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

Abstract

We review the state of the practice for the development of Medical Imaging (MI) software based on data available in open-source repositories. We selected 29 projects from 48 candidates and assessed 9 software qualities by answering 108 questions for each. Using the Analytic Hierarchy Process (AHP) on the quantitative data, we ranked the MI software. The top five are *3D Slicer*, *ImageJ*, *Fiji*, *OHIF Viewer*, and *ParaView*. This is consistent with the community's view, with four of these also appearing in the top five using GitHub metrics (stars-per-year). The quality and quantity of documentation present in a project correlate quite well with its popularity. Generally, MI software is in a healthy state: in the repositories, we observed 88% of the documentation artifacts recommended by research software development guidelines and 100% of MI projects use version control tools. However, the current state of the practice deviates from existing guidelines as some recommended artifacts are rarely present (like a test plan, requirements specification, and code style guidelines), low usage of continuous integration (17% of the projects), low use of unit testing (about 50% of projects), and room for improvement with documentation. From developer interviews, we identified 7 concerns: lack of development time, lack of funding, technology hurdles, correctness, usability, maintainability, and reproducibility. We recommend: increasing effort on documentation, increasing testing by enriching datasets, increasing continuous integration, moving to web applications, employing linters, using peer reviews, and designing for change.

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

1. Introduction

We 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 critical information for diagnostic, analytic, and medical applications. Given its importance, we want to understand the merits and drawbacks of the current development processes, tools, and methodologies. We use a software engineering lens to assess the quality of existing MI software.

1.1. Research Questions

As well as state of the practice for MI software, we would like to understand the impact of the often cited gap between recommended software engineering practices and the practices used for most research software [1]. Although scientists spend a substantial proportion of their working times on software development [2, 3], few are formally trained [2].

Our investigation is based on ten research questions:

RQ1: What MI open source software projects exist? (Section 2)

RQ2: Based on quantitative measurements of each project's development practices, which projects follow best practices? (Section 3)

RQ3: How similar are the top projects identified in RQ2 to the most popular projects as viewed by the community? (Section 4)

RQ4: How do the artifacts (documents, scripts and code) present in MI repositories compare to the artifacts used for research software in general? (Section 5)

RQ5: How does the use of development tools compare between MI software and research software in general (Section 6) (RQ5.a); and, project management (RQ5.b)?

RQ6: How does the use of principles, processes, and methodologies compare between MI software and research software in general? (Section 7)

RQ7: For MI software developers, what pain points do they experience? (Section 8)

RQ8: How do the pain points compare between MI software and research software in general? (Section 8)

RQ9: What best practices are taken by MI developers to address their pain points (Section 8) and quality concerns? (Section 9)

RQ10: What processes, techniques and tools can potentially address the identified pain points from RQ7? (Section 10)

1.2. Scope

We only cover MI visualization software. We exclude other categories of MI software, including segmentation, registration, visualization, enhancement, quantification, simulation, plus MI archiving and telemedicine systems (compression, storage, and communication). We also exclude statistical analysis and image-based physiological modelling and feature extraction, classification, and interpretation. Software that provides MI support functions is also out of scope; therefore, we have not assessed the toolkit libraries VTK and ITK. Finally, Picture Archiving and Communication System (PACS), which helps users to economically store and conveniently access images, are considered out of scope.

1.3. Methodology

We have a standard set of questions designed to assess the qualities of any research software project. [blind review redacted details on the author's previous studies using the same methodology.] We maintain the previous constraint that the work load for measuring a given domain should take around one person-month's worth of effort (20 working days at 8 person-hours per day).

We identify a list of potential packages (through online searches) which is then filtered and vetted by a domain expert. We aim for roughly 30 packages. For each remaining package, we measure its qualities by filling in a grading template [citation redacted for double blind]. This data is used to rank the projects with the Analytic Hierarchy Process (AHP). We summarize further details on the interaction with the domain expert, software qualities, grading the software and AHP below and in longer form in [redacted for double blind].

1.3.1. Domain Expert

The Domain Expert vets the proposed list because online resources can be inaccurate. The expert also vets the AHP ranking. For the current assessment, our Domain Expert is [details of our domain expert removed for double-blind].

In advance of the first meeting with the Domain Expert, they are asked to independently create a list of top software packages in the domain. This helps get the expert's knowledge refreshed in advance of the meeting.

1.3.2. Software Qualities

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." For this study we selected 9 qualities to measure: installability, correctness/verifiability, reliability, robustness, usability, maintainability, reusability, understandability, and visibility/transparency. With the exception of installability, all the qualities are defined in Ghezzi et al. (2003) [4]. Installability is defined as the effort required for the installation and/or uninstallation of software in a specified environment [5].

1.3.3. Grading

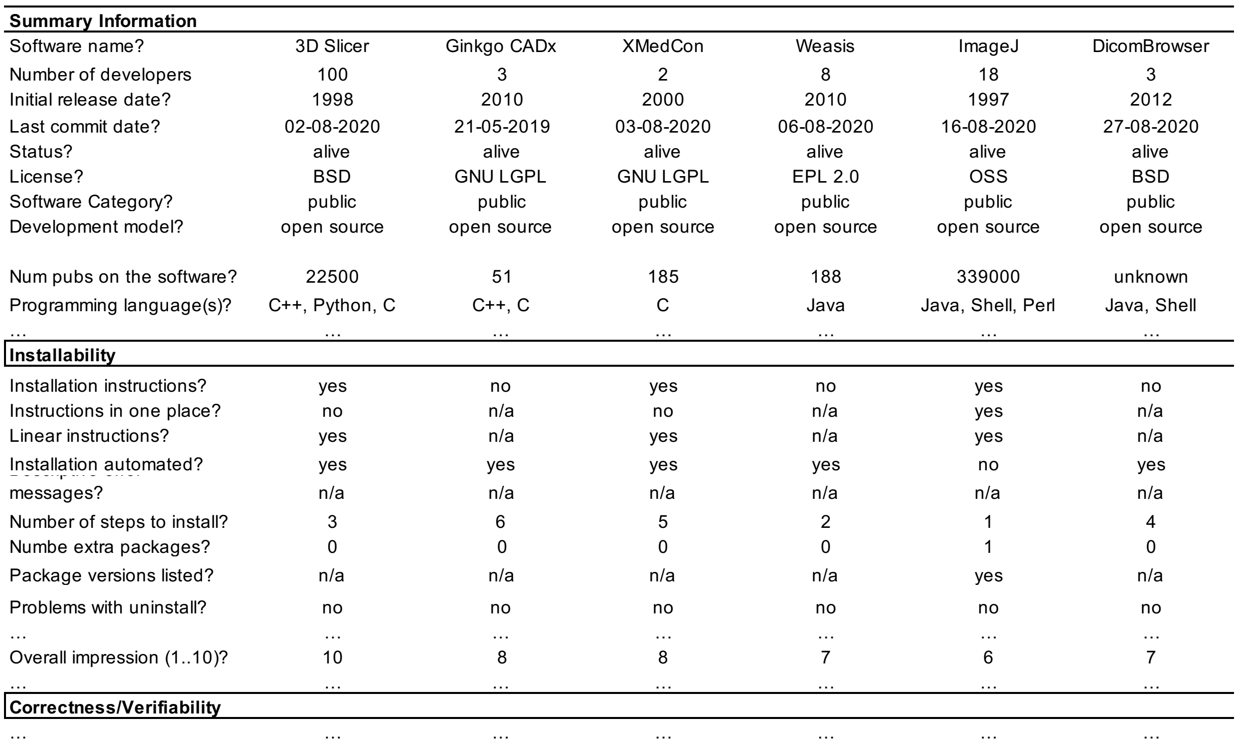
We use an existing template [citation redacted] that is designed to measure the aforementioned qualities. To stay within our given measurement time frame, each package gets up to five hours of time. Project developers can be contacted for help regarding installation, if necessary, but we impose a cap of about two hours on the installation process. Figure 1 shows an excerpt of the measurement spreadsheet. The rows are the measures and the columns correspond to the software packages. [The full data is available on Mendeley; link will be provided after refereeing.]

