STATE OF PRACTICE FOR MEDICAL IMAGING SOFTWARE

ASSESSING THE CURRENT STATE OF THE PRACTICE FOR MEDICAL IMAGING SOFTWARE

By

AO DONG.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the degree

Master of Engineering in Computing and Software

McMaster University

MASTER OF ENGINEERING (2021) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Assessing the Current State of the Practice for Medical Imaging Software

AUTHOR: Ao Dong

SUPERVISOR: Dr. Spencer Smith and Dr. Jacques Carette

NUMBER OF PAGES: vii, 87

Abstract

Abstract here

Acknowledgments

acknowledgements here

Contents

Chapter 1

Introduction

This paper analyzes the state of the practice for Medical Imaging (MI) software. MI is the clinical tool to image the interior of a body, providing information for diagnostic, analytic, and medical applications [1] [74]. MI is an essential part of collecting accurate information during clinical diagnosis [79]. MI computing and software aim to visualize and process medical images and produce clinically meaningful information [73].

Scientific computing (SC) is the technique and process of mathematically modeling real-world engineering or scientific systems with computing tools, as well as analysis and prediction with the models [65]. MI software belongs to a specific domain of SC.

We aim to study the current status of SC software development in the MI domain; understand the current merits, drawbacks, and pain points during the development process, as well as the software qualities in the domain; provide guidelines and recommendations for future development.

Section 1.1 presents our motivation to start the research set the above goals, Section 1.2 lists our research questions, and Section 1.3 explains the domain analysis of MI software scope of our research.

1.1 Motivation

Most scientists think developing and using SC software play significant roles in their research [28]. They spend a substantial proportion of their working hours on SC software development [28] [54], and this proportion of time has increased over the years [28].

Developing SC software requires solid knowledge in specific domains [75]. Many of them learn software engineering skills by themselves or from their peers, instead of proper training [28]. Hannay et al. [28] also pointed out that many scientists showed ignorance and indifference to standard software engineering concepts. According to a survey by Prabhu et al. [54], more than half of the 114 subjects did not use any proper debugger for their software.

Due to its nature, SC software born from one project can be part of many other projects in the future, with the potential to disproportionately causing damages to scientific researches [75].

As a result, the development process and quality of SC software concern us. We want to understand their status in SC domains and improve them.

1.2 Research Questions

To achieve our objectives, we designed a few research questions and tried to answer them by our research methods. The questions are as follows,

1. What are the pain points for developers working on MI software projects? What are the solutions for these pain points?

2. What artifacts the projects generated?

3. What role does documentation play in the projects? What are the developers' attitudes toward it?

4. What principles, processes, methodologies, and tools the projects used?

5. 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

6. How does software quality ranking generated by our methods compare with the ratings from the community?

1.3 Scope

According to Bankman [8], MI software deals with six different basic problems, while Angenentet et al. [6] 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 a section covering some other functions [8]. On the other hand, Angenent's team included Simulation [6]. According to Wikipedia contributors [73], MI software has primary functions in categories such as Segmentation, Registration, Visualization (including the basic display, reformatted views, and 3D volume rendering), Statistical Analysis, Image-based Physiological Modelling, etc. As Kim et al. [39] describe, 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 [64] and ITK [46], developers build software with Visualization and Analysis functions; Picture Archiving and Communication System (PACS) helps users to economically store and conveniently access images [13].

We divided MI software into five sub-groups and several sub-sub-groups by their major functions shown in Figure 1.1.

In this project, the scope of the software is limited to the software library providing the Visualization tools and functions.

