State of the Practice for Medical Imaging Software

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

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

Abstract

We present a general method to assess the state of the practice for Scientific Computing (SC) software and apply this method to Medical Imaging (MI) software. We selected 29 MI software projects from 48 candidates, assessed 10 software qualities (installability, correctness/ verifiability, reliability, robustness, usability, maintainability, reusability, understandability, visibility/transparency and reproducibility) by answering 103 questions for each software project, and interviewed 8 of the 29 development teams. Based on the quantitative data for the first 9 qualities, we ranked the MI software with the Analytic Hierarchy Process (AHP). The top three software products were *3D Slicer*, *ImageJ*, and *OHIF Viewer*. By interviewing the developers, we identified three major pain points: i) lack of resources; ii) difficulty balancing between compatibility, maintainability, performance, and security; and, iii) lack of access to real-world datasets for testing. For future MI software projects, we propose adopting test-driven development, using continuous integration and continuous delivery (CI/CD), using git and GitHub, maintaining good documentation, supporting third-party plugins or extensions, considering web application solutions, and improving collaboration between different MI software projects.

Keywords: medical imaging, scientific computing, software engineering, software quality, Analytic Hierarchy Process, developer interviews

1. Introduction

We aim to study the state of software development practice for Medical Imaging (MI) software. MI tools use images of the interior of the body (from sources such as Magnetic Resonance Imaging (MRI), Computed Tomography (CT), Positron Emission Tomography (PET) and Ultrasound) to provide information for diagnostic, analytic, and medical applications (Administration, 2021; Wikipedia contributors, 2021d; Zhang et al., 2008). Given the importance of MI software and the high number of competing software projects, we wish to understand the merits and drawbacks of the current development processes, tools and methodologies. We aim to assess through a software engineering lens the quality of the existing software with the hope of highlighting standout examples, understanding current pain points and providing guidelines and recommendations for future development.

1.1. Purpose

Not only do we wish to gain insight into the state of the practice for MI software, we also wish to understand the development of research software in general. We wish to understand the impact of the often cited gap, or chasm, between software engineering and scientific programming (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 any proper debugger for their software.

To gain insights, we devised five research questions, which can be applied to MI, and to other domains, of research software (Smith et al., 2021):

RQ1. What artifacts are present in current software projects? What role does documentation play in the projects? What are the developers' attitude toward it?

RQ2. What tools are used in the development of current software packages?

RQ3. What principles, processes, and methodologies are used in the development of current software packages?

RQ4. What are the pain points for developers working on research software projects? What aspects of the existing processes, methodologies, and tools do they consider as potentially needing improvement? What changes to processes, methodologies, and tools can improve software development and software quality?

RQ5. What is the current status of the following software qualities for the projects? What actions have the developers taken to address them?

- Installability
- Correctness & Verifiability
- Reliability
- Robustness
- Usability
- Maintainability
- Reusability
- Understandability
- Visibility/Transparency
- Reproducibility

RQ6. How does software designated as high quality by this methodology compare with top-rated software by the community?

1.2. Overview of the Methodology

We designed a general methodology to assess the state of the practice for SC software (Smith et al., 2021). Details can be found in Section **??**. This methodology builds off prior work to assess the state of the practice for such domains as Geographic Information Systems (Smith et al., 2018a), Mesh Generators (Smith et al., 2016), Seismology software (Smith et al., 2018c), and Statistical software for psychology (Smith et al., 2018b). In keeping with the previous methodology, we have maintained the constraint that the work load for measuring a given domain should be feasible for a team as small as one person, and for a short time, ideally around a person month of effort. A person month is considered to be 20 working days (4 weeks in a month, with 5 days of work per week) at 8 person hours per day, or $20 \cdot 8 = 160$ person hours.

With our methodology, we first choose an SC domain (in the current case MI) and identify a list of about 30 software packages. We approximately measure the qualities of each package by filling in a grading template (forward ref?). 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. (forward ref?). With the quantitative data in the grading template, we rank the software with the Analytic Hierarchy Process (AHP) (forward ref?). After this, as another addition to our previous methodology, we interview some of the development teams to further understand the status of their development process (forward ref?). Finally, we summarize the results and propose recommendations for future SC software development (forward ref?).

1.3. Scope

We need to restrict the scope of MI software that we consider so that the measurement exercise is feasible within a person month of effort. MI considers a broad range of different problems. According to Bankman (2000), MI software deals with six different basic problems, while Angenent et al. (2006) pointed out that four fundamental problems are solved by MI software. While both mentioned Segmentation, Registration, and Visualization of medical images, Bankman also included Enhancement, Quantification, and three functions for MI archiving and telemedicine systems (Compression, Storage, and Communication) (Bankman, 2000). On the other hand, Angenent et al. (2006) included Simulation. According to Wikipedia contributors (Wikipedia contributors, 2021c), MI software has primary functions in Segmentation, Registration, Visualization (including the basic display, reformatted views, and 3D volume rendering), Statistical Analysis, and Image-based Physiological Modelling. As Kim et al. (2011) describes, the general steps of medical image analysis after obtaining digital data include Enhancement, Segmentation, Feature Extraction, Classification, and Interpretation. Besides the above major functions, some MI software provides supportive functions. For example, with Tool Kit libraries VTK (Schroeder et al., 2006) and ITK (McCormick et al., 2014), developers build software with Visualization and Analysis functions; Picture Archiving and Communication System (PACS) helps users to economically store and conveniently access images (Choplin et al., 1992).

Based on our literature survey, we divided MI software into five sub-groups and several sub-sub-groups by their major functions, as shown in Figure 1.

To keep the data collection and analysis feasible, we limited the scope of the software to the software packages providing the Visualization tools and functions in this project.

[Roadmap - instead sprinkle forward references throughout the introduction?]

2. Background

When designing a method for evaluating the state of the practice of domain-specific software, we included a step to select domain and software. Knowledge of different software categories is essential for the selection. To compare and rank the software qualities with the grading template in Appendix **??**, we need the definitions of qualities and the AHP.

In this section, we introduce the relevant software categories (Section 2.1). We also cover the software quality definitions (Section 2.2) and an overview of the AHP (Section 2.3).

2.1. Software Categories

We target specific software categories to narrow down the scope when selecting software packages for measuring. One way to categorize them is grouping by functions, such as analyzing the scope for the MI software (Section 1.3), which requires specific domain knowledge. In this section, we discuss three common software categories that we can apply to all SC domains: Open source software (OSS), freeware, and commercial software.

2.1.1. Open Source Software

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

2.1.2. Freeware

Freeware is software that can be used free of charge. Unlike with OSS, the authors of freeware do not allow users to access or modify the source code of the software (Project, 2006). The term *freeware* should not be confused with *free software*, which is similar to OSS. To the end-users, the differences between freeware and OSS often do not bother them. The fact that these products are free of charge is likely to make them popular with many users. However, software developers, end-users who wish to modify the source code, and researchers looking for insight into software development process will find the inaccessible source code a problem.

2.1.3. Commercial Software

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

For some specific software, the backgrounds of commercial software developers often differ from the ones of non-commercial OSS. In such a case, the former is usually the product of software engineers, and the latter is likely to have developers who work in the domain and are also end-users of the products. One example of such software is SC software, since the developers need to utilize their domain-specific during the development process (Wilson et al., 2014).

2.2. Software Quality Definitions

This section lists the definitions of 10 software qualities, which are from Smith et al. (Smith et al., 2020). We aim to measure each of them for selected SC software packages. The order of the first nine qualities follows our grading template in Appendix **??**. We do not measure *reproducibility* with the grading template, but discuss it with the developers by interviews.

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

- **Correctness & Verifiability** A program is correct if it behaves according to its stated specifications. Verifiability is the extent to which a set of tests can be written and executed, to demonstrate that the delivered system meets the specification (Ghezzi et al., 2003).

- **Reliability** The probability of failure-free operation of a computer program in a specified environment for a specified time, i.e., the average time interval between two failures also known as the mean time to failure (MTTF) (Musa et al., 1987) (Ghezzi et al., 2003).

- **Robustness** Software possesses the characteristic of robustness if it behaves "reasonably" in two situations: i) when it encounters circumstances not anticipated in the requirements specification, and ii) when the assumptions in its requirements specification are violated (Ghezzi et al., 1991) (Boehm, 2007).

- **Usability** "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" (ISO/TR, 2002) (ISO/TR, 2018).

- **Maintainability** The effort with which a software system or component can be modified to i) correct faults; ii) improve performance or other attributes; iii) satisfy new requirements (IEEE, 1991) (Boehm, 2007).

- **Reusability** "The extent to which a software component can be used with or without adaptation in a problem solution other than the one for which it was originally developed" (Kalagiakos, 2003).

- **Understandability** "The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use" (ISO, 2001).

- **Visibility/Transparency** The extent to which all of the steps of a software development process and the current status of it are conveyed clearly (Ghezzi et al., 1991).

- **Reproducibility** "A result is said to be reproducible if another researcher can take the original code and input data, execute it, and re-obtain the same result" (Benureau and Rougier, 2017).

2.3. Analytic Hierarchy Process

To generate ranking scores for a set of software packages, we use AHP, which utilizes pairwise comparisons between all of the packages. Thomas L. Saaty developed this tool, and people widely used it to make and analyze multiple criteria decisions (Vaidya and Kumar, 2006). AHP organizes multiple criteria factors in a hierarchical structure and uses pairwise comparisons between alternatives to calculate relative ratios (Saaty, 1990).

For a project with m criteria, we can use an $m \times m$ matrix A to record the relative importance between factors. When comparing criterion i and criterion j, the value of A_{ij} is decided as follows, and the value of A_{ji} is $1/A_{ij}$ (Saaty, 1990),

- $A_{ij} = 1$ if criterion i and criterion j are equally important;
- $A_{ij} = 9$ if criterion i is extremely more important than criterion j;
- A_{ij} equals to an integer value between 1 and 9 according the the relative importance of criterion i and criterion j.

The above process assumes that criterion i is not less important than criterion j, otherwise, we need to reverse i and j and determine A_{ji} first, then $A_{ij} = 1/A_{ji}$.

The priority vector w can be calculated by solving the following equation (Saaty, 1990),

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

where λ_{max} is the maximal eigenvalue of A.

In this project, w is approximated with the classic *mean of normalized values* approach (Ishizaka and Lusti, 2006),

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

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

In this project, the first nine software qualities mentioned in Section 2.2 are the criteria ($m = 9$), while 29 software packages ($n = 29$) are compared for each of the m criteria. The software are evaluated with the grading template in Appendix **??** and a subjective score from one to ten is given for each quality for each package. For each quality, for a pair of packages i and j, such that $score_i >= score_j$, the difference between two scores is $diff_{ij} = score_i - score_j$. The relationship between $A_{ij} = 1$ and $diff_{ij}$ is as follows:

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

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

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

3. Methodology

We designed a general process for evaluating the state of the practice of domain-specific SC software, that we instantiate for a specific scientific domain.

Our method involves four steps:

1. choosing a software domain (Section 3.1);
2. collecting and filtering software packages (Section 3.2);
3. grading the selected software (Section 3.3);
4. interviewing development teams for further information (Section 3.4).

Section 3.5 presents an example of how we applied the method on the MI domain.

3.1. Domain Selection

Our methods are generic for any SC software, but they need to be applied to a specific domain. When choosing a candidate domain, we prefer one with a large number of active OSS. The reason is that we aim to finalize a list of 30 software packages (Smith et al., 2021) after the screening step. For example, we remove the ones without recent updates or specific functions. Besides, we prefer OSS projects because our grading method requires access to the code. In addition, we prefer a domain with an active community developing and using the software. As a result, it is easier to invite enough developers for interviews.

We prefer 30 software packages providing similar functions or falling into different sub-groups depending on our research purpose. So the domain needs to have enough candidates in one sub-group or enough sub-groups to cross-compare.

We also prefer domains in which our team has expertise. We invite domain experts to join and support our projects. They help us in many aspects, such as vetting the software list and interview questions.

3.2. Software Product Selection

The process of selecting software packages contains two steps: i) identify software candidates in the chosen domain, ii) filter the list according to our needs (Smith et al., 2021).

3.2.1. Identify Software Candidates

We start with finding candidate software in publications of the domain. Then, we search various websites, such as GitHub, swMATH and the Google search results for software recommendation articles. We should also include the ones suggested by the domain experts (Smith et al., 2021).

3.2.2. Filter the Software List

The goal is to build a software list with a length of about 30 (Smith et al., 2021).

The only "mandatory" requirement is that the software must be OSS, as defined in Section 2.1.1. We need this because evaluating some software qualities requires the source code.

The other filters are optional, and we consider them according to the number of software candidates and the objectives of the research project. We apply them in the following priority order:

1. The functions and purpose of the software. An SC domain often contains software with various functions and purposes. For example, some MI software packages are Tool Kit for developers to use, and some others offer Visualization function to end-users. We have two options:
   - selecting a set of software with the same major function. In this case, we can use the identical process to assess all packages, e.g., same input to measure *Robustness*. Also, when we give impression scores to qualities such as *Installability* and *Usability*, the results are more comparable. Thus, it is more feasible to collect the results and rank the software.

