch, Stefan Trieflinger, and Dominic Lang. 2019. Product Roadmap – From Vision to Reality: A Systematic Literature Review. In *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. 1–8. https://doi.org/10.1109/ICE.2019.8792654

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

NASA. 1989. *Software requirements DID, SMAP-DID-P200-SW, Release 4.3.* Technical Report. National Aeronautics and Space Agency.

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 Open-Source 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).

D.L. Parnas, P.C. Clement, and D. M. Weiss. 1984. The modular structure of complex systems. In *International Conference on Software Engineering*. 408–419.

R. Phaal, C.J.P. Farrukh, and D.R. Probert. 2005. Developing a technology roadmapping system. In *A Unifying Discipline for Melting the Boundaries Technology Management:*. 99–111. https://doi.org/10.1109/PICMET.2005.1509680

Roman Pichler. 2012. Working with an Agile Product Roadmap. https://www.romanpichler.com/blog/agile-product-roadmap/.

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]

Professional Engineers Act. 2021. Professional Engineers Act, RSO 1990, c P. 28. https://canlii.ca/t/5568z.

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

Suzanne Robertson and James Robertson. 1999. *Mastering the Requirements Process*. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, Chapter Volere Requirements Specification Template, 353–391.

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

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

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

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.

Vandana Singh, Brice Bongiovanni, and William Brandon. 2021. Codes of conduct in Open Source Software—for warm and fuzzy feelings or equality in community? *Software Quality Journal* (2021). https://doi.org/10.1007/s11219-020-09543-w

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

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

Spencer Smith, Zheng Zeng, and Jacques Carette. 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, Lei Lai, and Ridha Khedri. 2007. Requirements Analysis for Engineering Computation: A Systematic Approach for Improving Software Reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation* 13, 1 (Feb. 2007), 83–107. https://doi.org/10.1007/s11155-006-9020-7

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

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

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

W. Spencer Smith and Wen Yu. 2009. A Document Driven Methodology for Improving the Quality of a Parallel Mesh Generation Toolbox. *Advances in Engineering Software* 40, 11 (Nov. 2009), 1155–1167. https://doi.org/10.1016/j.advengsoft.2009.05.003

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/.

Parastou Tourani, Bram Adams, and Alexander Serebrenik. 2017. Code of conduct in open source projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 24–33. https://doi.org/10.1109/SANER.2017.7884606

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

Jarno Vähäniitty, Casper Lassenius, and Kristian Rautiainen. 2002. An Approach to Product Roadmapping in Small Software Product Businesses. University of Technologie, Software Business and Engineering Institute, Helsinki, Finland.

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

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

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

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. 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. 2021b. 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

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.