The full template consists of 108 questions over 9 qualities. These questions are designed to be unambiguous, quantifiable, and measurable with constrained time and domain knowledge.

The grader, after answering questions for each quality assigns an overall score (between 1 and 10) based on the answers. Several of the qualities use the word "surface" to highlight that these particular qualities are a shallow measure. For example, usability is not measured using user studies. Instead, we look for signs that the developers considered usability. We use two freeware tools to collect repository related data: GitStats and Sloc Cloc and Code (scc). Further details on quality measurement are provided in [redacted for double blind].

1.3.4. Analytic Hierarchy Process (AHP)

Developed by Saaty in the 1970s, AHP is widely used to analyze multiple criteria decisions [6]. AHP organizes multiple criteria in a hierarchical structure and uses pairwise comparisons between



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 1: Grading template example

alternatives to calculate relative ratios [7]. 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). With AHP the sum of the grades (scores) for all products for a given quality will be 1.0. 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.

1.3.5. Interview Methods

The repository-based measurements are incomplete because they don't generally capture the development process, the developer pain points, the perceived threats to software quality, and the developers' strategies to address these threats. Therefore, part of our methodology involves interviewing developers. We based our interviews on a list of 20 questions, which can be found in [citation redacted for double blind]. Some questions are about the background of the software, the development teams, the interviewees, and how they organize their projects. We also ask about the developer's understanding of the users. Additional questions focus on the current and past difficulties, and the solutions the team has found, or plan to try. We also discuss documentation, both with respect to how it is currently done, and how it is perceived. A few questions are about specific software qualities, such as maintainability, understandability, usability, and reproducibility. The interviews are semi-structured based on the question list.

2. In-Scope Open-Source MI Software

We initially identified 48 candidate software projects from the literature [8, 9, 10], on-line articles [11, 12, 13], and forum discussions [14]. Then we filtered as follows:

1. Removed the packages with no source code available, such as *MicroDicom*, *Aliza*, and *jivex*.

2. Focused on MI software that provides visualization functions. We removed seven packages that were toolkits or libraries, such as *VTK*, *ITK*, and *dcm4che*, and another three that were for PACS.

3. Removed *Open Dicom Viewer* as it has not received any updates since 2011.

The Domain Expert provided a list 12 software packages. We found 6 packages were on both lists: *3D Slicer*, *Horos*, *ImageJ*, *Fiji*, *MRIcron* (we use its descendant *MRIcroGL*) and *Mango* (we use the web version *Papaya*). The remaining six packages were on our out-of-scope list. The Domain Expert agreed with our final choice of 29 packages. Table 1 summarizes the in-scope open-source MI software that is available at the time of measurement (the year 2020), thus answering RQ1.

In Table 1 the projects are sorted in descending order of lines of code. 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 on [removed for blind review]

The programming languages used in order of decreasing popularity are C++, JavaScript, Java, C, Python, Pascal, Matlab. The most popular language is C++, for 11 of 29 projects; Pascal and Matlab were each used for a single project.

3. Which Projects Follow Best Practices?

We measured the software as described in Section 1.3. In the absence of a specific real world context, we assumed all nine qualities are equally important. Figure 2 shows the overall AHP scores in descending order.

The top four software products *3D Slicer*, *ImageJ*, *Fiji*, and *OHIF Viewer* have higher 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 qualities.

Software	Rlsd	Updated	Fnd	NOC	LOC	OS			Web
						W	M	L	
ParaView [15]	2002	2020-10	✓	100	886326	✓	✓	✓	✓
Gwyddion [16]	2004	2020-11		38	643427	✓	✓	✓	
Horos [17]	?	2020-04		21	561617		✓		
OsiriX Lite [18]	2004	2019-11		9	544304		✓		
3D Slicer [19]	1998	2020-08	✓	100	501451	✓	✓	✓	
Drishti [20]	2012	2020-08		1	268168	✓	✓	✓	
Ginkgo CADx [21]	2010	2019-05		3	257144	✓	✓	✓	
GATE [22]	2011	2020-10		45	207122		✓	✓	
3DimViewer [23]	?	2020-03	✓	3	178065	✓	✓		
medInria [24]	2009	2020-11		21	148924	✓	✓	✓	
BioImage Suite Web [25]	2018	2020-10	✓	13	139699	✓	✓	✓	✓
Weasis [26]	2010	2020-08		8	123272	✓	✓	✓	
AMIDE [27]	2006	2017-01		4	102827	✓	✓	✓	
XMedCon [28]	2000	2020-08		2	96767	✓	✓	✓	
ITK-SNAP [29]	2006	2020-06	✓	13	88530	✓	✓	✓	
Papaya [30]	2012	2019-05		9	71831	✓	✓	✓	
OHIF Viewer [31]	2015	2020-10		76	63951	✓	✓	✓	✓
SMILI [32]	2014	2020-06		9	62626	✓	✓	✓	
INVESALIUS 3 [33]	2009	2020-09		10	48605	✓	✓	✓	
dwv [34]	2012	2020-09		22	47815	✓	✓	✓	✓
DICOM Viewer [35]	2018	2020-04	✓	5	30761	✓	✓	✓	
MicroView [36]	2015	2020-08		2	27470	✓	✓	✓	
MatrixUser [37]	2013	2018-07		1	23121	✓	✓	✓	
Slice:Drop [38]	2012	2020-04		3	19020	✓	✓	✓	✓
dicompyler [39]	2009	2020-01		2	15941	✓	✓		
Fiji [40]	2011	2020-08	✓	55	10833	✓	✓	✓	
ImageJ [41]	1997	2020-08	✓	18	9681	✓	✓	✓	
MRIcroGL [42]	2015	2020-08		2	8493	✓	✓	✓	
DicomBrowser [43]	2012	2020-08		3	5505	✓	✓	✓	

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

The overall score is based on the measurement of the identified qualities. Full details of how the projects compare for each quality can be found in [citation redacted for blind review]. The highlights are as follows:

Installability We found installation instructions for 16 projects, but two did not need them (*BioImage Suite Web* and *Slice:Drop*) as they are web applications. 10 of the projects required extra dependencies: Five depend on a specific browser; *dwv*, *OHIF Viewer*, and *GATE* needs extra