- selecting software from a set of different sub-groups. For example, we can choose 10 MI software from each of the three sub-groups: Visualization, Tool Kit, and PACS. The downside: we may need different processes to measure each sub-group; it may be less accurate to mix all three sub-groups and rank them together. The benefit: we can cross-compare the development processes between the sub-groups.

2. The version control tool. The empirical measurement tools listed in Section 3.3.2 only work on projects using Git, so we prefer software with Git. Some manual steps in empirical measurement depend on a few metrics of GitHub, which makes projects held on GitHub more favored (Smith et al., 2021).
3. The age of the software. Some of the OSS projects may experience a lack of recent maintenance. So we eliminate packages without recent updates, unless they are still popular and highly recommended by the domain users (Smith et al., 2021). We consider a software project as "alive" if it has any update within the last 18 months; otherwise, we mark it as "dead".

The order of filters 2 and 3 is flexible. We adjust it according to the number of software packages affected by the filters, and the number of ones remaining on the list.

3.2.3. Vet the Software List

Before showing our filtered list to the domain experts, we ask them to list their top 10 software in the domain. Then, we cross-compare the two lists and discuss the commonality and variability. In addition, we ask them to vet our filtered list. They provide views on whether the list is reasonable. We also use their opinions for a filtering process. For example, if a software package is not OSS and has had no updates for a long time, but the domain experts identify it as a valuable product, we still consider it in our final list.

3.3. Grading Software

We grade the selected software using a template (Section 3.3.1) and a specific empirical method (Section 3.3.2). Some technical details for the measurements are in Section 3.3.3.

3.3.1. Grading Template

The full grading template can be found in Appendix **??**. The template contains 103 questions that we use for grading software products. Figure 2 shows an example of this grading template.

We use the first section of the template to collect general information, such as the 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, etc. Information in this section helps us understand the projects better and may be helpful for further analysis, but it does not directly affect the grading scores.

We designed the following nine sections in the template for the nine software qualities mentioned in Section 2.2. For each quality, we ask several questions and the typical answers are among the collection of “yes”, “no”, “n/a”, “unclear”, a number, a string, a date, a set of strings, etc. Each quality needs an overall score between 1 and 10 based on all the previous questions. For some qualities, we perform surface measurements, which allow us to measure all packages with reasonable efforts. The surface measurements reveal some traits of a underlying quality, but may not fully represent it.

- **Installability** We check the existence and quality of installation instructions. The user experience is also an important factor, such as the ease to follow the instructions, number of steps, automation tools, and the prerequisite steps for the installation. If any problem interrupts the process of installation or uninstallation, we give a lower score to this quality. We also record the Operating System (OS) for the installation test and whether we can verify the installation.

- **Correctness & Verifiability** For *correctness*, we check the projects to identify the techniques 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 the requirements specifications, theory manuals, and getting started tutorials. If a getting started tutorial exists and provides expected results, we follow it and check if the outputs match.

- **Surface Reliability** We check whether the software breaks during the installation and the operations following a getting started tutorials, whether there are descriptive error messages, and if we can recover the process after an error. We use the software to load damaged images during the assessment to *surface robustness*, and lower its score for *surface reliability* if a software product "break" in this process.

- **Surface Robustness** We check how the software handles unexpected/unanticipated input. For example, we prepare broken image files for MI software packages with the function to 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 with the unexpected/unanticipated ones.

- **Surface Usability** We examine the documentation of the projects, and we consider software with a getting started tutorial and a user manual easier to use. Meanwhile, we check if users have any channels to get support. We also record our impressions and user experiences when testing the software. Easy-to-use graphical user interfaces give us a better experience, which leads to better scores.

- **Maintainability** We search the projects' documents and identify the process of contributing and reviewing code. We believe that the artifacts of a project - including source code, documents, building scripts, etc. - can significantly affect its *maintainability*. Thus we check each project for its artifacts, such as API documentation, bug tracker, release notes, test cases, build files, version control, etc. We also check the tools supporting issue tracking and version control, the percentages of closed issues, and the proportion of comment lines in code.

- **Reusability** We count the total number of code files for each project. Projects with a large number of components provide more choices to reuse. Furthermore, well-modularized code, which tends to have smaller parts in separated files, is typically easier to reuse. Thus, we consider the projects with more code files and less code lines per file to be more reusable. We use a command-line tool *scc* to count the number of text-based files and LOC for all projects. We also decide that the projects with API documentation can deliver better *Reusability*.

- **Surface Understandability** We randomly examine 10 code files. We check the code's style within each file, such as whether the identifiers, parameters, indentation, and formatting are consistent, whether the constants (other than 0 and 1) are hardcoded, and whether the developers modularized the code. 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 algorithms in the code.

- **Visibility/Transparency** To measure this quality, we check the existing documents to find out 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. If any information is missing or poorly conveyed, the *visibility/transparency* will be lower.

For some qualities, the empirical measurements also affect the score. We use tools to extract information from the source code repositories. For projects held on GitHub, we manually collect additional metrics, such as the stars of the GitHub repository, and the numbers of open and closed pull requests. Section 3.3.2 presents more details about the empirical measurements.

3.3.2. Empirical Measurements

We use two command-line tools for the empirical measurements. One is *GitStats* that generates statistics for git repositories and displays outputs in the form of web pages (Gieniusz, 2019); the other one is Sloc Cloc and Code (as known as *scc*) (Boyter, 2021), aiming to count the lines of code, comments, etc.

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.

Additionally, we manually collect information for projects held on GitHub, 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. A git repository can have a creation date much earlier than the first day on GitHub. For example, the developers created the git repository of *3D Slicer* in 2002, but did not upload a copy of it to GitHub until 2020. We get the creation date of the GitHub copy by using API *https://api.github.com/repos/:owner/:repository* (e.g., https://api.github.com/repos/slicer/slicer). In the response, the value of "created_at" is what we want. The number of months a repository has been on GitHub helps us understand the average change of metrics over time, e.g., the average new stars per month.

These empirical measurements help us from two aspects. Firstly, they help us with getting a project overview faster and more accurately. For example, the number of commits over the last 12 months shows how active this project has been, and the number of stars and forks may reveal its popularity. 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 the *maintainability*.

3.3.3. Technical Details

To test the software on a "clean" system, we create a new virtual machine (VM) for each software and only install the necessary dependencies before measuring. We make all 30 VMs on the same computer, one at a time, and destroy them after measuring.

We spend about two hours grading each package, unless we find technical issues and need more time to resolve them. In most of the situation, we finish all the measurements for one software on the same day.

3.4. Interview Methods

This section introduces our interview questions (Section 3.4.1), method of selecting interviewees (Section 3.4.2), and interview process (Section 3.4.3).

3.4.1. Interview Questions

We designed a list of 20 questions to guide our interviews, which can be found in Section 5 and Appendix **??**.

Some questions are about the background of the software, the development teams, the interviewees, and how they organize the projects. We also ask about their understandings of the users. Some questions focus on the current and past difficulties, and the solutions the team has found or will try.

We also discuss the importance and current situations of documentation. A few questions are about specific software qualities, such as *maintainability*, *understandability*, *usability*, and *reproducibility*.

The interviews are semi-structured based on the question list; we ask follow-up questions when necessary. Based on our experience, the interviewees usually bring up some exciting ideas that we did not expect, and it is worth expanding on these topics.

3.4.2. Interviewee Selection

For a software list with a length of roughly 30, we aim to interview about ten development teams. Interviewing multiple individuals from each team gives us more comprehensive information, but a single engineer well-knowing the project is also sufficient.

Ideally, we select projects after the grading measurements and prefer the ones with higher overall scores. However, if we do not find enough participants, we reach all teams on the list. As mentioned in Section 3.5.4, when we applied this process to the MI domain, we eventually contacted all teams.

We try to find the contacts of the teams on the projects' websites, such as the official web pages, repositories, publications, and bio pages of the teams' institutions. Then, we send at most two emails to one contact asking for its participation before receiving any replies. We operate the invitation according to our ethics approval, such as the one in Appendix **??**. For example, we ask for participants' consent before interviewing them, recording the conversation, or including it in our report.

3.4.3. Interview Process

Before contacting any interviewee candidate, we need to receive ethics clearance from the McMaster University Research Ethics Board. Since the members of the development teams are usually around the world, we organize these interviews as virtual meetings online with Zoom. After receiving consent from the interviewees, we also record and transcribe our discussions.

3.5. Applying the Method to MI

This section shows an overview of applying our method to the MI domain.

3.5.1. Domain Selection

Based on the principles in Section 3.1, we selected the MI domain and the sub-group of software with the Visualization function shown in Figure

1. We also included Dr. Michael Noseworthy, a 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, and some of his students as the MI domain experts in our team.

3.5.2. Software Product Selection

By using the method in Section 3.2.1, we identified 48 MI software projects as the candidates from publications (Björn, 2017) (Brühschwein et al., 2019) (Haak et al., 2015), online articles related to the domain (Emms, 2019) (Hasan, 2020) (Mu, 2019), forum discussions related to the domain (Samala, 2014), etc. Appendix **??** shows all 48 software packages.

Guided by the method in Section 3.2.2, we filtered the list with a process as follows:

1. Among them, there were eight that we could not find their source code, such as *MicroDicom*, *Aliza*, and *jivex*. These packages are likely to be freeware defined in Section 2.1.2 and not OSS. So following guidelines in Section 3.2.2 we removed them from the list.
2. Next, we focused on the MI software providing Visualization functions, as described in Section 1.3. Seven of the software on the list were Tool Kits or libraries for other software to use as dependencies, but not for end-users to view medical images, such as *VTK*, *ITK*, and *dcm4che*; another three were PACS. We also eliminated these from the list.
3. Finally, we removed *Open Dicom Viewer* from the list because it had not received any updates for a long time (since 2011). After that, only *MatrixUser* and *AMIDE* were considered as "dead". However, both of them had much more recent updates (after 2017) than *Open Dicom Viewer*.

We still preferred projects using git and GitHub and being updated recently, but did not apply this filter since packages were already below 30. Even without this filter, 27 out of the 29 software packages on the filtered list used git, and 24 chose GitHub.

Following the process in Section 3.2.3, our domain experts provided a list of top software that contains 12 software products (Table 1). We compared two lists and found six common ones.

We included *Mango* in the initial list, but removed it because it was not OSS. However, we kept *Papaya*, a the web version of *Mango*. Instead of

Software	On both lists
3D Slicer	X
Horos	X
ImageJ	X
Fiji	X
AFNI	
FSL	
Freesurfer	
Mricron	X
Mango	X
Tarquin	
Diffusion Toolkit	
MRItrix	

Table 1: Top software by the MI domain experts

MRIcron, we chose *MRIcroGL*, because *MRIcron* development had moved to *MRIcroGL* (Rorden, 2021).

Six software packages on the domain experts' list were not on our filtered list. We believed their primary function was Analysis mentioned in Section 1.3. Thus, we did not include them in our final list.

After vetting our filtered list, the domain experts believed it was reasonable and did not identify any problem. Thus, as shown in Appendix **??**, eventually, we had 29 software products on the final list.

3.5.3. Grading Software

Then we followed the steps in Section 3.3 to measure and grade the software. 27 out of the 29 packages are compatible with two or three different OS 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, most likely Windows.

3.5.4. Interviews

We received ethics clearance from the McMaster University Research Ethics Board (Appendix **??**). Going through the interview process in Section 3.4, we contacted all of the 29 teams. Members from eight teams responded and agreed to participate. As a result, we interviewed nine developers and architects from the eight teams.

4. Measurement Results

As discussed in Section 3.3, we use a grading template and a empirical method to measure the selected software. We applied this step to the MI domain (Section 3.5.3). This section shows the summary of the measurement results. The detailed data can be found in the repository https://data.mendeley.com/datasets/k3pcdvdzj2/1. This section contains part of the answers to RQ5.

Table 2 shows the 29 software packages that we measured, along with some summary data collected in the year 2020. As mentioned in Section 3.3.1, we used *scc* (Section 3.3.2) to count the Lines of Code (LOC), excluding the comment and blank lines. We arrange the items in the descending order of the LOC. We found the initial release dates (Rlsd) in their documents for most projects and marked the two unknown dates with "?". We used the date of the latest change to each code repository to decide the latest update. We found out funding information (Fnd) for only eight projects.

We counted the number of contributors (NOC). We considered anyone who made at least one accepted commit to the source code as a contributor. Thus, the NOC is not usually the same as the number of long-term members. Many of these projects received change requests and code from the community, such as pull requests and git commits on GitHub.

Table 2 also shows the supported OS for each software package. Twenty-five of them could work on all three OSs: Windows (W), macOS (M), and Linux (L). However, there was a significant difference in the philosophy to achieve cross-platform compatibility. Most of them were native software products, but five were naturally platform-independent web applications, as shown in column "Web".