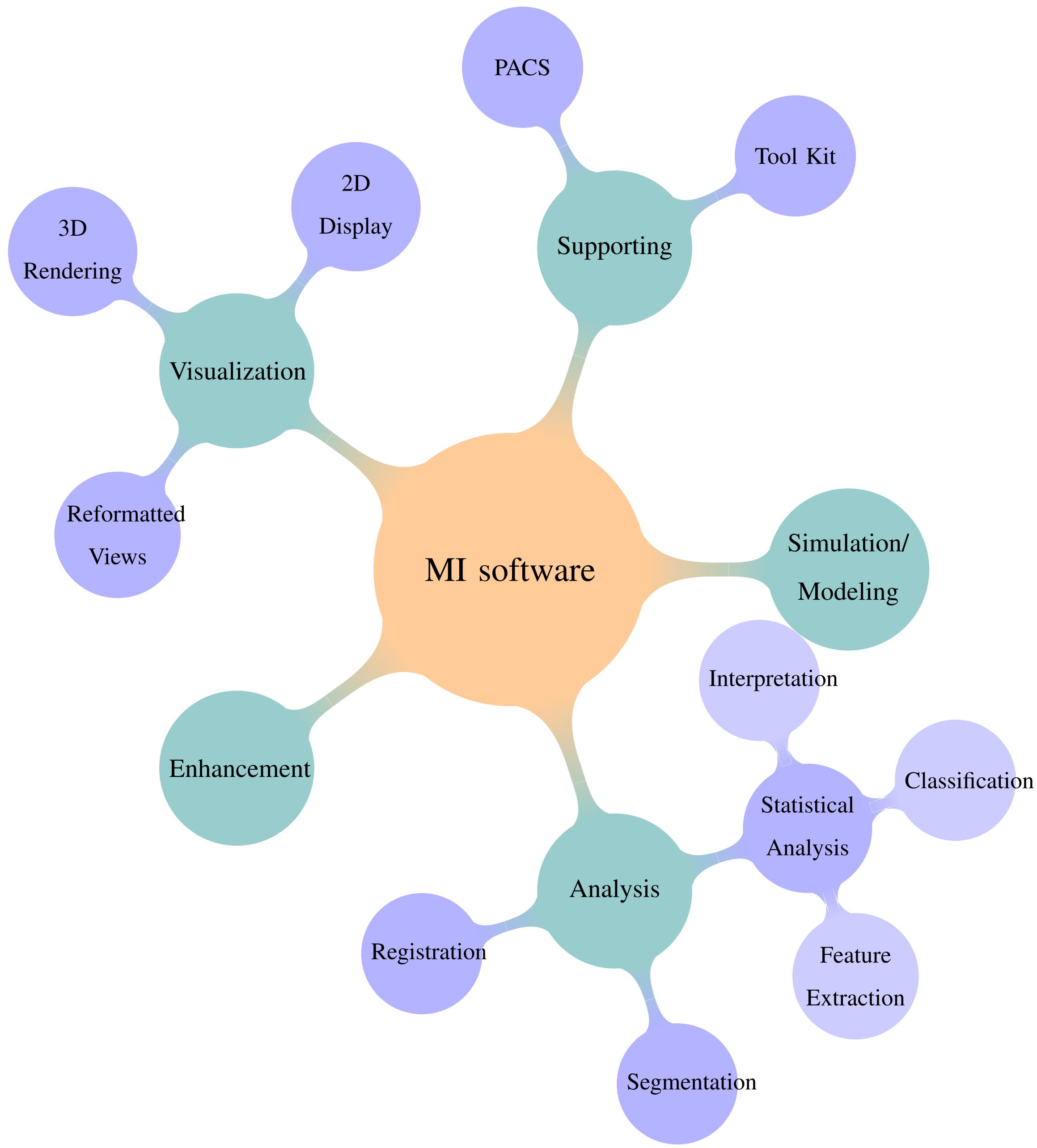

Figure 1.1: Major functions of MI software

Chapter 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 grade the software qualities with the grading template in Appendix A, we need the definitions of qualities and the Analytic Hierarchy Process (AHP).

2.1 Software Categories

We usually target specific software categories to narrow down the scopes when selecting software domains and software packages. In this section, we discuss three common software categories and also SC software.

2.1.1 Open Source Software

For Open Source software (OSS), its source code is openly accessible, and 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 [16]. 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 the 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 typically do not allow users to access or modify the source code of the software [55]. The term *freeware* should not be confused with *free software*, which is similar to OSS but with a few differences. 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 inner characteristics may 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” [23]. 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 [23]. 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 is software in Scientific Computing (SC), since the developers need to utilize their domain-specific during the development process [76].

2.1.4 Scientific Computing Software

Software development in SC depends on the knowledge of three areas - the inside of a specific engineering or science domain, the ability to mathematically build models and apply algorithms, and to implement theoretical models and algorithms with computational tools. SC software is built with mathematical and computational tools to serve the purpose of solving scientific problems in a domain [47]. However, the majority of scientists developing their software are self-taught programmers [76], so there may be a bigger Room for software quality improvement in SC domains.

2.2 Software Quality Definitions

Our grading template in Appendix A contain nine software qualities. This section lists the definitions of them, which are from Smith et al. [66]. The order of the qualities follows the grading template.

- **Installability** The effort required for the installation, uninstallation, or reinstallation of a software or product in a specified environment.

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

- **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).

- **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.

- **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.

- **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.

- **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.

- **Understandability** (To be completed)

- **Visibility/Transparency** The extent to which all of the steps of a software development process and the current status of it are conveyed clearly.

2.3 Analytic Hierarchy Process

To generate grading scores for a group of software packages, we use the AHP to pairwise compare them. Thomas L. Saaty developed this tool, and people widely used it to make and analyze multiple criteria decisions [69]. The AHP organizes multiple criteria factors in a hierarchical structure and pairwise compares the alternatives to calculate relative ratios [59].

For a project with m criteria, we can use a $m \times m$ matrix A to record the relative importance between factors. By pairwise compare criterion i and criterion j, the value of A_{ij} is decided as follows, and the value of A_{ji} is $1/A_{ij}$ [59],

- $A_{ij} = 1$ 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 vecotr w can be calculated by solving the following equation [59],

$$Aw = \lambda_{max} w, \tag{2.1}$$

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

In this project, w is approximated with the approach classic *mean of normalized values* [34],

$$w_i = \frac{1}{m} \sum_{j=1}^{m} \frac{A_{ij}}{\sum_{k=1}^{m} A_{kj}} \tag{2.2}$$

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 9 software qualities mentioned above are the criteria ($m = 9$), while 29 software packages ($n = 29$) are compared. The software are evaluated with the grading template in Appendix A and a subjective score is given for each quality. For a pair of qualities or software, i and j, such that i is not less significant than j, the pairwise comparison result of i versus j is converted from $min((score_i - score_j) + 1, 9)$.

Chapter 3

Methods

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

Our method involves: 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).

Details of how we applied the method on the MI domain are expanded upon in Section 3.5.

3.1 Domain Selection

Our methods are generic, but we have only applied them to scientific domains due to the objective of our research.

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 [67] after the screening step. For example, we remove the ones without recent updates or specific functions. Thus, we need enough software candidates from the beginning. Besides, we prefer OSS projects because our grading method works better on them. Moreover, we prefer domains with active communities developing and using SC software, so 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 needs [67].

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. Meanwhile, we include the suggested ones from domain experts [67].

3.2.2 Filter the Software List

The goal is to build a software list with a length of about 30 [67].

The only "mandatory" requirement is that the software must be OSS, as defined in Section 2.1.1. Because to evaluate all software qualities, our methods need to access the source code.

The other factors to filter the list are optional, and we consider them according to the number of software candidates and the objectives of the research project.

One factor is the functions and purposes of the software. For example, we can choose a group of software with similar functions, so that the comparison is between the software; or we can cross-compare sub-groups in the domain, then we need to select candidates from each sub-group.