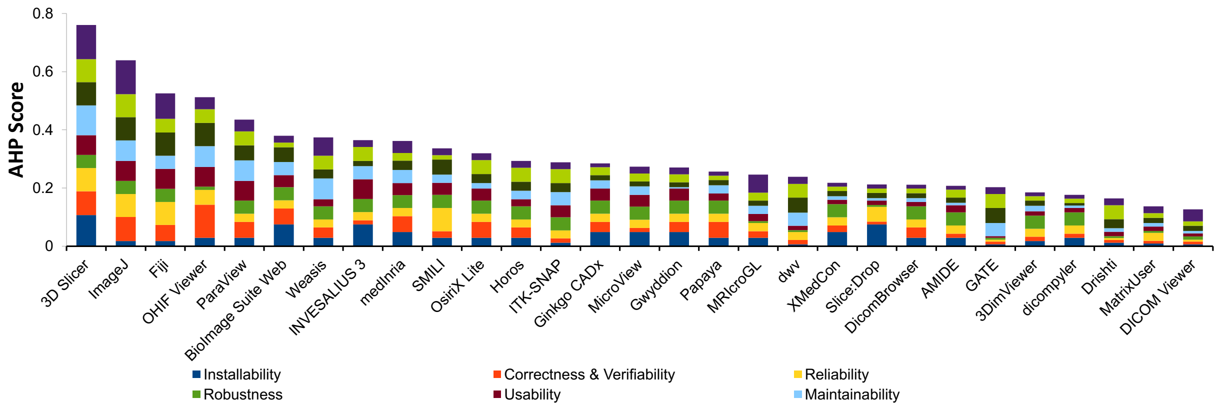


Figure 2: Overall AHP scores with an equal weighting for all 9 software qualities

libraries to build; *ImageJ* and *Fiji* need an unzip tool; *MatrixUser* needs Matlab; *DICOM Viewer* needs a Nextcloud platform. We were not able to install *GATE*, *dwv*, *DICOM Viewer* even after trying for 2 hours (each).

Correctness & Verifiability The packages with higher scores for correctness and verifiability used a wider array of techniques to improve correctness, and had better documentation to support this. For instance, we looked for evidence of unit testing, and found evidence for only about half of the projects. We identified five projects using continuous integration tools: *3D Slicer*, *ImageJ*, *Fiji*, *dwv*, and *OHIF Viewer*. The only requirements-related document we found was a road map of *3D Slicer*, which contained design requirements for upcoming changes.

Surface Reliability We were able to follow the steps in the tutorials that existed (seven packages had them.) However, *GATE* could not open macro files and became unresponsive several times, without any descriptive error message. We found that *Drishti* crashed when loading damaged image files, without showing any descriptive error message.

Surface Robustness 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 relabelling 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, 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, but the software then crashed.

Surface Usability The software with higher usability 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.

Maintainability We gave *3D Slicer* the highest score for maintainability because we found it had the most comprehensive artifacts. Only a few of the 29 projects had a product, developer's manual, or API (Application Programming Interface) 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%). Twenty-seven of the 29 projects used git for version control, with 24 of these using GitHub. *AMIDE* used Mercurial and *Gwyddion* used Subversion. *XMedCon*, *AMIDE*, and *Gwyddion* used SourceForge. *DicomBrowser* and *3DimViewer* used BitBucket.

Reusability We have assumed that smaller code files are likely more reusable – see Table 2 for the details.

Surface Understandability All projects had a consistent coding style with parameters in the same order for all functions, modularized code, and, clear comments that indicate 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*.

Visibility/Transparency Generally speaking, the teams that actively documented their development process and plans scored higher. *3D Slicer* and *ImageJ* were the only projects to include documentation for all of the following: development process, project status, development environment and release notes.

4. Comparison to Community Ranking

We use GitHub stars, number of forks and number of people watching the projects as proxies for community ranking. Table 3 show the statistics for data collected in July 2021.

Our ranking and GitHub popularity, at least for the top five projects, seems to line up well. However, we ranked some popular packages fairly low, such as *dwv*. This is because we were unable to build it locally, even though we followed its installation instructions. However, we were able to use its web version for the rest of the measurements. Additionally, this version did not detect a broken DICOM file and instead displayed a blank image (Section 3). *DICOM Viewer* ranked low as we were unable to install the NextCloud platform.

We weighted all qualities equally, which is not the likely the same weighting that users implicitly use. To properly assess this would require a broad user study. Furthermore our measures of popularity (like stars) are only *proxies* which are biased towards past rather than current preferences [44], as these are monotonically increasing quantities. Finally there are often more factors than just quality that influence the popularity of products.

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 2: Number of files and lines (by reusability scores)

Although both rankings are imperfect measures, they nevertheless suggest a correlation between best practices and popularity. We don't know if this is causal, in either direction (i.e. if best practices enable popularity or if popularity increases the need for best practices).

5. Comparing with Recommended Software Artifacts

We use a set of nine research software development guidelines to compare recommended software artifacts versus those present in MI software:

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 3: 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)

- United States Geological Survey Software Planning Checklist [45],
- DLR (German Aerospace Centre) Software Engineering Guidelines [46],
- Scottish Covid-19 Response Consortium Software Checklist [47],
- Good Enough Practices in Scientific Computing [48],
- xSDK (Extreme-scale Scientific Software Development Kit) Community Package Policies [49],
- Trilinos Developers Guide [50],
- EURISE (European Research Infrastructure Software Engineers') Network Technical Reference [51],

- CLARIAH (Common Lab Research Infrastructure for the Arts and Humanities) Guidelines for Software Quality [52], and
- A Set of Common Software Quality Assurance Baseline Criteria for Research Projects [53].

	[45]	[46]	[47]	[48]	[49]	[50]	[51]	[52]	[53]	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 4: Comparison of Recommended Artifacts in Software Development Guidelines to Artifacts in MI Projects (C for Common, U for Uncommon and R for Rare)

We show the recommended artifacts in Table 4, one per row. 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) (20 to 29 (>67%) packages), uncommonly (U) (10 to 19 (33-67%)

packages) or rarely (R) (1 to 9 (<33%) packages). 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. The content of README files shows significant variation between projects [54]. 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 [54].

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 (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 5: Artifacts Present in MI Packages, Classified by Frequency (number of occurrences)

Popularity in Table 4 does not imply that these oft recommended artifacts are the most important. Guidelines are often brief, to encourage adoption, and thus even guidelines that mention the need for installation instructions rarely mention uninstallation instructions. Two items in Table 5, which presents our measurements for the MI software, do not appear in any guidelines: *Troubleshooting guide* and *Developer's manual*. However the information within these documents overlaps with the recommended artifacts. Troubleshooting information often can be found in a User Manual, while the information in a "Developer's Manual" is often scattered amongst many other documents.

Three of the 26 recommended artifacts were never observed in the MI software: i) Uninstall, ii) Test plans, and iii) Requirements. It is possible that some of these were created but never put under version control.

Neglecting requirements documentation is unfortunately common for research software, and MI software is no exception to this trend. Although such documentation is recommended by some [46, 50, 55], in practice this is rare [56]. Sanders and Kelly [57] 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. Requirements are the least commonly produced type of documentation for research software in general [58].

This is unfortunate as when scientific developers are surveyed on their pain points, Wiese et al. [59] found that software requirements and management is the software engineering discipline that most hurts them, accounting for 23% of the technical problems reported by study participants. Further adding to the misfortune, there is a widespread perception that up-front requirements are impossible for research software [60, 61]. Fortunately an iterative approach to requirements is feasible [62], and research-software specific templates exist [63].