Most of the projects used more than one programming language, including a primary language that the developers used the most. Figure **??** shows the primary languages versus the number of projects using them.

We failed to install *DICOM Viewer*, so we could not test its *surface reliability* and *surface robustness*. We kept this software on our list because the other seven qualities do not rely on a successful installation. Besides, the *DICOM Viewer* team built it as a plugin software for NextCloud (https://apps.nextcloud.com/) platform, which was a unique choice we had not seen before. We wanted to keep it to enrich the diversity.

4.1. Installability

Figure 3 lists the scores of *installability*.

We found installation instructions for 16 projects. Among the ones without instructions, *BioImage Suite Web* and *Slice:Drop* are web applications with online versions to use, thus they do not need installation. Installing 10 of the projects required extra dependencies. Five of them are the web applications in Table 2, and depended on a browser; *dwv*, *OHIF Viewer*, and *GATE* needed extra dependencies to build; *ImageJ* and *Fiji* needed an unzip tool; *MatrixUser* was based on Matlab; *DICOM Viewer* needed to work on a Nextcloud platform.

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

dwv, *GATE*, and *DICOM Viewer* showed severe problems. We could not install them. We spent a reasonable amount of time on these problems, then considered them as major obstacles for normal users if we could not figure out any solutions. We suspect that only a part of the users faced the same problems, and given a lot of time, we might be able to find solutions. However, the difficulties greatly impacted the installation experiences, and we graded these software packages with lower scores. For example, *dwv* and *GATE* had the option to build from the source code, and we failed the building processes following the instructions. Although we could not locally build them, we could use a deployed online version for *dwv*, and a VM version for *GATE*. With those, we finished all the measurements for them. Furthermore, *DICOM Viewer* depended on the NextCloud platform, and we could not successfully install the dependency.

MatrixUser has a lower score because it depended on Matlab. We considered installing Matlab takes many more steps and time, and some users may not have a license to use Matlab.

4.2. Correctness & Verifiability

The scores of *correctness & verifiability* are shown in Figure 4. Generally speaking, the packages with higher scores adopted more techniques to improve *correctness*, and had better documents for us to verify it.

After examining the source code, we could not find any evidence of unit testing in more than half of the projects. Unit testing benefits most parts of the software's life cycle, such as designing, coding, debugging, and optimization (Hamill, 2004). It can reveal the bugs at an earlier stage of the development process, and the absence of unit testing may cause problems for *correctness & verifiability*.

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

We identified five projects using CI/CD tools, which are *3D Slicer*, *ImageJ*, *Fiji*, *dwv*, and *OHIF Viewer*.

In this section, the information about CI/CD tools is part of the answers to RQ2, and the information about software testing and documentation is part of the answers to RQ3.

4.3. Surface Reliability

As described in Section 4.1, we could not build *dwv* and *GATE*. However, since there was an online or VM version of them, successful deployment is possible. So the failure of installation did not affect their scores in *surface reliability*. Figure 5 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 *surface robustness* (Section 4.4), we found out that *Drishti* crashed during loading damaged image files and did not show any descriptive error message. On the other hand, we did not see any problems with the online version of *dwv*.

4.4. Surface Robustness

Figure 6 presents the scores for *surface robustness*.

The packages with higher scores elegantly handled the unexpected/unanticipated inputs, typically showing a clear error message. We might underestimate the score of *OHIF Viewer* since we needed further customization to load data, and the test was not complete.

Digital Imaging and Communications in Medicine (DICOM) is a widely used MI standard, and "it 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. We prepared two types of image files: the ones in correct formats and the broken ones. We used two MI sample files in the DICOM format as the image files with valid formats; we created a standard text file, changed its extension name from ".txt" to ".dcm", and used it as the unexpected/unanticipated input.

Being tested with the input files with correct formats, all software packages except *GATE* loaded the images correctly. *GATE* failed this test for unknown reasons.

With the unexpected/unanticipated input, *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 with noise pixels. *Drishti* successfully detected the broken format of the file, but the software crashed as a result. We recorded *Drishti*'s issue to the measurement of its *reliability* in Section 4.1.

4.5. Surface Usability

Figure 7 lists the AHP scores for *surface usability*.

We found a getting started tutorial for only 11 projects but a user manual for 22 projects. *MRIcroGL* was the only one with documentation for the expected user characteristics.

The software with higher scores usually provided both comprehensive document guidance and a good user experience. *INVESALIUS 3* set an excellent example of a detailed and precise user manual. *GATE* also provided a large number of documents, but we think that they conveyed the ideas poorly, as we had trouble understanding and using them.

Table 3 shows the user support models by the number of projects using them. This table contains part of the answers to RQ2. Maybe not every team intended to use GitHub issues to answer users' questions, but many users use them to seek help.

4.6. Maintainability

Figure 8 shows the AHP results for *maintainability*.

We marked *3D Slicer* with a much higher score than others because it did very well at closing the identified issues, and more importantly, we found it to have the most comprehensive artifacts. For example, as far as we could find out, only a few of the 29 projects had a project plan, developer's manual, or API documentation, and only *3D Slicer*, *ImageJ*, *Fiji* included all three documents. Meanwhile, *3D Slicer* has a much higher percentage of closed issues (91.65%) than *ImageJ* (52.49%) and *Fiji* (63.79%). Table 4 shows which projects had these documents, in the descending order of their *maintainability* scores. This table contains part of the answers to RQ1 and RQ3.

27 of the 29 projects used git as the version control tool; *AMIDE* used Mercurial; *Gwyddion* used Subversion. 24 projects used GitHub for their repositories; *XMedCon*, *AMIDE*, and *Gwyddion* used SourceForge; *DicomBrowser* and *3DimViewer* used BitBucket. The information about version control tools is part of the answers to RQ2.

4.7. Reusability

Figure 4.7 shows the AHP results for *reusability*.

As described in Section 3.3.1, we gave higher scores to the projects with an API document. As shown in Table 4, seven projects had API documents. We also considered projects with more code files and less LOC per code file as more reusable. Table 5 shows the number of text-based files by projects, which we used to approximate the number of code files. The table also lists the total number of lines (including comments and blanks), LOC, and average LOC per file. We arranged the items in the descending order of their *reusability* scores.

4.8. Surface Understandability

Figure 10 shows the scores for *surface understandability*.

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

4.9. Visibility/Transparency

Figure 11 shows the AHP scores for *visibility/transparency*. Generally speaking, the teams that actively documented their development process and plans scored higher because they delivered better communication to people outside the team.

Table 6 shows the projects which had documents for the development process, project status, development environment, and release notes, in the descending order of their *visibility/transparency* scores. This table contains part of the answers to RQ1 and RQ3.

4.10. Overall Scores

As described in Section 2.3, for our AHP measurements, there are nine criteria which are the nine software qualities and 29 software packages as the alternatives. In the absence of requirements for an actual project, we made all nine qualities equally important, so the score of each quality affects the overall scores on the same scale.

Figure 12 shows the overall scores of all 29 software packages in descending order. Since we produced the scores from the AHP process, the total sum of the 29 scores is precisely 1.

The top three software products *3D Slicer*, *ImageJ*, and *OHIF Viewer* had higher scores in most criteria. *3D Slicer* ranked in the top two software products for all qualities except *surface robustness*; *ImageJ* ranked in the top three products for *correctness & verifiability*, *surface reliability*, *surface usability*, *maintainability*, *surface understandability*, and *visibility/transparency*; *OHIF Viewer* ranked in the top five products for *correctness & verifiability*, *surface reliability*, *surface usability*, *maintainability*, and *reusability*. We might underestimate its scores of qualities *surface reliability* and *surface robustness* for *DICOM Viewer*, but equally compared it with the other software for the rest seven qualities.

5. Interviews with Developers

The measurement results in Section 4 are based on the information collected by ourselves. Such information is sufficient to measure the projects with reasonable efforts, but incomplete for us to understand the development process more deeply. For example, we usually cannot identify the following information in a project's documents: the pain points during the development, the threats to certain software qualities, and the developers' strategies to address them. We believe interviews with developers can collect the additional information we need. As a result, our method involves interviews with developers in a domain (Section 3.4). We applied this step to the MI domain (Section 3.5.4).

In this section, we summarize some answers from the interviews. We highlight the answers that are the most informative and interesting in this section, and summarize the rest in Appendix **??**.

As mentioned in Section 3.5.4, we contacted all 29 teams. Some of them responded and participated in the interviews. Eventually, we interviewed nine developers from eight of the 29 MI software projects. The eight projects are *3D Slicer*, *INVESALIUS 3*, *dwv*, *BioImage Suite Web*, *ITK-SNAP*, *MRIcroGL*, *Weasis*, and *OHIF*. We spent about 90 minutes for each interview and asked 20 prepared questions. We also asked following-up questions when we thought it was worth diving deeper. One participant was too busy to have an interview, so they wrote down their answers. The interviewees may have provided multiple answers to each question. Thus, when counting the number of answers, the total result is sometimes larger than nine.

5.1. Current and Past Pain Points

By asking questions 9, 10, and 12, we tried to identify the pain points during the development process in the eight projects. The pain points include current and past obstacles. We also asked the interviewees how they would solve the problems. This section contains the answers to RQ4. Questions 9, 10, and 12 are as follows:

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

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

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

Table 7 shows the number of times the interviewees mentioned the current and past obstacles in their projects.

The interviewees provided some potential and proven solutions for the problems in Table 7. We group these pain points into three major categories of obstacles: *resource*, *balance*, and *testing*. We put the less mentioned ones into the category *Others*. Sections 5.1.1, 5.1.2, and 5.1.3 include further discussion about the three major groups of pain points and their solutions.

5.1.1. Resource Pain Points

We summarize the pain points in the *resource* category as **the lack of fundings and time.**

The potential and proven solutions are:
Potential solutions from interviewees:

- when the team does not have enough developers for building new features and fixing bugs at the same time, shifting from development mode toward maintenance mode;
- licensing the software to commercial companies that integrate it into their products;
- better documentation to save time for answering users' and developers' questions;
- supporting third-party plugins and extensions.

Proven solutions from interviewees:

- GitHub Actions, which is a good CI/CD tool to save time.

Many interviewees thought lack of fundings and lack of time were the most significant obstacles. The interviewees from *3D Slicer* team and *OHIF* team pointed out that it was more challenging to get fundings for software maintenance as opposed to research. The interviewee from the *ITK-SNAP* team thought more fundings was a way to solve the lack of time problem, because they could hire more dedicated developers. On the other hand, the interviewee from the *Weasis* team did not think that fundings could solve the same problem, since he still would need a lot of time to supervise the project.

No interviewee suggested any solution to bring extra funding to the project. However, they provided ideas to save time, such as better documentation, third-party plugins, and good CI/CD tools.

5.1.2. Balance Pain Points

We summarize the pain points in the *balance* category as **the difficulty to balance between four factors: cross-platform compatibility, convenience to development & maintenance, performance, and security.** They are also related to the choice between native application and web application.

The potential and proven solutions are:

Potential solutions from interviewees:

- web applications that use computing power from computers GPU;
- to better support lower-end computers, adopting a web-based approach with backend servers;
- to better support lower-end computers, using memory-mapped files to consume less computer memory;
- more funding;
- maintaining better documentations to ease the development & maintenance processes;

Proven solutions from interviewees:

- one interviewee saw the performance problem disappeared over the years when computers became more and more powerful.

Table 8 shows the teams' choices between native application and web application. In all the 29 teams on our list, most of them chose to develop native applications. For the eight teams we interviewed, three of them were building web applications, and the *MRIcroGL* team was considering web-based solutions. So we had a good chance to discuss the differences between the two choices with the interviewees.

The interviewees talked about the advantages and disadvantages of the two choices. We summarize the opinions from the interviewees in Table 9.

5.1.3. Testing Pain Point

The pain point in the *testing* category is **the lack of access to real-world datasets for testing.** The information about software testing in this section is part of the answers to RQ3.

The potential and proven solutions are:
Potential solutions from interviewees:

- using open datasets

Proven solutions from interviewees:

- asking the users to provide deidentified copies of medical images if they have problems loading the images;
- sending the beta versions of software to medical workers who can access the data and complete the tests;
- if (part of) the team belongs to a medical school or a hospital, using the datasets they can access;
- if the team has access to MRI scanners, self-building sample images for testing;
- if the team has connections with MI equipment manufacturers, asking for their help on data format problems;
- storing all images that cause special problems, and maintaining this special dataset over time.

No interviewee provided a perfect way to solve this problem. However, connections between the development team and medical professionals/institutions could ease the pain.

5.2. Documents in the Projects

We tried to understand the interviewees' opinions on documentation and the quality of documentations with questions 11 and 19. The information about documentation is part of the answers to RQ3.

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

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

Table 10 summarizes interviewees' opinions on documentation. Interviewees from each of the eight projects thought that documentation was important to their projects, and most of them said that it could save their time to answer questions from users and developers. Most of them saw the need to improve their documentation, and only three of them thought that their documentations conveyed information clearly enough.