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 [67].

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 [67].

With domain experts in the team, we value their opinions on the filtering process. For example, if a software package is not OSS and has no updates for a long while, but the domain experts identify it as a valuable product, we still consider it to keep.

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 A. The template contains 101 questions that we use for grading software products.

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, and 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 carry out on 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 that whether the software break during the installations and tutorials, whether there are descriptive error messages, and if we can recover the process after the errors.

- **Surface Robustness** We check that how the software handle unexpected/unanticipated input. For example, for software packages with the function to load image files, we prepare broken image files for them to open. 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 documents of the projects and consider software with a getting started tutorial and a user manual easier to use. Meanwhile, we check if users have any channels to get supports. Our impressions and user experiences when testing the software also affect the scores. For example, easy-to-use graphical user interfaces give us a better experience, thus lead 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 and the percentages of closed issues and 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 tend to have smaller parts in separated files, are easier to reuse. Thus, we consider the projects with more code files to be more reusable. We use *GitStats* as a tool to count the number of text-based files for all projects, and consider the projects with more text-based files to also have more code files. 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 code file, such as whether the identifiers, parameters, indentation, and formatting are consistent, whether the constants (other than 0 and 1) are hardcoded into the code, 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, such as all of the steps of a software development process and the current status of a project, we check the existing documents. 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* is not ideal.

All the last three sections are about the empirical measurements. For some qualities, the empirical measurements also affect the score. We use two command-line software tools *GitStats* and *scc* 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 display outputs in the format of web pages [22]; the other one is Sloc Cloc and Code (as known as *scc*) [10], 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 also manually collect information for projects held on GitHub, such as the numbers of stars, forks, people watching this repository, open pull requests, and closed pull requests.

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 VM on the same computer, one at a time, and destroy them after measuring.

We spend about two hours grading one 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

3.4.1 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 contact all teams on the list.

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.

3.4.2 Interview Question Selection

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

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, and we also ask follow-up questions when necessary. Based on our experience, the interviewees usually bring up some exciting ideas that we do not expect, and it is worth expanding on these topics.

3.4.3 Interview Process

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 our discussions to transcribe them better.

3.5 Applying the Method to MI

Based on the principles in Section 3.1, we selected the MI domain and the sub-group of software with Visualization function shown in Figure 1.1. We also included MI domain experts in our team.

By using the method in Section 3.2.1, we identified 48 MI software projects as the candidates from publications [9] [11] [24], online articles related to the domain [20] [29] [49], forum discussions related to the domain [60], etc.

Appendix B shows all the 48 software packages. Among them, there were several ones that we could not find in their source code, such as *MicroDicom*, *Aliza*, and *jivex*, etc. 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. Next, we focused on the MI software providing visualization functions. Some 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*, *dcm4che*, etc. We also eliminated these from the list. After that, there were 29 software products left on the list. We still preferred projects using git and GitHub and being updated recently, but did not apply another filtering since the number of packages was already below 30. However, 27 out of the 29 software packages on the final list used git, and 24 chose GitHub. Furthermore, 27 packages had the latest updates after the year 2018, and 23 after 2020.

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.

Going through the interview process in Section 3.4, we started with the teams with higher scores on our list, and eventually contacted all of them. As a result, developers/architects from 8 teams have participated in our interviews so far. Before contacting any interviewee candidate, we received ethics clearance from the McMaster University Research Ethics Board (Appendix D).

Chapter 4

Measurement Results

This section shows the summary of the measurement results. The detailed data can be found in the repository https://github.com/smiths/AIMSS.

Table 4.1 shows the 29 software packages that we measured. We tried to find the initial release dates (Rlsd) in the documents, such as the introduction and release notes. We did not find this date for two projects and marked them 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 eight projects, and we are not sure about the rest.

We counted the number of contributors (NOC) and lines of code (LOC). we considered anyone who made at least one accepted commit to the source code as a contributor. Thus, it does not mean that any development team has are 100 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 4.1 also shows the supported OS for each software package, and 25 of them could work on all three Windows (W), macOS (M), and Linux (L) systems. 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.

Software	Rlsd	Updated	Fnd	NOC	LOC	OS			W
						W	M	L	
3D Slicer [38]	1998	2020-08	X	100	501451	X	X	X	
Ginkgo CADx [77]	2010	2019-05		3	257144	X	X	X	
XMedCon [51]	2000	2020-08		2	96767	X	X	X	
Weasis [57]	2010	2020-08		8	123272	X	X	X	
MRIcroGL [40]	2015	2020-08		2	8493	X	X	X	
SMILI [12]	2014	2020-06		9	62626	X	X	X	
ImageJ [58]	1997	2020-08	X	18	9681	X	X	X	
Fiji [63]	2011	2020-08	X	55	10833	X	X	X	
DicomBrowser [7]	2012	2020-08		3	5505	X	X	X	
3DimViewer [68]	?	2020-03	X	3	178065	X	X		
Horos [31]	?	2020-04		21	561617		X		
OsiriX Lite [62]	2004	2019-11		9	544304		X		
dwv [45]	2012	2020-09		22	47815	X	X	X	X
Drishti [42]	2012	2020-08		1	268168	X	X	X	
BioImage Suite Web [53]	2018	2020-10	X	13	139699	X	X	X	X
OHIF Viewer [80]	2015	2020-10		76	63951	X	X	X	X
Slice:Drop [25]	2012	2020-04		3	19020	X	X	X	X
GATE [35]	2011	2020-10		45	207122		X	X	
ITK-SNAP [78]	2006	2020-06	X	13	88530	X	X	X	
ParaView [3]	2002	2020-10	X	100	886326	X	X	X	X
MatrixUser [43]	2013	2018-07		1	23121	X	X	X	