A theme emerges amongst the artifacts rarely observed in practice: they are developer-focused (a list of library dependencies, a contributor's guide, a developer Code of Conduct, coding style guidelines, product roadmap, design documentation and API documentation).

Other communities use checklists to help with best practices. Examples include checklists for merging branches [64], for saving and sharing changes to the project [48], for new and departing team members [65], for processes related to commits and releases [50] and for overall software quality [51, 66].

MI projects fall somewhat short of recommended best practices, but are not alone amongst research software projects. This gap has been documented before [1, 67], and is known to cause sustainability and reliability problems [68], and to waste development effort [69].

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

We use data from interviews, as described in Section 1.3.5 to answer RQ5 (tool usage), RQ6 (methodology) and RQ7 (pain points). Requests were sent to all 29 projects. Nine developers from eight of the projects agreed to participate: *3D Slicer*, *INVESALIUS 3*, *dwv*, *BioImage Suite Web*, *ITK-SNAP*, *MRIcroGL*, *Weasis*, and *OHIF*. We spent about 90 minutes for each interview. The full interview answers can be found in [citation redacted for blind review].

Developers use software tools for user support, version control, Continuous Integration and Deployment (CI/CD), documentation and project management. We summarize the tool usage in each of these categories, and compare this to the usage by the research software community.

User support. Table 6 summarizes the user support models by the number of projects using each model (projects may use more than one support model).

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

Table 6: User support models by number of projects

Version control. Most teams interviewed (eight of nine) use GitHub for version control, which makes it unsurprising that they also use pull requests for managing community contributions. The experience of using GitHub was reported as generally positive; some teams had actively migrated to GitHub from other version control systems. This represents a considerable uptake since its rare use in 2006 [70], which matches the uptake in other communities: 10 years ago Nguyen et al [58] estimated that only 50% of research software projects used version control, and by 2018 this had jumped to over 80% [71] with many communities [72] reaching 100% already in 2018. In 2022, version control use sits at over 95% [73].

CI/CD. Only 17% of MI projects use CI/CD tools. We estimated this using artifacts present in the repository – actual usage may be higher as some tools are entirely external. This was the case for a study of Lattice Boltzmann Method (LBM) software [citation redacted for blind review]. This contrasts with the high frequency of guidelines recommending continuous integration [47, 64, 51, 74, 52]. A recent survey [73] of practitioners (rather than projects) suggests higher use of CI/CD with 54% of respondents indicating that they use it.

Documentation. The most popular (mentioned by about 30% of developers) were forum discussions and videos. The second most popular options (20%) were GitHub, wiki pages, workshops, and social media. The least frequently mentioned options (about 10% of developers) included writing books, and Google forms. As a point of contrast, LBM software most often uses API document generation tools, like doxygen and sphinx [citation redacted for blind review].

Project management. Two types of tools were mentioned: i) trackers, including GitHub, issue trackers, bug trackers and Jira; and, ii) documentation tools, including GitHub, Wiki page, Google Doc, and Confluence. Of the named tools, interviewees mentioned GitHub 3 times, and each of the other tools once.

Tool use for MI software is similar to tool use for ocean modelling software [75]. Both use tools for editing, compiling, code management, testing, building, and project management, but differ in the latter's use of Kanban boards for project management.

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

In our interviews, developers' responses to questions about development model were vague, with only two interviewees following a definite model, with others feeling their process was "similar" to an existing model. Three teams followed agile or agile-like, two teams followed waterfall or waterfall-like, and three teams explicitly stated that their process was undefined or self-directed.

This matches observations for research software in general. Scientific developers naturally use an agile philosophy [76, 60, 77, 78, 56], or an amethododical process [79], or a knowledge acquisition driven process [80]. A waterfall-like process can work for research software [62], especially if the developers work iteratively and incrementally, but externally document their work as if they followed a rational design process [81].

No interviewee mentioned a strictly defined project management process. The most common approach was issue-driven via bug reports and feature requests. [Redacted for blind review] provides details on the specific approaches used by the interviewees.

Less than half of the projects used unit testing, even though the interviewees believed that testing (including usability tests) was the top approach to improve correctness, usability, and reproducibility. This level of testing matches what was observed for LBM software [citation redacted for blind review] and is apparently greater than the level of testing for ocean modelling software [75].

All interviewees thought that documentation was essential to their projects, and most of them (7 of 8) said that it could save time spent answering questions from users and developers. Half of them (4 of 8) saw the need to improve their documentation, and only three of them thought their documentations conveyed information clearly enough.

8. Developer Pain Points

Interviews identified the following issues: 1. lack of time, 2. lack of funding, 3. technology hurdles, 4. correctness, and 5. usability. We first detail each pain point, and contrast this experience with that from other domains. We then separately cover potential way to mitigate these issues.

Pinto et al. [82] list some pain points that did not come up in our conversations: interruptions while coding, scope bloat, lack of user feedback, hard to collaborate on software projects, and aloneness. Wiese et al. [59] mentions two more: reproducibility, and software scope determination. [citation redacted for blind review] adds lack of software experience, technical debt, and documentation. We cannot conclude that these are not also issues with MI software due to our small response size.

P1: **Lack of Development Time:** This was felt to be the most significant obstacle, and is not uncommon [82, 83, 59].

P2: **Lack of Funding:** As research software is not yet seen as essential infrastructure, there is an ongoing struggle to attract funding to build and, even harder, maintain research software. This is a common complaint [84, 85, 86]. Software output is not always counted when judging the academic excellence of academics, which forces researchers who write software to spend extra time on publicity [59]. In some domains, like biology [87], software is infrequently cited even when used pervasively. In other words, there is a lack of a formal reward system for research software [82].

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

P4: **Correctness:** The most mentioned thread to correctness was complexity. Example sources of complexity include: a large variety of data formats, complicated data standards, differing outputs between medical imaging machines, and the addition of (non-viewing related) functionality. Other identified threats to correctness:

- Lack of real world image data for testing, in part because of patient privacy concerns;

- Tests are expensive and time-consuming because of the need for huge datasets;
- Software releases are difficult to manage;
- No systematic unit testing; and,
- No dedicated quality assurance team.

P5: **Usability:** The discussion with the developers focused on usability issues for two classes of users: the end users and other developers. The threats to usability for end users include an unintuitive user interface, inadequate feedback from the interface (such as lack of a progress bar), users being unable to determine the purpose of the software, not all users knowing if the software includes certain features, not all users understanding how to use the command line tool, and not all users understanding that the software is a web application. For developers the threats to usability include not being able to find clear instructions on how to deploy the software, and the architecture being difficult for new developers to understand.

At least to some extent the problems for MI software users are due to holes in their knowledge of software and computing technology, and sometimes also lack of expertize in their domain to be able to adequately use the software [59].

For each of the above, the MI developers suggested various potential solutions, some of which have proven their worth in other domains.

P1: **Lack of Development Time:**

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

P2: **Lack of Funding:** One interviewee proposed an interesting idea: Licensing the software to commercial companies to integrate it into their products. In general, solutions need to be more systemic, thus we are seeing the rise of "research software engineers" in various jurisdictions.

P3: **Technology Hurdles:**

- Adopting a web-based approach with backend servers, to better support lower-end computers;
- Using memory-mapped files to consume less computer memory, to better support lower-end computers;
- Using computing power from the computers GPU for web applications;
- Maintaining better documentations to ease the development and maintenance processes;

- Improving performance via more powerful computers, which one interviewee pointed out has already happened.