Table 11 lists some of the documentation tools and methods mentioned by the interviewees. This table contains part of the answers to RQ2.

5.3. Contribution Management and Project Management

We tried to understand how the teams managed the contributions and their projects by asking the following questions:

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

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

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

Although some team may have a documented process for accepting new contributions, no one talked about it during the interview. However, most of them mentioned using GitHub and pull requests to manage contributions from the community. The interviewees generally gave very positive feedback on using GitHub. Some also said they had handled the project repository with some other tools, and eventually transferred to git and GitHub. Table 12 shows the number of times the interviewees mentioned the methods of receiving contributions. This table contains part of the answers to RQ2.

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

Table 13 shows the software development models by the numbers of interviewees with the answers. Only two interviewees confirmed their development models. The others did not think they used a specific model, but three of them suggested that their processes were similar to Waterfall or Agile.

Some interviewees mentioned the project management tools they used, which are in Table 14. This table contains part of the answers to RQ2. Generally speaking, the interviewees talked about two types:

- Trackers, including GitHub, issue trackers, bug trackers and Jira;
- Documents, including GitHub, Wiki page, Google Doc, and Confluence.

No interviewee introduced any strictly defined project management process. The most common way was following the issues, such as bugs and feature requests. Additionally, the *3D Slicer* team had weekly meetings to discuss the goals for the project; the *INVESALIUS 3* team relied on the GitHub process for their project management; the *ITK-SNAP* team had a fixed six-month release pace; only the interviewee from the *OHIF* team mentioned that the team has a project manager; the *3D Slicer* team and *BioImage Suite Web* team were doing nightly builds and tests.

Most interviewees skipped the second part of Q14 "Do you think improving this process can tackle the current problem?". In retrospect, we should not have asked a yes-or-no question, since it is not very informative. The interviewee from the *OHIF* team gave a positive answer to this question. They believed that a better project management process can improve the efficiency of junior developers. They also improved the project management tools (from public Jira to public GitHub repository plus private Jira), so they could better communicate externally and internally.

The information about contribution management and project management in this section is part of the answers to RQ3.

5.4. Discussions on Software Qualities

Questions 15–18, and 20 are about the software qualities of *correctness*, *maintainability*, *understandability*, *usability*, and *reproducibility*, respectively. We asked these questions to understand the threats to these qualities and the developers' strategies to improve them. We discuss each quality in a separate section below. This section contains part of the answers to RQ5.

5.4.1. Correctness

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

Table 15 shows the threats to *correctness* by the numbers of interviewees with the answers.

Table 16 shows the strategies to ensure *correctness* by the numbers of interviewees with the answers. The interviewees from the *3D Slicer* and *ITK-SNAP* teams thought that the self-tests and automated tests were beneficial and could significantly save time. The interviewee from the *Weasis* team kept collecting medical images for more than ten years. These images have caused problems with the software. So he had samples to test specific problems.

The information about software testing in this section is part of the answers to RQ3.

5.4.2. Maintainability

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

Table 17 shows the strategies to ensure *maintainability* by the numbers of interviewees with the answers. The modular approach is the most talked-about solution to improve *maintainability*. The *3D Slicer* team used a well-defined structure for the software, which they named as "event-driven MVC pattern". Moreover, *3D Slicer* discovers and loads necessary modules at runtime, according to the configuration and installed extensions. The *BioImage*

Suite Web team had designed and re-designed their software multiple times in the last 10+ years. They found that their modular approach effectively supported the maintainability (Joshi et al., 2011).

5.4.3. Understandability

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

Table 18 shows the threats to *understandability* by the numbers of interviewees with the answers. It separates *understandability* issues to users and developers by the horizontal dash line.

Table 19 shows the strategies to ensure *understandability* by the numbers of interviewees with the answers.

5.4.4. Usability

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

Table 20 shows the strategies to ensure *usability* by the numbers of interviewees with the answers. The information about software testing in this table is part of the answers to RQ3.

5.4.5. Reproducibility

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

Table 21 shows the threats to *reproducibility* by the numbers of interviewees with the answers.

Table 22 shows the strategies to ensure *reproducibility* by the numbers of interviewees with the answers. The interviewee from the *3D Slicer* team provided various suggestions. One interviewee from another team suggested that they used *3D Slicer* as the benchmark to test their *reproducibility*.

The information about software testing in this section is part of the answers to RQ3.

6. Answers to Research Questions

This section answers our research questions from Section **??**. Sections 6.1– 6.6 summarize the answers to the six questions, respectively. Section 6.7 lists the threats to the validity of our research. The answers are based on our quality measurements in Section 4 and developer interviews in Section 5. We refer to these sections to avoid repetition, and organize the references in tables (e.g. Table 23).

6.1. Artifacts in the Projects

RQ1. What artifacts are present in current software projects?

We answer this question by examining the documentation, scanning the source code, and interviewing the developers of the projects. Table 23 shows the sections and tables containing answers to this research question.

As mentioned in Section 3.3.1, we search for all the artifacts in a project when measuring *maintainability*. The detailed records of the existing artifacts in the 29 MI projects are at https://data.mendeley.com/datasets/k3pcdvdzj2/1. Table 24 summarizes the frequency of the artifacts. This table also contains part of the answers to RQ3.

We summarized the definitions of the artifacts in https://github.com/.../Artifacts_MiningV3.xlsx Source code is a type of artifact. Since we only included OSS on the final list (Section 3.5.2), every project's source code was available. Thus, we excluded it in the above table.

6.2. Tools in the Projects

RQ2. What tools are used in the development of current software packages?

We answer this question by measuring the qualities and interviewing the developers. This section summarizes the tools used for CI/CD, user support, version control, documentation, contribution management, and project management. Table 25 shows the sections and tables containing answers to this research question.

As mentioned in Section 4.2, we identified five projects using CI/CD tools. *3D Slicer* and *OHIF Viewer* used CircleCI; *ImageJ*, *Fiji*, and *dwv* used Travis. We identified the above projects and tools by examining the documentation and source code of all projects. Thus, we may missed some projects or tools. According to the interviews with developers, *dwv* and *Weasis* used GitHub Actions.

6.3. Principles, Processes, and Methodologies in the Projects

RQ3. What principles, processes, and methodologies are used in the development of current software packages?

We answer this question by measuring the qualities and interviewing the developers. This section shows the principles, processes, and methodologies in software testing, documentation, contribution management, project management, and improving software qualities. Table 26 shows the sections and tables containing answers to this research question.

We identified the use of unit testing in less than half of the 29 projects. On the other hand, the interviewees believed that testing (including usability tests with users) was the top solution to improve *correctness*, *usability*, and *reproducibility*. One pain point in the development process is the lack of access to real-world datasets for testing. The developers' strategies to address it is in Section 5.1.3. One threat to *correctness* is: with huge datasets for testing, the tests are expensive and time-consuming. Three interviewees endorsed self tests / automated tests, which may save time for testing.

All 29 projects did not have theory manuals. We identified a road map in the *3D Slicer* project, and no requirements specifications for the rest. Eight of the nine interviewees thought that documentation was essential to their projects. However, they hold the common opinion that their documentation needed improvements. Nearly half of them also believed that the lack of time prevented them from improving the documentation.

6.4. Pain Points and Solutions

RQ4. What are the pain points for developers working on research software projects? What aspects of the existing processes, methodologies, and tools do they consider as potentially needing improvement? What changes to processes, methodologies, and tools can improve software development and software quality?

We answer this question by interviewing the developers. The answers to this question, including the pain points and the solutions proposed by the developers, are in Section 5.1.

6.5. Software Qualities

RQ5. What is the current status of the following software qualities for the projects? What actions have the developers taken to address them?

This section includes our answers from the qualities measurements and interviews with the developers. Table 27 shows the sections and tables containing answers to this research question.

6.6. Our Ranking versus the Community Ratings

RQ6. How does software designated as high quality by this methodology compare with top-rated software by the community?

We answer this question by grading the qualities of the software, collecting ratings from the MI community, then conducting two comparisons:

- comparing our ranking with the community ratings on GitHub, such as GitHub stars, number of forks, and number of people watching the projects (Section 6.6.1);
- comparing top-rated software designated by our methodology with the ones recommended by our domain experts (Section 6.6.2).

6.6.1. Our Ranking versus the GitHub Popularity

Table 28 shows our ranking to the 29 MI projects, and their GitHub metrics if applicable. As mentioned in Section 4.6, 24 projects used GitHub. Since GitHub repositories have different creation dates, we collect the number of months each stayed on GitHub, and calculate the average number of new stars, people watching, and forks per 12 months. The method of getting the creation date is described in Section 3.3.2, and we obtained these metrics in July, 2021.

In Table 28, we used the average number of new stars per 12 months as an indicator of the GitHub popularity, and listed the items in the descending order of this number. We ordered the non-GitHub items by our ranking. 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 are the same.

Project *dwv* was popular on GitHub, but we ranked it low. 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*. We weighted all qualities equally, which is not likely to be the case with all users. As a result, some projects with high popularity may not perform well in all qualities.

6.6.2. Designated Top Software versus the Domain Experts' Recommendation

As shown in Section 3.5.2, our domain experts recommended a list of top software with 12 software products. Table 29 and 30 compare the top 12 software projects ranked by our methodology with the ones from the domain experts.

All of the top 4 software from the domain experts are among the top 12 ones ranked by our methodology. 3 of the top 4 on both lists are the same ones: *3D Slicer*, *ImageJ*, and *Fiji*. *3D Slicer* is also the top 1 of both rankings.

As mentioned in Section 3.5.2, six of the recommended software packages did not have visualization as the primary function, so we did not include them on our final list.

* *MRIcron* development had moved to *MRIcroGL*, as mentioned in Section 3.5.2. Thus, we measured and ranked *MRIcroGL* at 18.

** We included *Mango* in the initial list, but removed it because it was not OSS. *Papaya* is a the web version OSS of *Mango*. We measured and ranked *Papaya* at 17.

6.7. Threats to Validity

This section lists all the potential threats to validity in our research. The definitions of common validity are (Ampatzoglou et al., 2019)(Zhou et al., 2016):

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

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

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

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

We categorize and present the threats to validity in the following subsections.

6.7.1. Threats to Construct Validity

- We compared nine software qualities for 29 software packages, so we could only spend a limited time on each of them. As a result, our assessments may have missed something relevant.

- Our ranking is partly based on surface (shallow) measurement, which may not fully reveal the underlying qualities.

6.7.2. Threats to Internal Validity

- It was not practical to ask each development team for every piece of information. We collected much information - such as artifacts and funding situations of software - by ourselves. There may be cases that we missed some information.

- As mentioned in Section 5, one interviewee was too busy to participate in a full interview, so he provided a version of written answers to us. Since we did not have the chance to explain our questions or ask him follow-up questions, there is a possibility of misinterpretation of the questions or answers.

- As mentioned in Section 4.1, we could not install or build *dwv*, *GATE*, and *DICOM Viewer*. We used a deployed online version for *dwv*, a VM version for *GATE*, but no alternative for *DICOM Viewer*. We might underestimate their rank due to uncommon technical issues.

6.7.3. Threats to External Validity

- We interviewed eight teams, which is a good proportion of the 29. However, there is still a risk that they might not well represent the whole MI software community.

- Our ranking gave all qualities equal weight, which may not be the case with all users. Thus, it may not represent the popularity of software among users.

- The number of GitHub stars, watches, and forks are not perfect measures of popularity, but they are what we had available.

6.7.4. Threats to Conclusion Validity

- We used the grading template in Appendix **??** to guide our measurements. Our impressions of the software - such as user experience - were factors in deciding some scores. Thus, there is a risk that some scores may be subjective and biased.

7. Recommendations

This section presents our recommendations on SC software development. In general, our suggestions apply to all SC domains, unless we specifically mention that a particular guideline is only for MI software.

Section 7.1 discusses the actions that can potentially improve the ten software qualities. Sections 7.2, 7.3, and 7.4 are based on the primary pain points collected from the developers in the MI domain, but we believe scientists and developers are likely to face them in most SC domains. These sections contain our general suggestions tackling them.

7.1. Recommendations on Improving Software Qualities

Based on our quality measurements in Sections 4 and discussions with the developers in Sections 5.4, we collected many key points that may improve the software qualities. We list the primary ones by each quality as follows,

- **Installability** (Section 4.1)
  - clear instructions;
  - automated installer;
  - including all dependencies in the installer;

  - avoiding heavily depending on other commercial products (e.g. Matlab);
  - considering building a web application that needs no installation.

- **Correctness & Verifiability** (Section 4.2 and 5.4.1)
  - test-driven development with unit tests, integration tests, and nightly tests;
  - two stage development process with stable release & nightly builds;
  - CI/CD;
  - requirements specifications and theory manuals (Smith, 2016) (Smith and Lai, 2005).
  - static code analysis tools (e.g. Lint and SonarQube)

- **Reliability** (Section 4.3)
  - test-driven development with unit tests, integration tests, and nightly tests.
  - two stage development process with stable release & nightly builds;
  - descriptive error messages.