Software	Rlsd	Updated	Fnd	NOC	LOC	OS			W
						W	M	L	
DICOM Viewer [2]	2018	2020-04	X	5	30761	X	X	X	
INVESALIUS 3 [5]	2009	2020-09		10	48605	X	X	X	
medInria [21]	2009	2020-11		21	148924	X	X	X	
dicompyler [52]	2009	2020-01		2	15941	X	X		
MicroView [33]	2015	2020-08		2	27470	X	X	X	
Papaya [56]	2012	2019-05		9	71831	X	X	X	
AMIDE [44]	2006	2017-01		4	102827	X	X	X	
Gwyddion [50]	2004	2020-11		38	643427	X	X	X	

Table 4.1: Final software list

Most of the projects used more than one programming language, including a primary language that the developers used the most. Figure 4.1 shows the primary languages versus the number of projects using them.

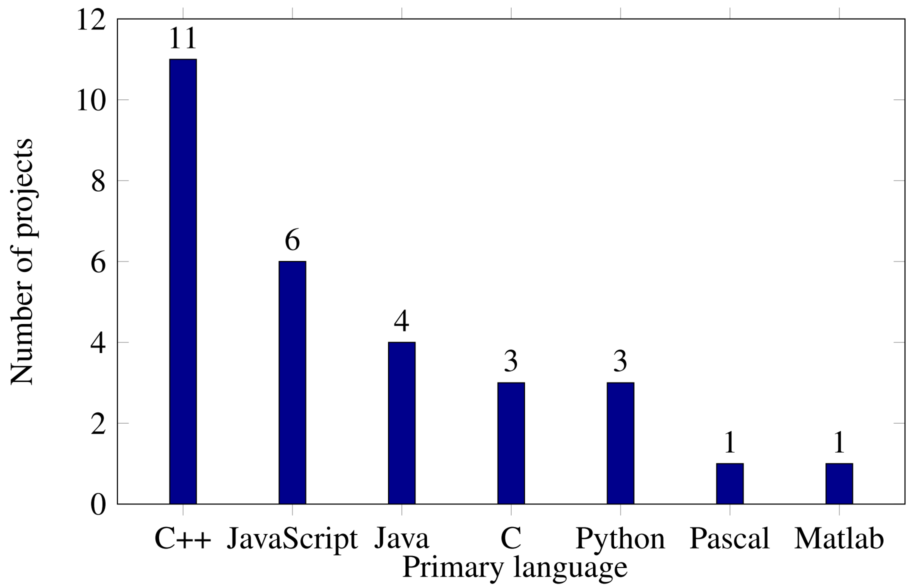

Figure 4.1: Primary languages versus number of projects using them

We failed installing *DICOM Viewer*, so we could not test *surface reliability* and *surface robustness* for it. We kept this software on our list because the other seven qualities do not rely on a successful installation. Besides, it used a unique dependency, and we wanted to keep the diversity.

4.1 Installability

Figure 4.2 lists the scores of *installability*.

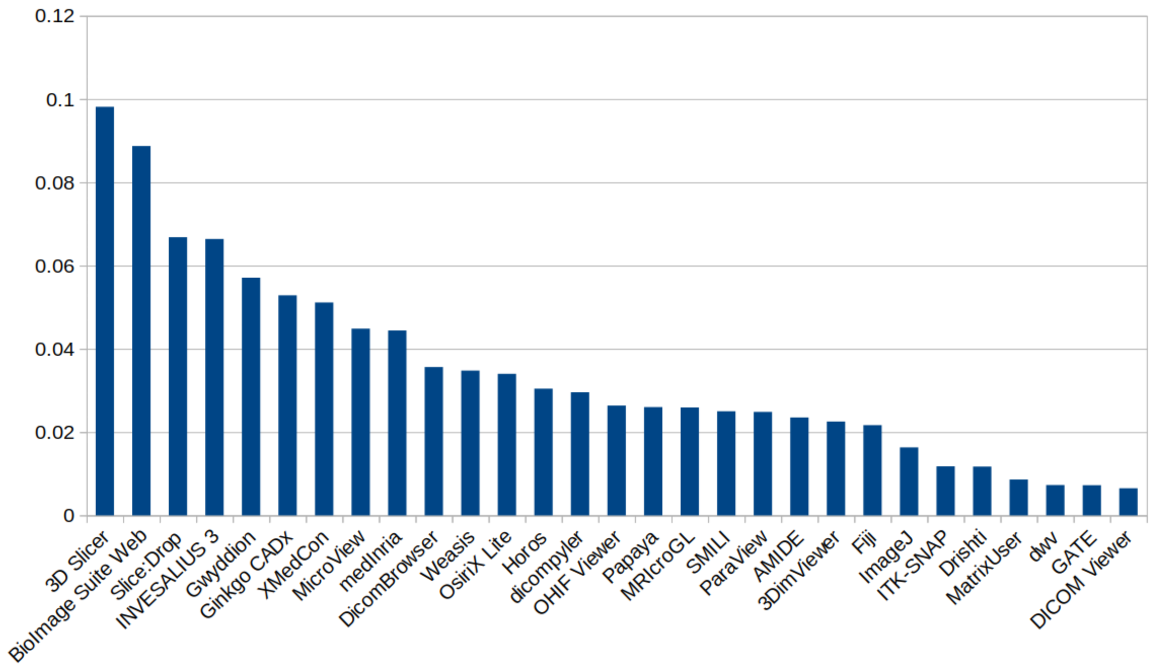

Figure 4.2: AHP installability scores

We found installation instructions for 16 projects. Among the ones without such an instruction, *BioImage Suite Web* and *Slice:Drop* were web applications and provided online versions to use, thus they needed no installation. Installing 10 of the projects needed extra dependencies. Five of them are the web applications in Table 4.1, 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 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 due to the significant convenience. *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 due to some issues that we could not solve. We spent a reasonable amount of time on these problems, then considered them major obstacles for normal users if we still did 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 a cloud 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 4.3. Generally speaking, the packages with higher scores adopted more techniques to improve *correctness*, and had better documents for us to verify it.