A number of developers perceive that some of the “shifting technologies” problems could be alleviated by moving to web-based applications. Most of the teams (83%) chose to develop native applications. Three of the eight we interviewed were building web applications, and another team was considering it.

P4: **Correctness:** Testing was the most often mentioned strategy for ensuring correctness. Teams mentioned several test-related activities, including test-driven development, component tests, integration tests, smoke tests, regression tests, self tests and automated tests. This puts MI software ahead of other domains where insufficient testing is a problem [82] and insufficiently well-understood [88].

Another correctness strategy mentioned multiple times is a development process that involves stable releases and nightly builds. Other strategies that were mentioned include: (a) using CI/CD, (b) using de-identified copies of medical images for debugging, (c) sending beta versions to medical workers who can access the data to do the tests, (d) collecting/maintaining a dataset of problematic images, (e) using open datasets, (f) if (part of) the team belongs to a medical school or a hospital, using the datasets they can access, (g) if the team has access to MRI scanners, self-building sample images for testing, and (h) if the team has connections with MI equipment manufacturers, asking for their help on data format problems.

P5: **Usability:**

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

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

9. Improving Software Qualities

The structured interviews also included an active discussion of software qualities, in particular maintainability and reproducibility. This discussion did not come up as “pain points”, but rather through planned questions we asked for improving these qualities.

Q1: **Maintainability:** Maintainability has been rated as the third most important software quality for research software [58], and developers complain about accumulating too much technical debt [89].

The main strategy mentioned to reduce code duplication is using properly structured libraries. Other strategies mentioned include supporting third-party extensions, an easy-to-understand architecture, a dedicated architect, starting from simple solutions, and documentation.

Q2: **Reproducibility:** An important precondition for reproducibility is increased and better documentation (which also helps with many of the aforementioned pain points). The challenges of inadequate documentation are a known problem for research software [82, 59] and for non-research software [90] alike.

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

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

10. Recommendations

We provide specific recommendations for future consideration – these are not criticisms of past and/or current development practices. We expand on some of the ideas from our interviews, and bring in further ideas that could have a noticeable impact. The "new" ideas are not novel to this paper but rather from the software engineering and software carpentry domains, but these ideas have not yet seen wide adoption for MI software. We list the ideas roughly in the order of increasing implementation effort.

10.1. Use Continuous Integration

Continuous integration involves building and testing the software every time a coherent change is made to the code (i.e. via a pull request) [91, p. 13], [92, 93]. Initial setup can be cumbersome, but the benefits are impressive:

- Elimination of headaches associated with a separate integration phase [93], [91, p. 20].
- Detection and removal of bugs [93] via automated testing.

- Everyone is always working on a stable base that always passes all tests.
- If the CI system uses generators and linters, it will also have current documentation and clean code.

CI helps relieve several point points: 1. by eliminating the time-consuming integration phase, freeing more time for development (P1), 2. Automated tests help with ensuring correctness (P4) and reproducibility (Q2).

10.2. Perhaps Move To Web Applications

Multiple pain points, especially technology hurdles (P3), could be alleviated by moving to a web application. However, this must be an active decision as it may not be a good fit in all cases. The decision needs to be based on whether, on balance, this would improve the four factors identified earlier for the technology hurdle: compatibility, maintainability, performance, and security. The following considerations may help a team make a decision about the move to web applications:

- **Modern technologies may improve frontend performance.** Complex, graphics-heavy user-interfaces often do not respond as quickly as native applications. However, new technologies may help. For example, some JavaScript libraries can help the frontend harness the power of the computer's GPU and accelerate graphical computing. In addition, there are new frameworks helping developers with cross-platform compatibility. For example, the Flutter project supports web, mobile, and desktop OS with one codebase. Other options include Vue, Angular, React, and Elm.
- **Backend servers can potentially deliver high performance.** If the principal bottleneck is computational, backend servers may outperform native applications, as long as latency is not an issue. Options include Django, Laravel and Node.js. When traffic and latency are an issue, Gin is an option.
- **Some backend servers can be inexpensive.** Serverless solutions from cloud service providers (like Amazon Web Services (AWS) and Google Cloud Platform) may be worth exploring. These still use a server, but only actual use is charged.
- **Web transmission may diminish security.** Transferring sensitive data on-line can be a problem for projects requiring high security, even if using encryption. Some jurisdiction may not allow transmission of some health data at all.

10.3. Enrich the Testing Datasets

Access to real-world imaging datasets can be a testing bottleneck, jeopardising correctness. Building on developers' suggestions:

- **Build and maintain good connections to datasets.** Teams can build connections with medical professionals, who may have access to private datasets and can perform tests for the team.

- **Collect and maintain datasets over time.** It is especially important to have access to "unique" inputs that were outliers in historical testing scenarios.

- **Search for open data sources.** There are many open MI datasets: Chest X-ray Datasets by National Institute of Health [94], Cancer Imaging Archive [95], MedPix by National Library of Medicine [96], and datasets for liver [97] and brain [98] tumor segmentation benchmarks.

- **Create sample data for testing.** If a team is able to create baseline and/or synthetic data sets with known characteristics, this is a worthwhile investment (and should hopefully be made public). For example, using an MRI scanner to create images of objects, animals, and volunteers who waive the privacy rights to their image (i.e. not patients).

10.4. Use Linters and other Static Analysis Tools

Linters are simple tools that analyze code to find programming errors, suspicious constructs, and stylistic inconsistencies [99]. While they can be used in an ad hoc fashion, coupling them with CI is where the largest gain happens. Automatic use of code quality tools is increasing commonplace for industrial software – it is regrettable that the research software guidelines have no caught up to this.

These tools have many benefits: finding potential bugs, finding memory leaks, improving performance, standardizing code with respect to formatting, removing silly errors before code reviews, and catching potential security issues [100]. Most popular programming languages possess such tools. This helps with P1, Q1, and correctness (P4).

The pervasive use of such tools to enforce documentation standards partially explains the relatively higher quality of statistical tools that are part of the Comprehensive R Archive Network (CRAN) [citation redacted for blind review].

10.5. Conduct Peer Reviews

Peer review is recommended by many [50, 53, 45]. Modern peer review processes are lightweight, informal, tool-based, asynchronous, and focused on reviewing code changes [101]. GitHub, for example, provides such tools for its pull requests.

Rigorous inspection finds 60–65% of latent defects on average, and often tops 85% in defect removal efficiency [102]. The success rate of code inspection is generally higher than most forms of testing, which average between 30–35% for defect removal efficiency [103, 102]. For research software, Kelly and Shepard [104] show a task based inspection approach can be effective.

Due to improving code quality and increasing knowledge transfer, peer review addresses the same pain points and qualities as linters (P1, Q1 and P4). The benefits can be increased by extending reviews to all artifacts, including documentation, build scripts, test cases and the development process itself.

10.6. Design For Change

Although the advice to modularize research software to handle complexity is common [105, 106, 1], specific guidelines on how to divide the software into modules is less prevalent. Not every decomposition is a good design for supporting change, as shown by Parnas [107]. For instance, a design with low cohesion and high coupling [4, p. 48] will make change difficult. Especially in research software, where change is inevitable, designers need to produce a modularization that supports change. Ocean modelling software is currently feeling the pain of not emphasizing modularization in legacy code [75].