- **Robustness** (Section 4.4)
  - designing with exceptions and make the software failures graceful;
  - descriptive error messages.

- **Usability** (Section 4.5 and 5.4.4)
  - usability tests and interviews with end users;
  - adjusting according to users' feedbacks;
  - getting started tutorials;
  - user manuals;
  - professional UX designs;
  - active supports to users.

- **Maintainability** (Section 4.6 and 5.4.2)

  - using GitHub;
  - modular approach with the design principle proposed by Parnas: "system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change." (Parnas et al., 2000)
  - documentation for developers: project plan, developer's manual, and API documentation.

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

- **Understandability** (Section 4.8 and 5.4.3)
  - modular approach;
  - good coding style: consistent indentation and formatting style; consistent, distinctive, and meaningful code identifiers; keeping parameters in the same order for all functions; avoiding hard-coded constants (other than 0 and 1);
  - clear comments, indicating what is being done, not how;
  - description of used algorithms;
  - documentation of explicit requirements on coding standard;
  - communication between developers and users via GitHub issues, mailing lists, and forums.
  - graphical user interface.

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

- **Reproducibility** (Section 5.4.5)

  - test-driven development with unit tests, integration tests, and nightly tests.
  - open-source;
  - making data and documentation available;
  - using open-source libraries.

7.2. Recommendations on Dealing With Limited Resources

The limitation of resources has many faces. We regard the lack of fundings, time, and developers as representations of this problem.

We summarize our discussion with the MI software developers in Section 5.1.1 with the following recommendations,

- **Identify the root cause.** More fundings or developers may not solve the problem of lacking time. It is beneficial to identify the underlying obstacles to the team.

- **Maintain a good documentation.** Creating and updating documentation consumes time, but can save much more time in the long term. If the users and developers can find answers to their questions themselves, they are less likely to abuse the team's issue tracker.

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

- **Use test-driven development process.** Many people think writing test cases is less fun than building the functional code, but this is only true before we encounter the bugs. Identifying and fixing bugs can consume substantial resources. Setting up the test cases costs time, but generates more benefits in the long run.

- **Consider supporting third-party plugins or extensions.** Why not let users share the burden? No software product can deliver every user's needs, and the large quantity of features leads to more bugs and maintenance problems. So it may be a good idea to shift some development and maintenance responsibilities to the users. The users may also be happy about the extra flexibility.

- **Consider "hibernating" for a while.** When developers are not enough, the team can shift from development mode toward maintenance mode for some time. Stop building new features, and instead fix bugs and design problems from the past. If the development team can repay some of its technical debt, the software qualities may improve as a result.

- **Commercialization is not always toxic.** Licensing the software to commercial companies to use as internal modules of their products may bring financial supports to the team. Meanwhile, the project can stay open-source for the community.

7.3. Recommendations on Choosing A Tech Stack

A tech stack refers to a set of technologies used by a team to build software and manage the project. Section 5.1.2 lists the advantages and disadvantages between native and web applications. In this section, we give further suggestions on the choice of a tech stack to address the *compatibility*, *maintainability*, *performance*, and *security* of software.

- **Identify the priorities of the qualities.** It is hard to cover all aspects. Some teams achieve all four above qualities for their software, but it is not an easy task. Sections 5.1.2 contains more details about the difficulty of balancing between the four qualities. A team needs to prioritize its objectives according to its resource and experience.

- **Be open-minded about new technologies.** Web applications with only a frontend are known for worse *performance* than native applications. However, new technologies may ease this difference. For example, some JavaScript libraries can help the frontend harness the power of computer GPU and accelerate graphical computing. In addition, there are new frameworks helping developers with cross-platform *compatibility*. For example, the Flutter project enables support for web, mobile, and desktop OS with one codebase.

- **Use git and GitHub.** As mentioned in Sections 4.6, almost all of the 29 MI software projects used git, and the majority of them used GitHub. We found from the projects' websites and our interviews with developers that, some projects moved from other version control tools to git and GitHub. GitHub provides convenient repository and project management, and OSS projects receive more attention and contribution on GitHub.

- **Web applications can also deliver high performance.** Web applications with backend servers may perform even better than native applications. If a team needs to support lower-end computers, it is good to use back-end servers for heavy computing tasks.

- **Backend servers can have low costs.** It is worth exploring the serverless solutions from major cloud service providers. Serverless still uses a server, but the team is only charged when they use it. The solution is event-driven, and costs the team by the number of requests it processes. Thus, serverless can be very cost-effective for the less intensively used functions.

- **Web transmission may diminish security.** Transferring sensitive data online can be a problem for projects requiring high security. Regulations in some SC domains may forbid doing so. In this case, a web application with a backend may not be a good choice.

- **Maintain a good documentation.** No matter what tech stack a team uses, a well-maintained project plan, developer's manual, and API documentation always help team members to contribute more and make fewer mistakes.

7.4. Recommendations on Enriching the Testing Datasets

As described in Section 5.1, it was difficult for some software development teams in the MI domain to access real-world medical imaging datasets. This problem restricted their capability and flexibility to test their software. We believe software developers in other SC domains may also face similar issues.

Based on Section 5.1.3, we provide some suggestions as follows,

- **Build and maintain good connections to datasets.** A team can build connections with professionals working in the SC domain, who may have access to private datasets and perform tests for the team. Moreover, if a team has such professionals as internal members, the process can be even simpler.

- **Collect and maintain datasets over time.** A team may face all kinds of strange problems caused by various unique inputs over the years of development. It is worth collecting and maintaining this data, which can form a good dataset for testing.

- **Search for open data sources.** In general, there are many open datasets in different SC domains. Take MI as an example, there are Chest X-ray Datasets by National Institute of Health (https://nihcc.app.box.com/v/ChestXra NIHCC) (Wang et al., 2017), Cancer Imaging Archive (https://www.cancerimagingarchive.net (Prior et al., 2017), and MedPix by National Library of Medicine (https://medpix.nlm.nih.gov/home) (Smirniotopoulos, 2014). A team developing MI software should be able to find more open datasets according to their needs.

- **Create sample data for testing.** If a team can access tools creating sample data, they may also self-build datasets for testing. For example, an MI software development team can use an MRI scanner to create images of objects, animals, and volunteers. The team can build the images based on specific testing requirements.

- **Remove privacy from sensitive data.** For data with sensitive information, a team can ask the data owner to remove such information or add noise to protect privacy. One example is using deidentified copies of medical images for testing.

- **Establish community collaboration in the domain.** During our interviews with developers in the MI domain, we heard many stories of asking for supports from other professionals or equipment manufacturers. However, we believe that broader collaboration between development teams can address this problem better. Some datasets are too sensitive to share, but if the community has some kind of "group discussion", teams can better express their needs, and professionals can better offer voluntary support for testing. Ultimately, the community can establish a nonprofit organization as a third-party, which maintains large datasets, tests OSS in the domain, and protects privacy.

8. Conclusions

We analyzed the state of the practice for SC software in the MI domain. To better achieve our goals in Section 1, we proposed six research questions in Section **??**.

Our methods in Section 3 form a general process to evaluate domain-specific software, that we apply on specific SC domains. As mentioned in Section 3.5, following this process, we chose the MI domain, identified 48 SC software candidates in it, then selected 29 of them to our final list. Section 4 lists our measurements to nine software qualities for the 29 projects, and Section 5 contains our interviews with eight of the 29 teams, discussing their development process and five software qualities.

We answered our six research questions in Section 6. In addition, Section 7 presents our recommendations on SC software development.

8.1. Key Findings

With the measurement results in Section 4, we revealed some current status of SC software development and qualities in the MI domain. We ranked the 29 software projects in nine qualities based on the grading scores. *3D Slicer*, *ImageJ*, and *OHIF Viewer* are the top three software by their overall scores.

The interview results in Section 5 show some merits, drawbacks, and pain points within the development process. The three primary categories of pain points are:

- the lack of fundings and time;
- the difficulty to balance between four factors: cross-platform compatibility, convenience to development & maintenance, performance, and security;
- the lack of access to real-world datasets for testing.

We summarized the solutions from the developers to address these problems. We also collected the status of software testing, documentation, contribution management, and project management in the eight projects.

Our answers to the research questions (Section 6) are based on the above findings. We identified the existing artifacts, tools, principles, processes, and methodologies in the 29 projects. By comparisons in Section 6.6, we found out: 1) four of the top five software projects in our ranking were also among the top five ones receiving the most GitHub stars per year (Table 28); 2) three of the top four in our ranking were among the top four provided by the domain experts (Table 29).

Section 7 presents our recommendations on improving software qualities and easing pain points during development. Some highlighted ones are:

- adopting test-driven development with unit tests, integration tests, and nightly tests;
- maintaining good documentation (e.g., installation instructions, requirements specifications, theory manuals, getting started tutorials, user manuals, project plan, developer's manual, API documentation, requirements on coding standards, development process, project status, development environment, and release notes);
- using CI/CD;
- using git and GitHub;
- modular approach with the design principle proposed by Parnas (Parnas et al., 2000);
- considering newer technologies (e.g., web application and serverless solution);
- various ways of enriching the testing datasets in Section 7.4.

8.2. Future Works

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

- we can make the surface measurements less shallow. For example:
  - *surface reliability*: our current measurement relies on the processes of installation and getting started tutorials. However, not all software needs installation or has a getting started tutorial. We can design a list of operation steps, perform the same operations with each software, and record any errors.
  - *surface robustness*: we used damaged images as inputs for this measuring MI software. This process is similar to fuzz testing (Wikipedia contributors, 2021b), which is one type of fault injection (Wikipedia contributors, 2021a). We may adopt more fault injection methods, and identify tools and libraries to automate this process.

- *surface usability*: we can design usability tests and test all software projects with end-users. The end-users can be volunteers and domain experts.
- *surface understandability*: our current method does not require understanding the source code. As software engineers, perhaps we can select a small module of each project, read the source code and documentation, try to understand the logic, and score the ease of the process.

- we can further automate the measurements on the grading temple in Appendix **??**. For example, with automation scripts and the GitHub API, we may save significant time on retrieving the GitHub metrics. We have started to build a tool for this purpose, with its source code at this repository https://github.com/smiths/AIMSS/.../GitHubMetricsCollector. This GitHub Metrics Collector can take GitHub repository links as input, automatically collect metrics from the GitHub API, and record the results. We can improve and use this tool in our future projects;
- the grading standard can be more explicit. For example, we can explicitly define scores for each item in the grading temple.
- we can improve some interview questions. Some examples are:
  - in Q14, "Do you think improving this process can tackle the current problem?" is a yes-or-no question, which is not informative enough. As mentioned in Section 5.3, most interviewees ignored it. We can change it to "By improving this process, what current problems can be tackled?";
  - in Q16, we can ask more details about the modular approach, such as "What principles did you use to divide code into modules? Can you give an example of using the principles?";
  - Q17 and Q18 should respectively ask *understandability* to developers and *usability* to end-users.
- we can better organize the interview questions. Since we use audio conversion tools to transcribe the answers, we should make the transcription easier to read. For example, we can order them together for questions about the five software qualities and compose a similar structure for each.

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

In addition, we propose a few SC domains that are potentially suitable for future works:

- Metallurgy
- Quantitative Finance
- Computational Fluid Dynamics
- Basic Linear Algebra
- Finite Elements
- Sparse Linear Solvers

After applying our method on various domains, we can start a meta-study to compare the state of the practice for software in different domains.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 1991. *Fundamentals of software engineering*. Prentice Hall PTR.

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

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

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

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

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

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

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

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

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

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

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

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`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chris Rorden. 2021. MRIcron. https://github.com/neurolabusc/MRIcron. [Online; accessed 11-August-2021].

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

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

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

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

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

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

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

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

Spencer Smith, Zheng Zeng, and Jacques Carette. 2018c. 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 Lei Lai. 2005. A New Requirements Template for Scientific Computing. In *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, J. Ralyté, P. Ågerfalk, and N. Kraiem (Eds.). In conjunction with 13th IEEE International Requirements Engineering Conference, Paris, France, 107–121.

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

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

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

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

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`

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

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

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

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

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

Greg Wilson, Dhavide Aruliah, C. Titus Brown, Neil Chue Hong, Matt Davis, Richard Guy, Steven Haddock, Kathryn Huff, Ian Mitchell, Mark Plumbley, Ben Waugh, Ethan White, and Paul Wilson. 2014. Best Practices for Scientific Computing. *PLoS Biology* 12 (01 2014), e1001745. https://doi.org/10.1371/journal.pbio.1001745

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

Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. 2006. User-Guided 3D Active Contour Segmentation of Anatomical Structures: Significantly Improved Efficiency and Reliability. *Neuroimage* 31, 3 (2006), 1116–1128.