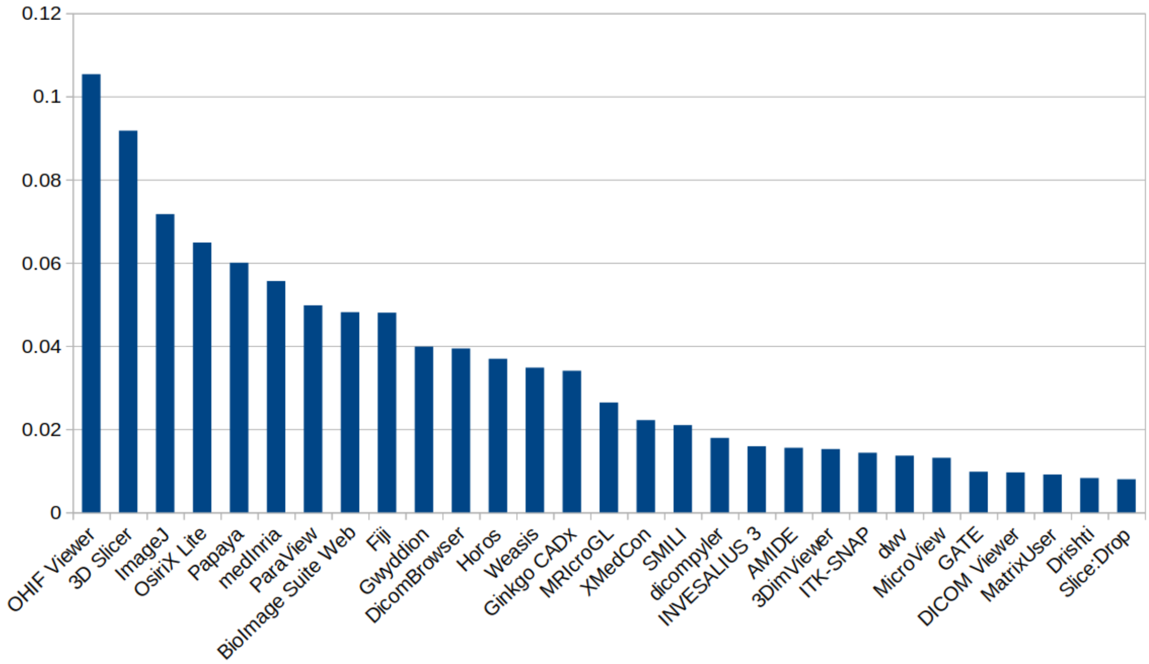

Figure 4.3: AHP correctness & verifiability scores

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 [27]. It can reveal the bugs at an earlier stage of the development process, and the absence of unit testing may cause worse *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 following changes. However, it did not record the conditions for previous versions. We also could not identify the theory manuals for all of the projects. It seems that 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*.

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 4.4 shows the AHP results.

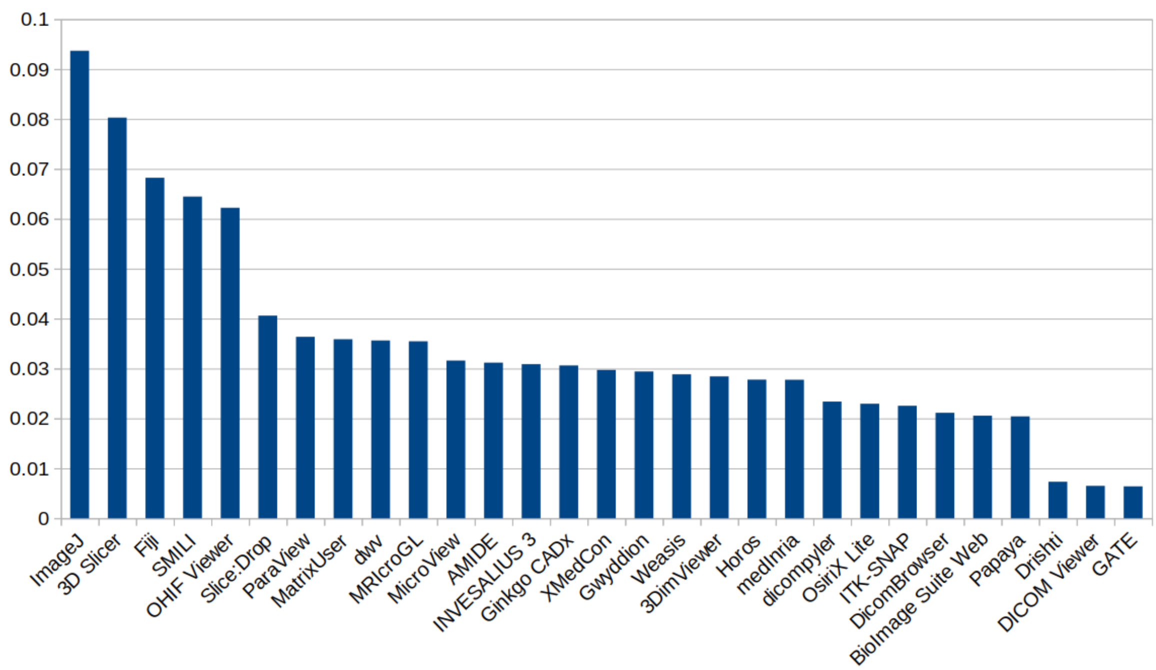

Figure 4.4: AHP surface reliability scores

When applying basic operations with the software packages, we found out that *Drishti* crashed during loading damaged image files, and *GATE* could not open macro files and lost response several times.