11. Threats to Validity

We follow Ampatzoglou et al.'s [108] analysis of threats to validity in software engineering secondary studies.

11.1. Reliability

A study is reliable if repeating it by another researcher, using the same methodology, would lead to the same results [109]. We identify the following threats: 1. One individual does the measures for all packages. A different evaluator might find different results, due to differences in abilities, experience, and biases. 2. The measurements for the full set of packages took several months (of elapsed time). Over this time the software repositories may have changed and the reviewer's judgement may have drifted.

The measurement process used has previously been shown to be reasonably reproducible. [citation redacted] reports grading 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.

11.2. Construct Validity

Construct validity is when the adopted metrics represent what they are intended to measure [109]. We have identified the following potential issues:

- 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 [110, p. 112].

- We could not do all measurements for some software as we could not install or build *dwv*, *GATE*, and *DICOM Viewer*. We used a deployed on-line version for *dwv*, a virtual machine version for *GATE*, but no alternative for *DICOM Viewer*.

- Robustness measurements involve only two pieces of data, leading to limited variation in the robustness scores. Our measurement-time budget limited what we could achieve here.

- Our maintainability proxies (higher ratio of comments to source, high percentage of closed issues) have not been validated.

- While smaller modules tend to be easier to reuse, small modules are not necessarily good modules, nor understandable modules.

- The understandability measure relies on 10 random source code files, but the 10 files will not necessarily be representative.
- Our overall AHP ranking makes the unrealistic assumption of equal weighting.
- We approximated popularity by stars and watches.
- Table 4 required judgement as not all guidelines use the same names for artifacts that contain essentially the same information.

11.3. Internal Validity

Internal validity means that discovered causal relations are trustworthy and cannot be explained by other factors [109]. We identify the following:

- Our search (Section 2) could have missed a relevant package.
- Our methodology assumes that development activities will leave a trace in the repositories, but this is not necessarily true. For instance, we saw little evidence of requirements (Section 5), but teams might keep this kind of information outside their repos.
- We interviewed a relatively small sample of 8 teams. Their pain points (Section 8) may not be representative of the rest of their community.

11.4. External Validity

If the results of a study can be generalized to other situations, then the study is externally valid [109]. In other words, 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 [other software (citation redacted)] and MI software to be similar. Except for the domain specific aspects, the trends observed in the current study are similar to that for other research software.

12. Conclusions

Our analysis of the state of the practice for MI domain along nine software qualities strongly indicates that "higher quality" is consistent with community ranking proxies. Although our quality measures are rather shallow, we see this as an advantage. The shallow measures are a proxy for the importance of *first impressions* for software adoption.

Our grading scores indicate that *3D Slicer*, *ImageJ*, *Fiji* and *OHIF Viewer* are the overall top four. 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 7, 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 7, or only appear once, are *Papaya*, *MatrixUser*, *MRIcroGL*, *XMedCon*, *dicompyler*, *DicomBrowser*, *AMIDE*, *3DimViewer*, and *Drishti*.

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

While we did find a reasonable amount of documentation, especially when we consider all MI projects, there were definitely some holes. Some important documentation (test plans and requirements documentation) was missing, and other (contributors' guide, code of conduct, code style guidelines, product roadmap, design documentation, and API documentation) were rare.

Our interviewees proposed strategies to improve the state of the practice, to address the identified pain points, and to improve software quality. To their list (Section 8) we added some of our own recommended strategies (Section 10). The strategies that emerged include increasing documentation, increasing testing by enriching datasets, increasing modularity, using continuous integration, moving to web applications, using linters, increasing peer review, and using design for change to guide modularization.

A deeper understanding of the needs of the MI community will require data beyond what is available in repositories.

Acknowledgements

[redacted for blind review]

Conflict of Interest

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

References

[1] Storer T. Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Comput. Surv.* August 2017, 50(4):47:1–47:32.

[2] Hannay J. E, MacLeod C, Singer J, Langtangen H. P, Pfahl D, Wilson G. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. 2009, pp. 1–8.

[3] Prabhu P, Jablin T. B, Raman A, Zhang Y, Huang J, Kim H, Johnson N. P, Liu F, Ghosh S, Beard S, Oh T, Zoufaly M, Walker D, August D. I. A survey of the practice of computational science. In *SC'11: State of the Practice Reports*. SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Ghezzi C, Jazayeri M, Mandrioli D. Fundamentals of software engineering. Prentice Hall. 2003, Upper Saddle River, NJ, USA, 2nd edition.

[5] ISO/IEC. Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models. Standard. International Organization for Standardization. Mar 2011.

[6] Vaidya O. S, Kumar S. Analytic hierarchy process: An overview of applications. *European Journal of Operational Research*. 2006, 169(1):1–29.

[7] Saaty T. L. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*. 1990, 48(1):9–26.

[8] Björn K. Evaluation of open source medical imaging software: A case study on health technology student learning experience. *Procedia Computer Science*. 01 2017, 121:724–731.

[9] Brühschwein A, Klever J, Hoffmann A.-S, Huber D, Kaufmann E, Reese S, Meyer-Lindenberg A. 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.

[10] Haak D, Page C.-E, Deserno T. A survey of DICOM viewer software to integrate clinical research and medical imaging. *Journal of digital imaging*. 10 2015, 29.

[11] Emms S. 16 best free linux medical imaging software. https://www.linuxlinks.com/medicalimaging/, 2019. [Online; accessed 02-February-2020].

[12] Hasan M. Top 25 best free medical imaging software for linux system. https://www.ubuntupit.com/top-25-best-free-medical-imaging-software-for-linux-system/, 2020. [Online; accessed 30-January-2020].

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

[14] Samala R. 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, 03 2014. [Online; accessed 31-January-2020].

[15] Ahrens J, Geveci B, Law C. Paraview: An end-user tool for large data visualization. *Visualization Handbook*. 01 2005.

[16] Nečas D, Klapetek P. Gwyddion: an open-source software for SPM data analysis. *Central European Journal of Physics*. 01 2012, 10:181–188.

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

[18] SARL P. Osirix lite. https://github.com/pixmeo/osirix, 2019. [Online; accessed 27-May-2021].

[19] Kikinis R, Pieper S, Vosburgh K. *3D Slicer: A Platform for Subject-Specific Image Analysis, Visualization, and Clinical Support*, Vol.3, pp. 277–289. Springer, 01 2014.

[20] Limaye A. Drishti, a volume exploration and presentation tool. In *Proceedings SPIE 8506, Developments in X-Ray Tomography*. Vol. 8506. 10 2012, p. 85060X.

[21] Wollny G. Ginkgo CADx. https://github.com/gerddie/ginkgocadx, 2020. [Online; accessed 27-May-2021].

[22] Jan S, Santin G, Strul D, Staelens S, Assié K, Autret D, Avner S, Barbier R, Bardiès M, Bloomfield P, Brasse D, Breton V, Bruyndonckx P, Buvat I, Chatziioannou A, Choi Y, Chung Y, Comtat C, Donnarieix D, Morel C. GATE: a simulation toolkit for PET and SPECT. *Physics in medicine and biology*. 11 2004, 49:4543–61.

[23] TESCAN. 3dimviewer. https://bitbucket.org/3dimlab/3dimviewer/src/master/, 2020. [Online; accessed 27-May-2021].