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

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

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

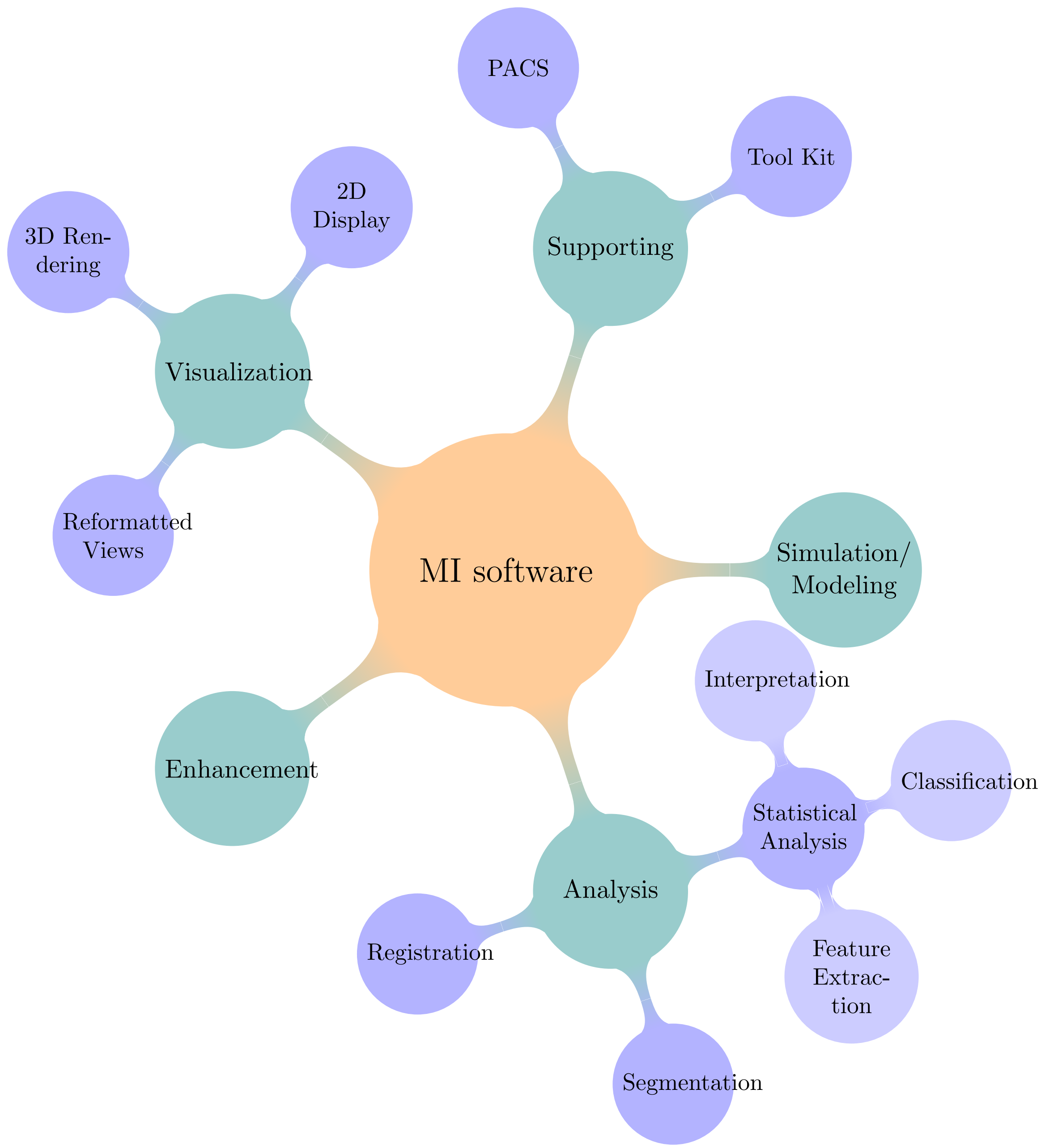

Figure 1: Major functions of MI software

Software name?	**(string)**	**3D Slicer**	**Ginkgo CADx**	**XMedCon**
URL?	**(URL)**	https://www.slicer.org/	http://ginkgo-cadx.com/en/	https://xmedcon.sourceforge.io/
Affiliation (institution(s))	**(string or {N/A})**	National Alliance for Medical Image Computing Neuroimage Analysis Center Surgical Planning Laboratory, Brigham and Women's Hospital National Center For Image Guided Therapy	Sacyl Public healthcare system of Castilla y León GNUmed team	n/a
Software purpose	**(string)**	An open source software platform for medical image informatics, image processing, and three-dimensional visualization.	An advanced DICOM viewer and dicomizer (converts png, jpeg, bmp, pdf, tiff to DICOM).	An open source toolkit for medical image conversion
Number of developers (all developers that have contributed at least one commit to the project) (use repo commit logs)	**(number)**	100	3	2
Popularity Measure	**({stars: number, forks: number, other*: number}), * explained via a string**	stars: 379, forks:171, watching: 25	stars: 102, forks:31, watching: 24	stars: n/a, forks:n/a, watching: n/a
Initial release date?	**(date)**	1998	2010	2000
Last commit date?	**(date)**	02-08-2020	21-05-2019	03-08-2020
Status? (alive is defined as presence of commits in the last 18 months)	**({alive, dead, unclear})**	alive	alive	alive
License?	**({GNU GPL, BSD, MIT, terms of use, trial, none, unclear, other*}) * given via a string**	BSD	GNU LGPL	GNU LGPL
Platforms?	**(set of {Windows, Linux, OS X, Android, other*}) * given via string**	Windows, Linux, OS X	Windows, Linux, OS X	Windows, Linux, OS X

Figure 2: Grading template example

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

Table 2: Final software list

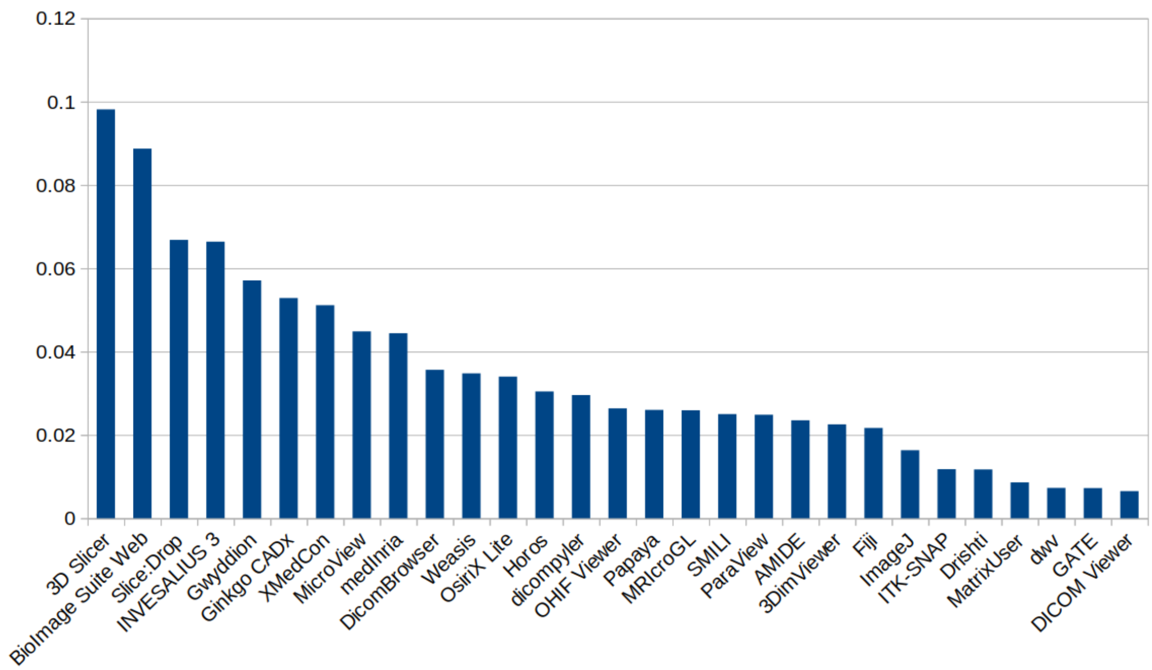

Figure 3: AHP installability scores

User support model	Number of 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 3: User support models by number of projects