4.4 Surface Robustness

Figure 4.5 presents the scores for *surface robustness*.

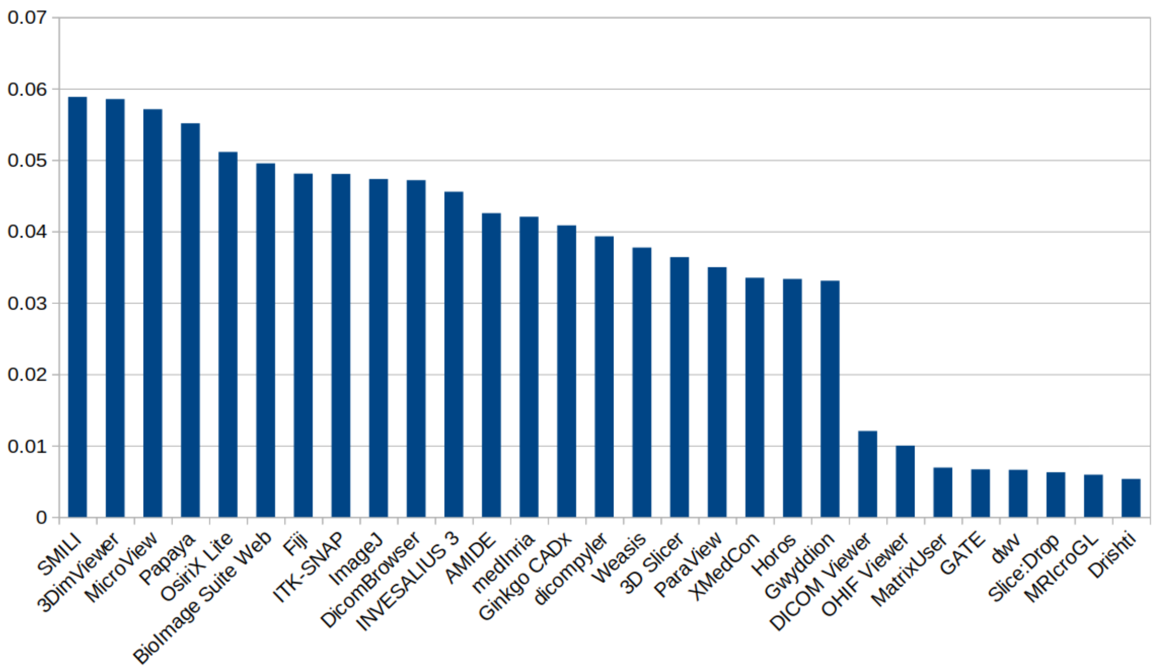

Figure 4.5: AHP surface robustness scores

The packages with higher scores elegantly handled the unexpected/unanticipated inputs, normally 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. *MatrixUser*, *dwv*, *Slice:Drop*, and *MRIcroGL* ignored the incorrect format of the input files, and displayed blank or meaningless images. *Drishti* successfully detected the unexpected/unanticipated inputs, but the software crashed as a result. For unknown reasons, *GATE* failed to load both correct and incorrect inputs.

4.5 Surface Usability

Figure 4.6 lists the AHP scores for *surface usability*.

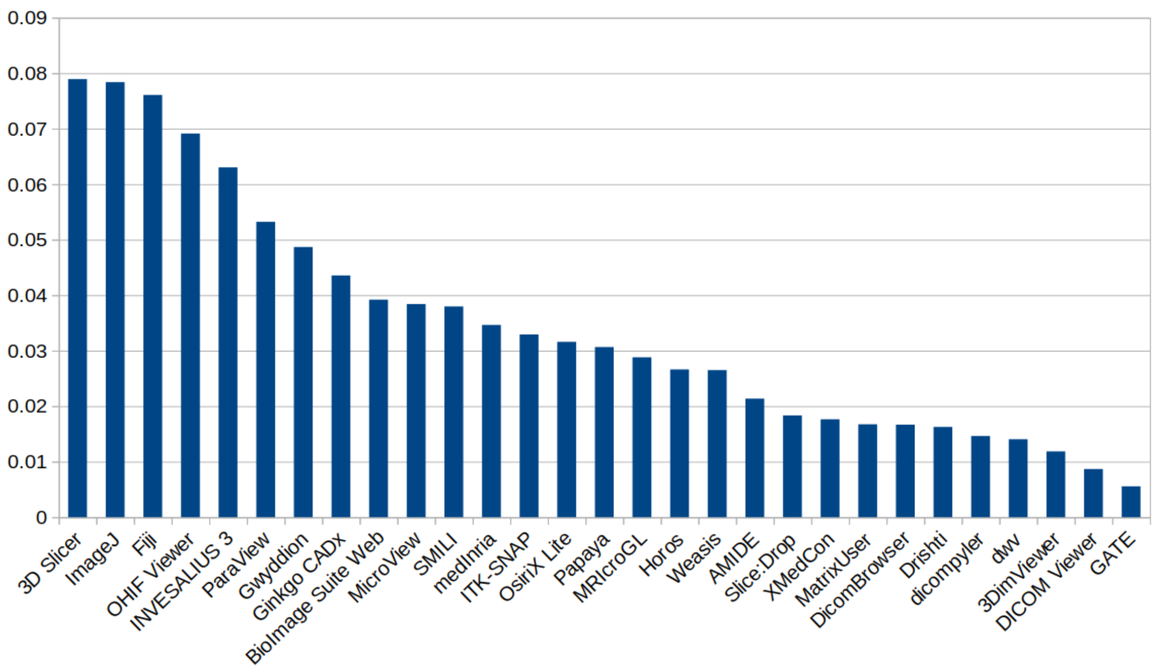

Figure 4.6: AHP surface usability scores

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

The ones 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 4.2 shows the user support models by the number of projects using them. Maybe not every team intended to use GitHub issues to answer users' questions, but many users use them to seek help.