[24] Fillard P, Toussaint N, Pennec X. Medinria: DT-MRI processing and visualization software. https://med.inria.fr/, 04 2012.

[25] Papademetris X, Jackowski M, Rajeevan N, Constable R, Staib L. Bioimage suite: An integrated medical image analysis suite. *The Insight Journal*. 08 2005, 1.

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

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

[28] Nolf E, Voet T, Jacobs F, Dierckx R, Lemahieu I. XMedCon – an opensource medical image conversion toolkit. *European Journal of Nuclear Medicine and Molecular Imaging*. 08 2003, 30:S246.

[29] Yushkevich P. A, Piven J, Cody Hazlett H, Gimpel Smith R, Ho S, Gee J. C, Gerig G. User-guided 3D active contour segmentation of anatomical structures: Significantly improved efficiency and reliability. *Neuroimage*. 2006, 31(3):1116–1128.

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

[31] Ziegler E, Urban T, Brown D, Petts J, Pieper S. D, Lewis R, Hafey C, Harris G. J. Open health imaging foundation viewer: An extensible open-source framework for building web-based imaging applications to support cancer research. *JCO Clinical Cancer Informatics*. 2020, 4:336–345, pMID: 32324447.

[32] Chandra S, Dowling J, Engstrom C, Xia Y, Paproki A, Neubert A, Rivest-Hénault D, Salvado O, Crozier S, Fripp J. A lightweight rapid application development framework for biomedical image analysis. *Computer Methods and Programs in Biomedicine*. 07 2018, 164.

[33] Amorim P, Franco de Moraes T, Pedrini H, Silva J. Invesalius: An interactive rendering framework for health care support. In *Advances in Visual Computing*. 12 2015, p.10.

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

[35] Afsar A. Dicom viewer. https://github.com/ayselafsar/dicomviewer, 2021. [Online; accessed 27-May-2021].

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

[37] Liu F, Velikina J, Block W, Kijowski R, Samsonov A. Fast realistic MRI simulations based on generalized multi-pool exchange tissue model. *IEEE Transactions on Medical Imaging*. 10 2016, PP:1–1.

[38] Haehn D. Slice:drop: collaborative medical imaging in the browser. In *SIGGRAH'13: ACM SIGGRAPH 2013 Computer Animation Festival*. 07 2013, pp. 1–1.

[39] Panchal A, Keyes R. Su-gg-t-260: Dicompyler: An open source radiation therapy research platform with a plugin architecture. *Medical Physics - MED PHYS*. 06 2010, 37.

[40] Schindelin J, Arganda-Carreras I, Frise E, Kaynig V, Longair M, Pietzsch T, Preibisch S, Rueden C, Saalfeld S, Schmid B, Tinevez J.-Y, White D, Hartenstein V, Eliceiri K, Tomancak P, Cardona A. Fiji: An open-source platform for biological-image analysis. *Nature methods*. 06 2012, 9:676–82.

[41] Rueden C, Schindelin J, Hiner M, Dezonia B, Walter A, Eliceiri K. Imagej2: Imagej for the next generation of scientific image data. *BMC Bioinformatics*. 11 2017, 18.

[42] Lab C. R. Mricrogl. https://github.com/rordenlab/MRIcroGL, 2021. [Online; accessed 27-May-2021].

[43] Archie K, Marcus D. Dicombrowser: Software for viewing and modifying dicom metadata. *Journal of digital imaging : the official journal of the Society for Computer Applications in Radiology*. 02 2012, 25:635–45.

[44] Szulik K. Don't judge a project by its GitHub stars alone. https://blog.tidelift.com/dont-judge-a-project-by-its-github-stars-alone, December 2017. [Online; accessed 8-Nov-2024].

[45] USGS. USGS (United States Geological Survey) software plannning checklist. https://www.usgs.gov/media/files/usgs-software-planning-checklist, December 2019. [Online; accessed 8-Nov-2014].

[46] Schlauch T, Meinel M, Haupt C. DLR software engineering guidelines. https://zenodo.org/records/1344612, August 2018. [Online; accessed 8-Nov-2014].

[47] Brett A, Cook J, Fox P, Hinder I, Nonweiler J, Reeve R, Turner R. Scottish Covid-19 response consortium. https://github.com/ScottishCovidResponse/modelling-software-checklist/blob/main/software-checklist.md, August 2021. [Online; accessed 8-Nov-2014].

[48] Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal T. K. Good enough practices in scientific computing. *CoRR*. 2016, abs/1609.00037.

[49] Smith B, Bartlett R, Developers x. xSDK community package policies. https://doi.org/10.6084/m9.figshare.4495136.v6, Dec 2018.

[50] Heroux M. A, Bieman J. M, Heaphy R. T. Trilinos developers guide part II: ASC softwar quality engineering practices version 2.0. https://faculty.csbsju.edu/mheroux/fall2012_csci330/TrilinosDevGuide2.pdf, April 2008.

[51] Thiel C. EURISE network technical reference. https://technical-reference.readthedocs.io/en/latest/, 2020. [Online; accessed 8-Nov-2014].

[52] van Gompel M, Noordzij J, de Valk R, Scharnhorst A. Guidelines for software quality, CLARIAH task force 54.100. https://github.com/CLARIAH/software-quality-guidelines/blob/master/softwareguidelines.pdf, September 2016. [Online; accessed 8-Nov-2014].

[53] Orviz P, García Á. L, Duma D. C, Donvito G, David M, Gomes J. A set of common software quality assurance baseline criteria for research projects. https://digital.csic.es/handle/10261/160086, 2017.

[54] Prana G. A. A, Treude C, Thung F, Atapattu T, Lo D. Categorizing the content of GitHub README files, 2018.

[55] Smith W. S, Koothoor N. A document-driven method for certifying scientific computing software for use in nuclear safety analysis. *Nuclear Engineering and Technology*. April 2016, 48(2):404–418.

[56] Heaton D, Carver J. C. Claims about the use of software engineering practices in science. *Information and Software Technology*. November 2015, 67(C):207–219.

[57] Sanders R, Kelly D. Dealing with risk in scientific software development. *IEEE Software*. July/August 2008, 4:21–28.

[58] Nguyen-Hoan L, Flint S, Sankaranarayana R. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10, New York, NY, USA, 2010. ACM.

[59] Wiese I. S, Polato I, Pinto G. Naming the pain in developing scientific software. *IEEE Software*. 2019, pp. 1–10.

[60] Carver J. C, Kendall R. P, Squires S. E, Post D. E. 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*. Washington, DC, USA, 2007. IEEE Computer Society.

[61] Segal J, Morris C. Developing scientific software. *IEEE Software*. July/August 2008, 25(4):18–20.

[62] Smith W. S. A rational document driven design process for scientific computing software. In: Carver J. C, Hong N. C, Thiruvathukal G (Eds.). *Software Engineering for Science*, chapter Section I – Examples of the Application of Traditional Software Engineering Practices to Science, pp. 33–63. Taylor & Francis, Oxfordshire, 2016.

[63] Smith W. S, Lai L, Khedri R. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*. February 2007, 13(1):83–107.

[64] Brown T. 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, June 2015. [Online; accessed 8-Nov-2014].

[65] Heroux M. A, Bernholdt D. E. 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, 2018.

[66] Institute S. S. Online sustainability evaluation. https://www.software.ac.uk/resources/online-sustainability-evaluation, 2022.