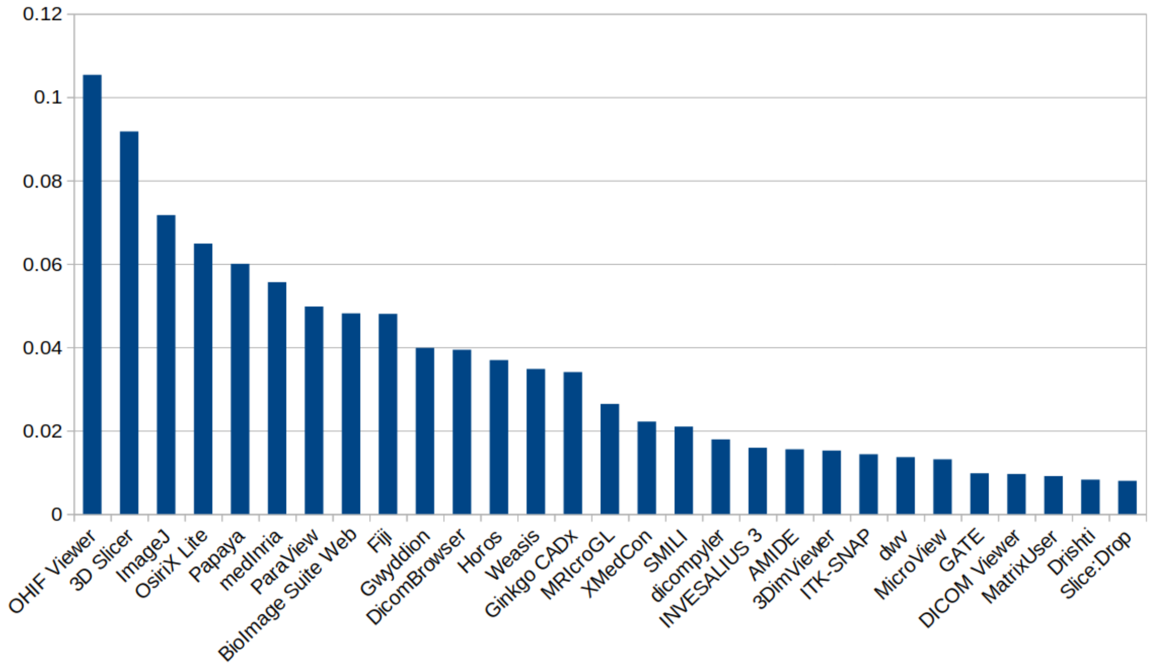

Figure 4: AHP correctness & verifiability scores

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

Table 4: Software with the maintainability documents

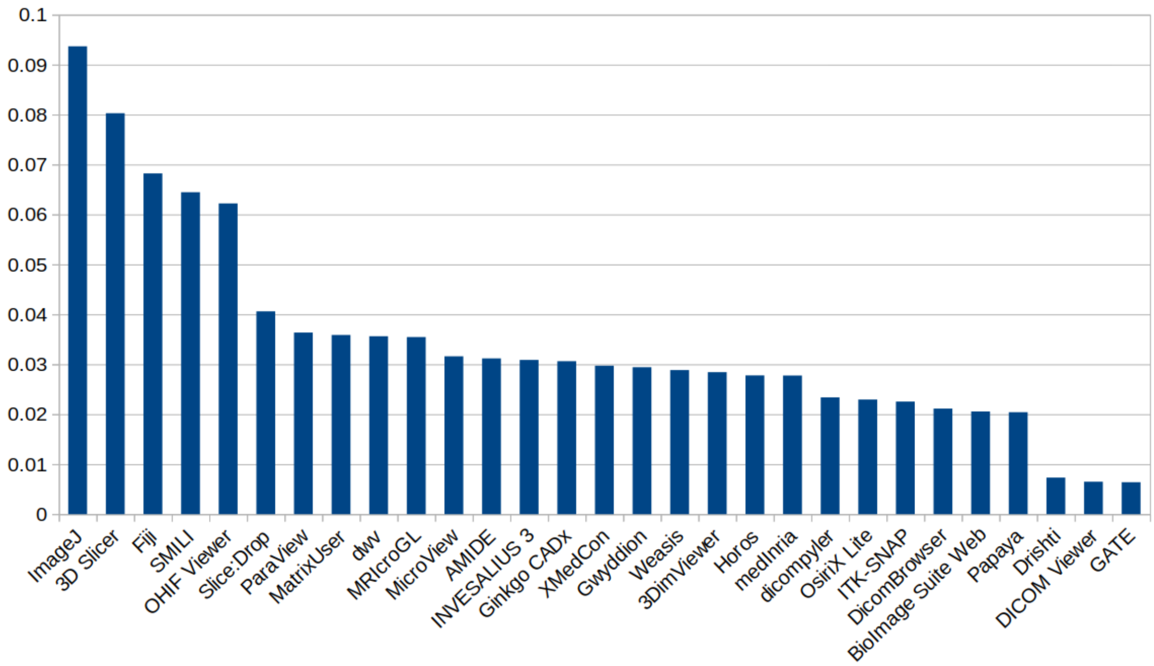

Figure 5: AHP surface reliability scores

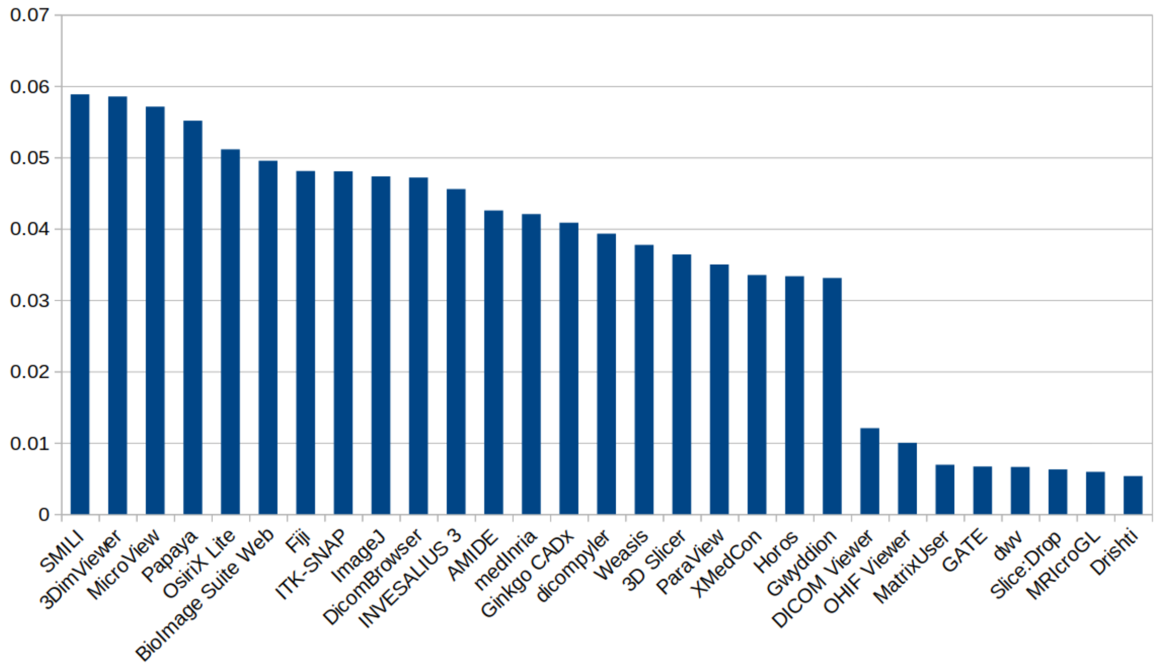

Figure 6: AHP surface robustness scores

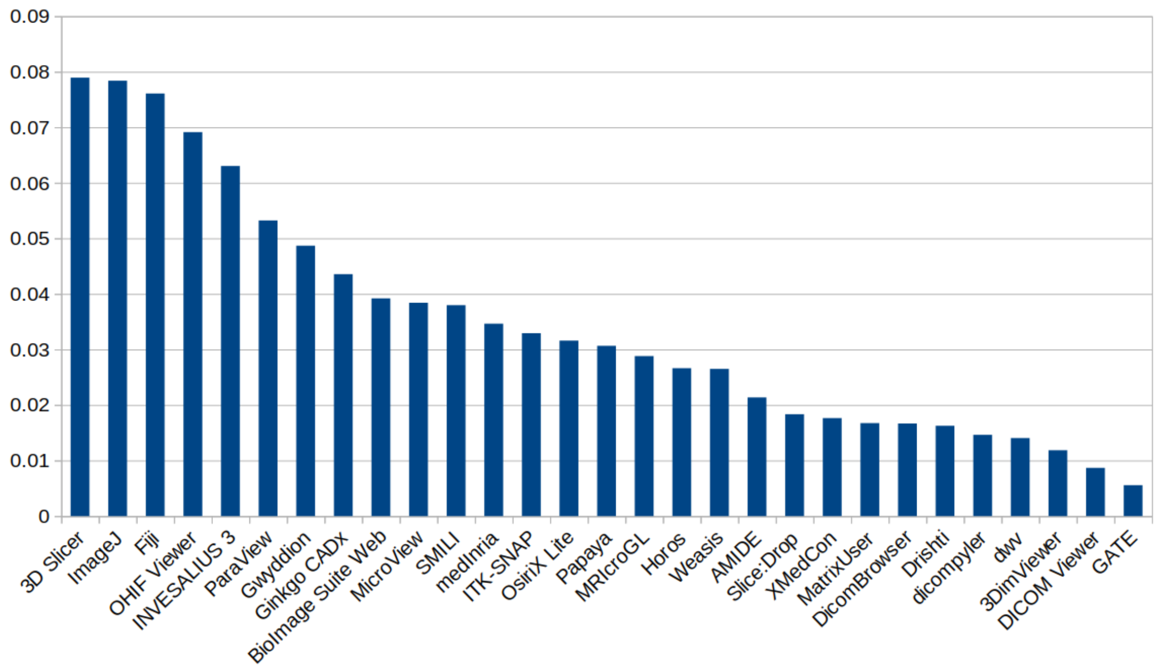

Figure 7: AHP surface usability scores

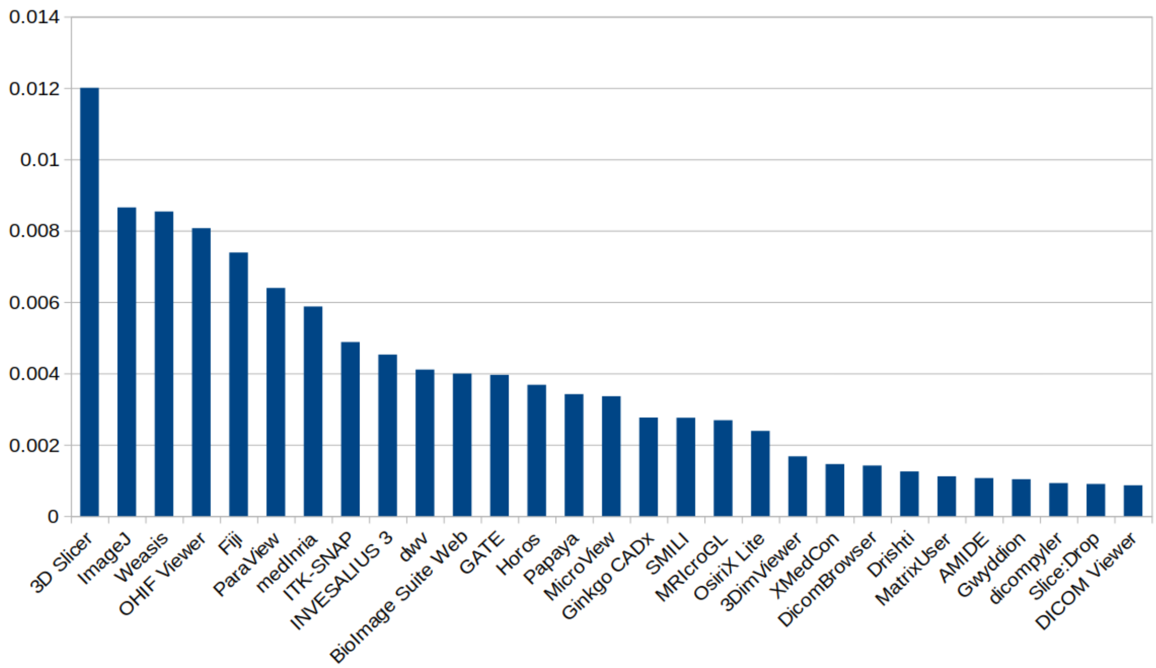

Figure 8: AHP maintainability scores

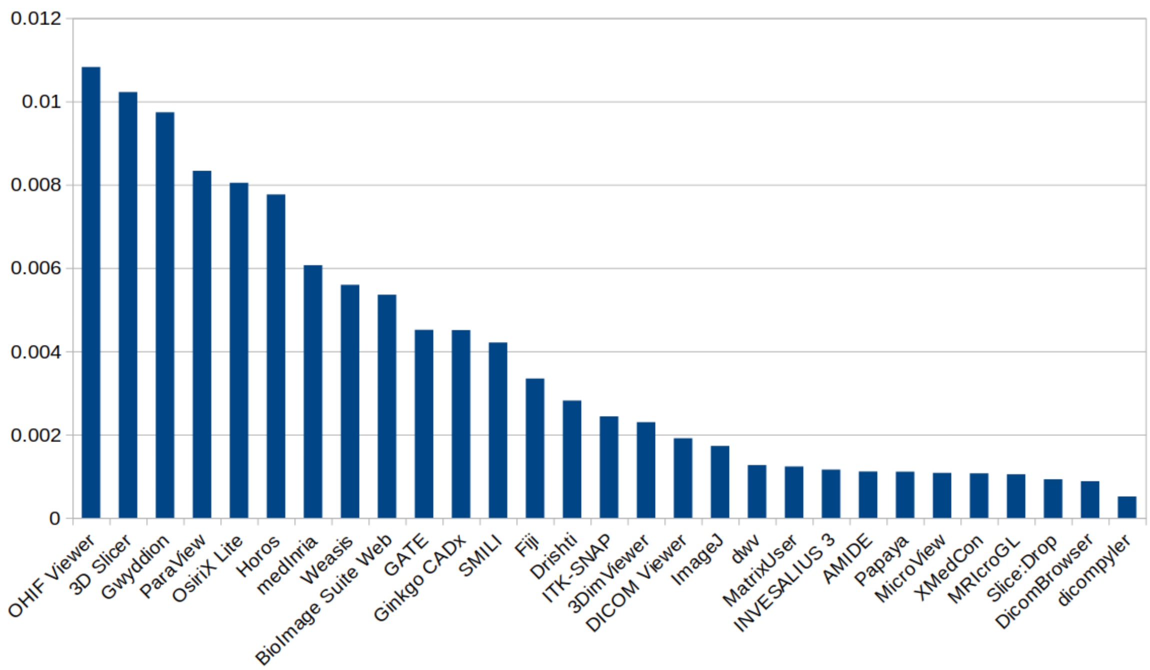

Figure 9: AHP reusability scores

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 5: Number of files and lines