User support model	Number of projects
GitHub issue	24
GitLab issue, SourceForge discussions	2
FAQ	12
Forum	10
E-mail address	9
Troubleshooting	2
Contact form	1

Table 4.2: User support models by number of projects

4.6 Maintainability

Figure 4.7 shows the AHP results for *maintainability*.

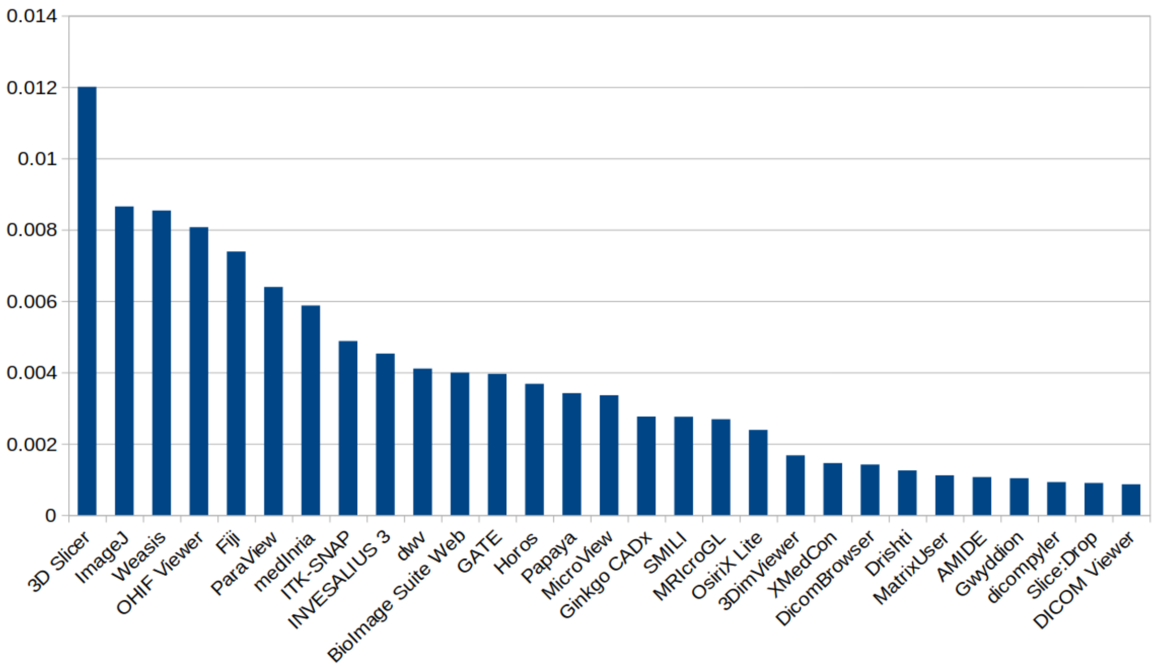

Figure 4.7: AHP maintainability scores

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 4.3 shows which projects had these documents.

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

Table 4.3: Software with the maintainability documents

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.

4.7 Reusability

Figure 4.7 shows the AHP results for *reusability*.

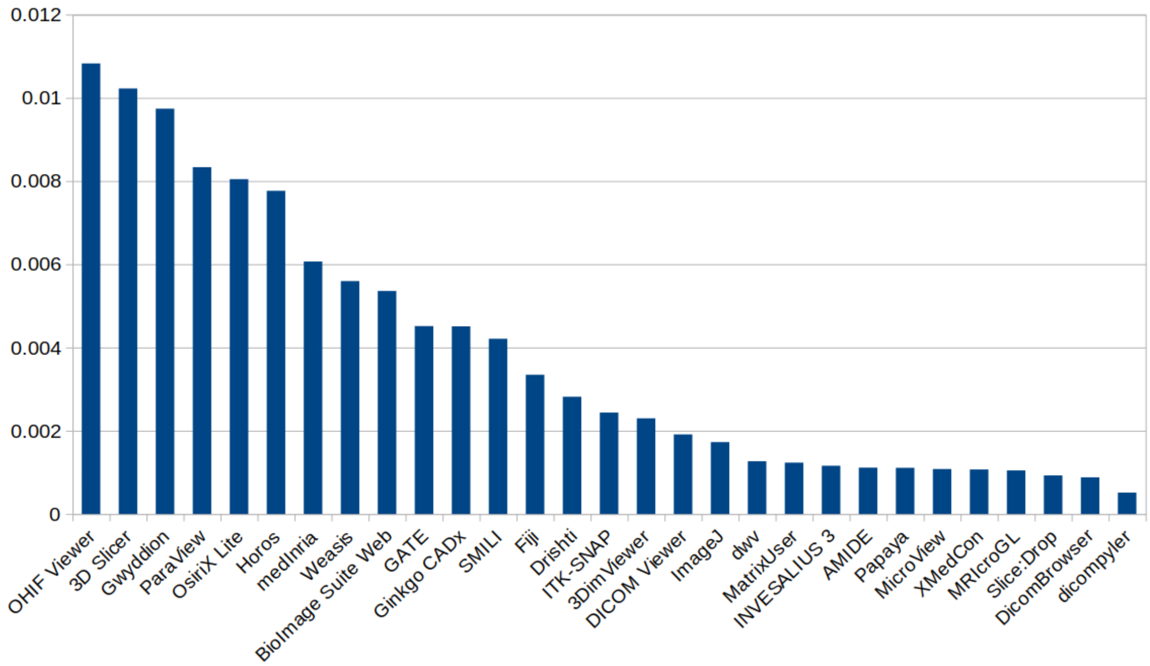

Figure 4.8: AHP reusability scores

As described in Section 3.3.1, we gave higher scores to the projects with an API document and more code files. As shown in Table 4.3, seven projects had API documents. Figure 4.9 shows the number of text-based files by projects.