[67] Johanson A. N, Hasselbring W. Software engineering for computational science: Past, present, future. *Computing in Science & Engineering*. 2018, Accepted:1–31.

[68] Faulk S, Loh E, Vanter M. L. V. D, Squires S, Votta L. G. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science Engineering*. Nov 2009, 11(6):30–39.

[69] de Souza M. R, Haines R, Vigo M, Jay C. What makes research software sustainable? an interview study with research software engineers. *CoRR*. 2019, abs/1903.06039.

[70] Wilson G. V. Where's the real bottleneck in scientific computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist*. 2006, 94(1).

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

[72] Smith W. S. Beyond software carpentry. In *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*. 2018, pp. 1–8.

[73] Carver J. C, Weber N, Ram K, Gesing S, Katz D. S. A survey of the state of the practice for research software in the united states. *PeerJ Computer Science*. 2022, 8(e963).

[74] Zadka M. How to open source your Python library. https://opensource.com/article/18/12/tips-open-sourcing-python-libraries, Dec 2018. [Online; accessed 8-Nov-2014].

[75] Jung R, Gundlach S, Hasselbring W. Thematic domain analysis for ocean modeling. *Environmental Modelling & Software*. Jan 2022, p. 105323.

[76] Ackroyd K. S, Kinder S. H, Mant G. R, Miller M. C, Ramsdale C. A, Stephenson P. C. Scientific software development at a research facility. *IEEE Software*. July/August 2008, 25(4):44–51.

[77] Easterbrook S. M, Johns T. C. Engineering the software for understanding climate change. *Comuting in Science & Engineering*. November/December 2009, 11(6):65–74.

[78] Segal J. When software engineers met research scientists: A case study. *Empirical Software Engineering*. October 2005, 10(4):517–536.

[79] Kelly D. Industrial scientific software: A set of interviews on software development. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '13, Riverton, NJ, USA, 2013. IBM Corp.

[80] Kelly D. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software*. 2015, 109:50–61.

[81] Parnas D. L, Clements P. C. A rational design process: How and why to fake it. *IEEE transactions on software engineering*. February 1986, 12(2):251–257.

[82] Pinto G, Wiese I, Dias L. F. How do scientists develop and use scientific software? An external replication. In *Proceedings of 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. February 2018, pp. 582–591.

[83] Pinto G, Steinmacher I, Gerosa M. A. More common than you think: An in-depth study of casual contributors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol.1. 2016, pp. 112–123.

[84] Gewaltig M.-O, Cannon R. Quality and sustainability of software tools in neuroscience. *Cornell University Library*. May 2012, p. 20 pp.

[85] Goble C. Better software, better research. *IEEE Internet Computing*. 2014, 18(5):4–8.

[86] Katerbow M, Feulner G. Recommendations on the development, use and provision of Research Software. https://zenodo.org/records/1172988, March 2018. [Online; accessed 8-Nov-2014].

[87] Howison J, Bullard J. Software in the scientific literature: Problems with seeing, finding, and using software mentioned in the biology literature. *J. Assoc. Inf. Sci. Technol.* sep 2016, 67(9):2137–2155.

[88] Hannay J. E, MacLeod C, Singer J, Langtangen H. P, Pfahl D, Wilson G. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. SECSE '09, Washington, DC, USA, 2009. IEEE Computer Society.

[89] Kruchten P, Nord R. L, Ozkaya I. Technical debt: From metaphor to theory and practice. *IEEE Software*. 2012, 29(6):18–21.

[90] Lethbridge T, Singer J, Forward A. How software engineers use documentation: the state of the practice. *IEEE Software*. 2003, 20(6):35–39.

[91] Humble J, Farley D. G. Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley. 2010, Upper Saddle River, NJ.

[92] Shahin M, Ali Babar M, Zhu L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*. 2017, 5:3909–3943.

[93] Fowler M. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, May 2006. [Online; accessed 8-Nov-2014].

[94] Wang X, Peng Y, Lu L, Lu Z, Bagheri M, Summers R. ChestX-ray8: Hospital-scale chest X-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. *arXiv:1705.02315*. 05 2017.

[95] Prior F, Smith K, Sharma A, Kirby J, Tarbox L, Clark K, Bennett W, Nolan T, Freymann J. The public cancer radiology imaging collections of the cancer imaging archive. *Scientific Data*. 09 2017, 4:sdata2017124.

[96] Smirniotopoulos J. Medpix medical image database. https://www.researchgate.net/publication/267597633_MedPix_Medical_Image_Database, 10 2014. [Online; accessed 8-Nov-2014].

[97] Bilic P, Christ P. F, Vorontsov E, Chlebus G, Chen H, Dou Q, Fu C, Han X, Heng P, Hesser J, Kadoury S, Konopczynski T. K, Le M, Li C, Li X, Lipková J, Lowengrub J. S, Meine H, Moltz J. H, Pal C, Piraud M, Qi X, Qi J, Rempfler M, Roth K, Schenk A, Sekuboyina A, Zhou P, Hülsemeyer C, Beetz M, Ettlinger F, Grün F, Kaissis G, Lohöfer F, Braren R, Holch J, Hofmann F, Sommer W. H, Heinemann V, Jacobs C, Mamani G. E. H, van Ginneken B, Chartrand G, Tang A, Drozdzal M, Ben-Cohen A, Klang E, Amitai M. M, Konen E, Greenspan H, Moreau J, Hostettler A, Soler L, Vivanti R, Szeskin A, Lev-Cohain N, Sosna J, Joskowicz L, Menze B. H. The liver tumor segmentation benchmark (lits). *CoRR*. 2019, abs/1901.04056.

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

[99] Wikipedia. Lint (software). https://en.wikipedia.org/wiki/Lint_(software), March 2022. [Online; accessed 8-Nov-2014].

[100] SourceLevel. What is a linter and why your team should use it? https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it, March 2022. [Online; accessed 8-Nov-2014].

[101] Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 2018, pp. 181–190.

[102] Jones C. Measuring defect potentials and defect removal efficiency. *Crosstalk, The Journal of Defense Software Engineering*. June 2008, 21(6):11–13.

[103] Ebert C, Jones C. Embedded software: Facts, figures, and future. *Computer*. April 2009, 42(4):42–52.

[104] Kelly D, Shepard T. Task-directed software inspection technique: an experiment and case study. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. IBM

Press. 2000, p.6.

[105] Wilson G, Aruliah D. A, Brown C. T, Chue Hong N. P, Davis M, Guy R. T, Haddock S. H. D, Huff K. D, Mitchell I. M, Plumbley M. D, Waugh B, White E. P, Wilson P. Best practices for scientific computing. *PLoS Biol*. January 2014, 12(1):e1001745.

[106] Stewart G, others. A Roadmap for HEP Software and Computing R&D for the 2020s. *arXiv*. 2017.

[107] Parnas D. L. On the criteria to be used in decomposing systems into modules. *Comm. ACM*. December 1972, 15(2):1053–1058.

[108] Ampatzoglou A, Bibi S, Avgeriou P, Verbeek M, Chatzigeorgiou A. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Information and Software Technology*. 02 2019, 106.

[109] Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*. Dec 2009, 14(2):131–164.

[110] van Vliet H. Software engineering (2nd ed.): Principles and practice. John Wiley & Sons, Inc. 2000, New York, NY, USA.