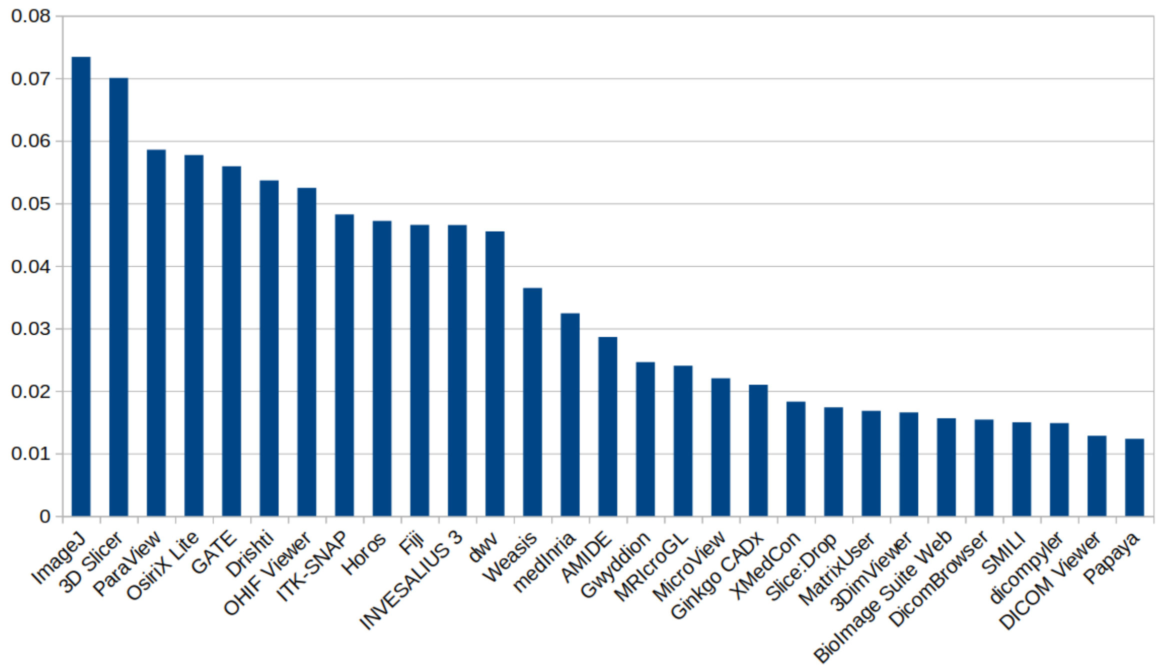

Figure 10: AHP surface understandability scores

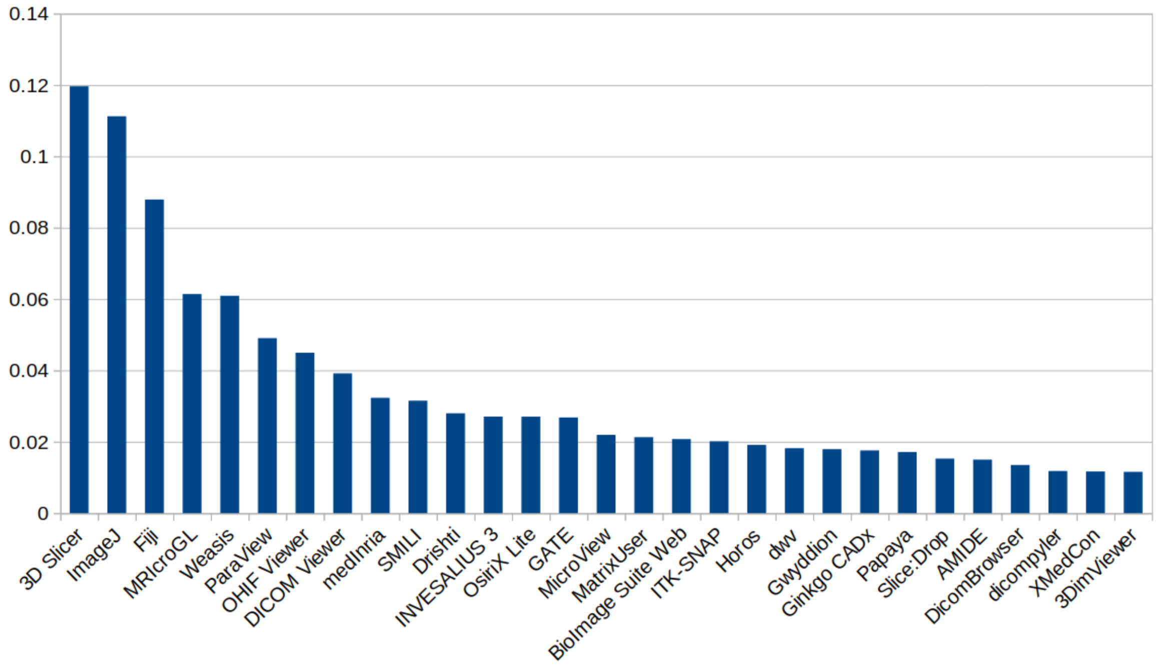

Figure 11: AHP visibility/transparency scores

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

Table 6: Software with the visibility/transparency documents

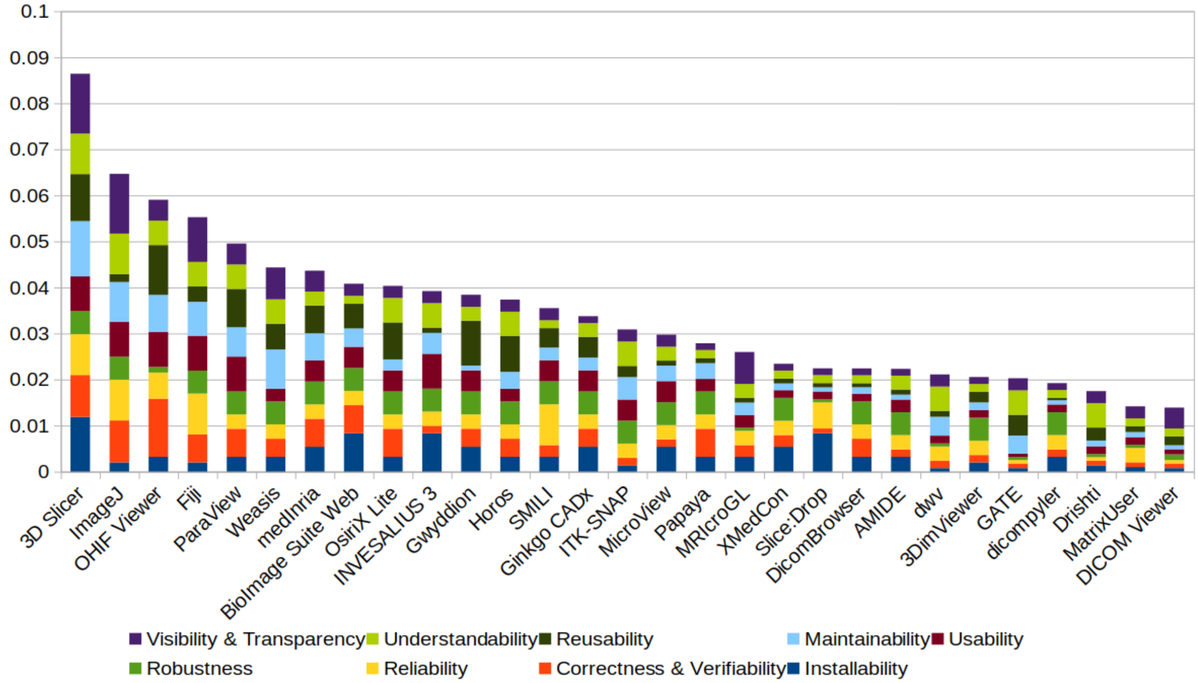

Figure 12: Overall AHP scores for all 9 software qualities

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

Table 7: Current and past obstacles by the numbers of interviewees with the answers

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

Table 8: Teams' choices between native application and web application

	Native application	Web application
Ad	- higher performance	- easy to achieve cross-platform compatibility - simpler build process
Disad	- hard to achieve cross-platform compatibility - more complicated build process	*Without a backend:* - lower performance *With a backend:* - harder for privacy protection - extra cost for backend servers

Table 9: Advantages and disadvantages of native application and web application

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

Table 10: Opinions on documentation by the numbers of interviewees with the answers

Tool or method for documentation	Num ans.
Forum discussions	3
Videos	3
GitHub	2
Mediawiki / wiki pages	2
Workshops	2
Social media	2
Writing books	1
Google Form	1
State management	1

Table 11: Tools and methods for documentation by the numbers of interviewees with the answers

Method of receiving contributions	Num ans.	
	current	past
GitHub with pull requests	8	
Code contributions from emails		3
Code contributions from forums		1
Sharing the git repository by email		1

Table 12: Methods of receiving contributions by the numbers of interviewees with the answers

Software development model	Num ans.
Undefined/self-directed	3
Similar to Agile	2
Similar to Waterfall	1
Agile	1
Waterfall	1

Table 13: Software development models by the numbers of interviewees with the answers

Project management tools	Num ans.
GitHub	3
Issue trackers	1
Bug trackers	1
Jira	1
Wiki page	1
Google Doc	1
Confluence	1

Table 14: Project management tools by the numbers of interviewees with the answers

Threat to correctness	Num ans.
Complexity - data in various formats and complicated standards.	2
Complexity - different MI machines create data in different ways.	2
Complexity - additional functions besides viewing.	1
The lack of real word image data for testing.	1
The team cannot use private data for debugging even when the data cause problems.	1
With huge datasets for testing, the tests are expensive and time-consuming.	1
It is hard to manage releases.	1
The project has no unit tests.	1
The project has no dedicated quality assurance team.	1

Table 15: Threats to correctness by the numbers of interviewees with the answers

Strategy to ensure correctness	Num ans.
Test-driven development, component tests, integration tests, smoke tests, regression tests.	4
Self tests / automated tests.	3
Two stage development process / stable release & nightly builds.	3
CI/CD.	1
Using deidentified copies of medical images for debugging.	1
Sending beta versions to medical workers who can access the data to do the tests.	1
Collecting and maintaining a dataset of problematic images.	1

Table 16: Strategies to ensure correctness by the numbers of interviewees with the answers

Strategy to ensure maintainability	Num ans.
Modular approach / maintain repetitive functions as libraries.	5
Supporting third-party extensions.	1
Easy-to-understand architecture.	1
Dedicated architect.	1
Starting from simple solutions.	1
Documentation.	1

Table 17: Strategies to ensure maintainability by the numbers of interviewees with the answers

Threat to understandability	Num ans.
Not all users understand how to use some features.	2
The team has no dedicated user experience (UX) designer.	1
Some important indicators are not noticeable (e.g. a progress bar).	1
Not all users understand the purpose of the software.	1
Not all users know if the software includes certain features.	1
Not all users understand how to use the command line tool.	1
Not all users understand that the software is a web application.	1
Not all developers understand how to deploy the software.	1
The architecture is difficult for new developers to understand.	1

Table 18: Threats to understandability by the numbers of interviewees with the answers

Strategy to ensure understandability	Num ans.
Documentation / user manual / user mailing list / forum.	4
Graphical user interface.	2
Testing every release with active users.	1
Making simple things simple and complicated things possible.	1
Icons with more clear visual expressions.	1
Designing the software to be intuitive.	1
Having a UX designer with the right experience.	1
Dialog windows for important notifications.	1
Providing an example if the users need to build the software by themselves.	1

Table 19: Strategies to ensure understandability by the numbers of interviewees with the answers

Strategy to ensure usability	Num ans.
Usability tests and interviews with end users.	3
Adjusting according to users' feedbacks.	3
Straightforward and intuitively designed interface / professional UX designer.	2
Providing step-by-step processes, and showing the step numbers.	1
Making the basic functions easy to use without reading the documentation.	1
Focusing on limited number of functions.	1
Making the software more streamlined.	1
Downsampling images to consume less memory.	1
An option to load only part of the data to boost performance.	1

Table 20: Strategies to ensure usability by the numbers of interviewees with the answers

Threat to reproducibility	Num ans.
If the software is closed-source, the reproducibility is hard to achieve.	1
The project has no user interaction tests.	1
The project has no unit tests.	1
Using different versions of some common libraries may cause problems.	1
CPU variability can leads to non-reproducibility.	1
The team may misinterpret how manufacturers create medical images.	1

Table 21: Threats to reproducibility by the numbers of interviewees with the answers

Strategy to ensure reproducibility	Num ans.
Regression tests / unit tests / having good tests.	6
Making code, data, and documentation available / OSS / open-source libraries.	5
Running same tests on all platforms.	1
A dockerized version of the software, insulating it from the OS environment.	1
Using standard libraries.	1
Monitoring the upgrades of the libraries.	1
Clearly documenting the versions.	1
Bringing along the exact versions of all the dependencies with the software.	1
Providing checksums of the data.	1
Benchmarking the software against other software with similar purposes.	1

Table 22: Strategies to ensure reproducibility by the numbers of interviewees with the answers

Section or table	Description
Table 4	Maintainability documents
Table 6	Visibility/transparency documents

Table 23: Sections and tables with answers to RQ1

Artifact	Number of projects
README	29
Version Control	29
License	28
Bug tracker	28
Change request	28
User Manual	22
Release notes	22
Build file	18
Tutorials	18
Installation Guide	16
Test cases	15
Authors	14
FAQ	14
Acknowledgements	12
Executable files	10
Developer's Manual	8
API documentation	7
Troubleshooting guide	6
Project Plan	5

Table 24: Artifacts by their frequency in the 29 MI projects

Section or table	Description
Section 4.2	CI/CD tools
Table 3	User support tools
Section 4.6	Version control tools
Table 11	Documentation tools
Table 12	Contribution management tools
Table 14	Project management tools

Table 25: Sections and tables with answers to RQ2

Section or table	Description
Section 4, 5.1.3, 5.4.1, and 5.4.5; Table 20	Software testing
Section 4 and 5.2; Table 4, 6, and 24	Documentation
Section 5.3	Contribution management
Section 5.3	Project management

Table 26: Sections and tables with answers to RQ3

Section or table	Description
Section 4.1	Installability
Section 4.2 and 5.4.1	Correctness & verifiability
Section 4.3	Reliability
Section 4.4	Robustness
Section 4.5 and 5.4.4	Usability
Section 4.6 and 5.4.2	Maintainability
Section 4.7	Reusability
Section 4.8 and 5.4.3	Understandability
Section 4.9	Visibility/transparency
Section 5.4.5	Reproducibility

Table 27: Sections and tables with answers to RQ4

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

Table 28: Software ranking versus GitHub metrics

Our ranking	Assessed software	Domain experts
1	3D Slicer	1
2	ImageJ	3
3	OHIF Viewer	n/a
4	Fiji	4
5	ParaView	n/a
6	Weasis	n/a
7	medInria	n/a
8	BioImage Suite Web	n/a
9	OsiriX Lite	n/a
10	INVESALIUS 3	n/a
11	Gwyddion	n/a
12	Horos	2

Table 29: Top software by our ranking versus domain experts' recommendation

Our ranking	Recommended software	Domain experts
1	3D Slicer	1
12	Horos	2
2	ImageJ	3
4	Fiji	4
n/a	AFNI	5
n/a	FSL	6
n/a	Freesurfer	7
18*	Mricron	8
17**	Mango	9
n/a	Tarquin	10
n/a	Diffusion Toolkit	11
n/a	MRItrix	12

Table 30: Domain experts' recommendation versus our ranking