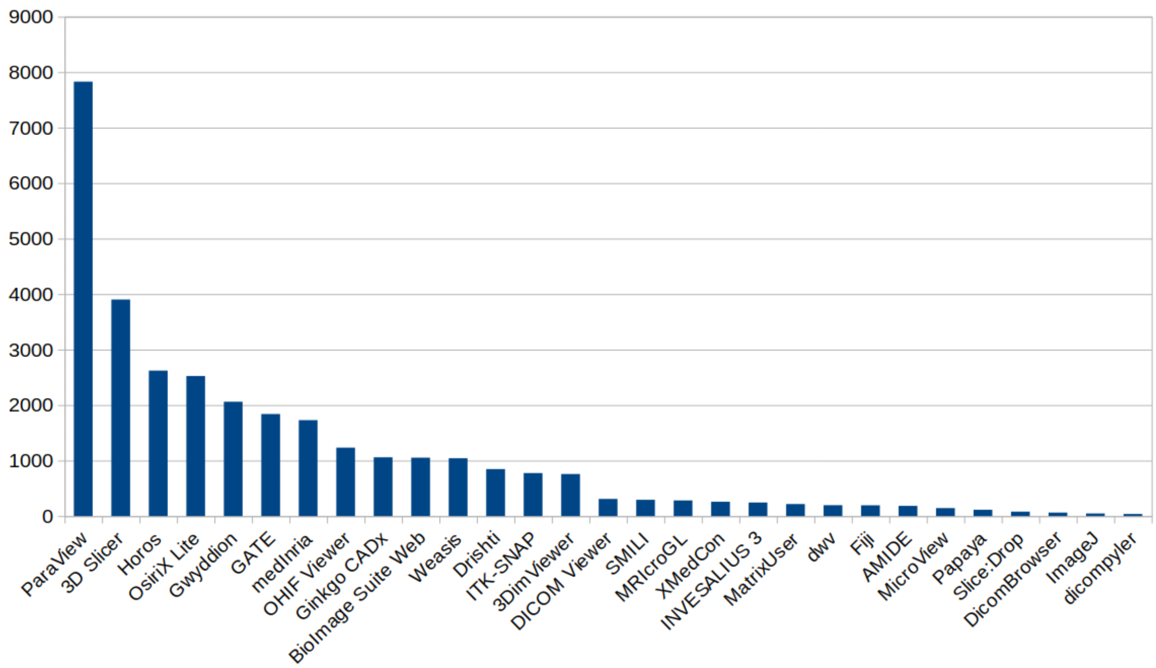

Figure 4.9: Number of text-based files files by projects

4.8 Surface Understandability

Figure 4.10 shows the scores for *surface understandability*.

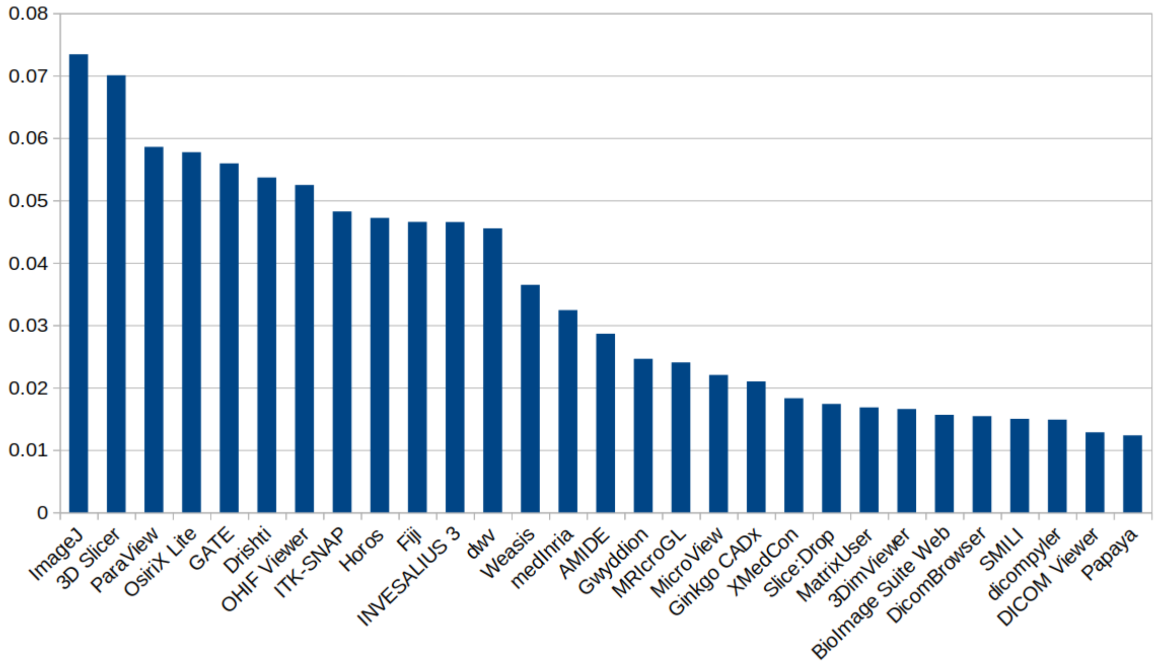

Figure 4.10: AHP surface understandability scores

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 only 3 out of the 29, which are *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 4.11 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.

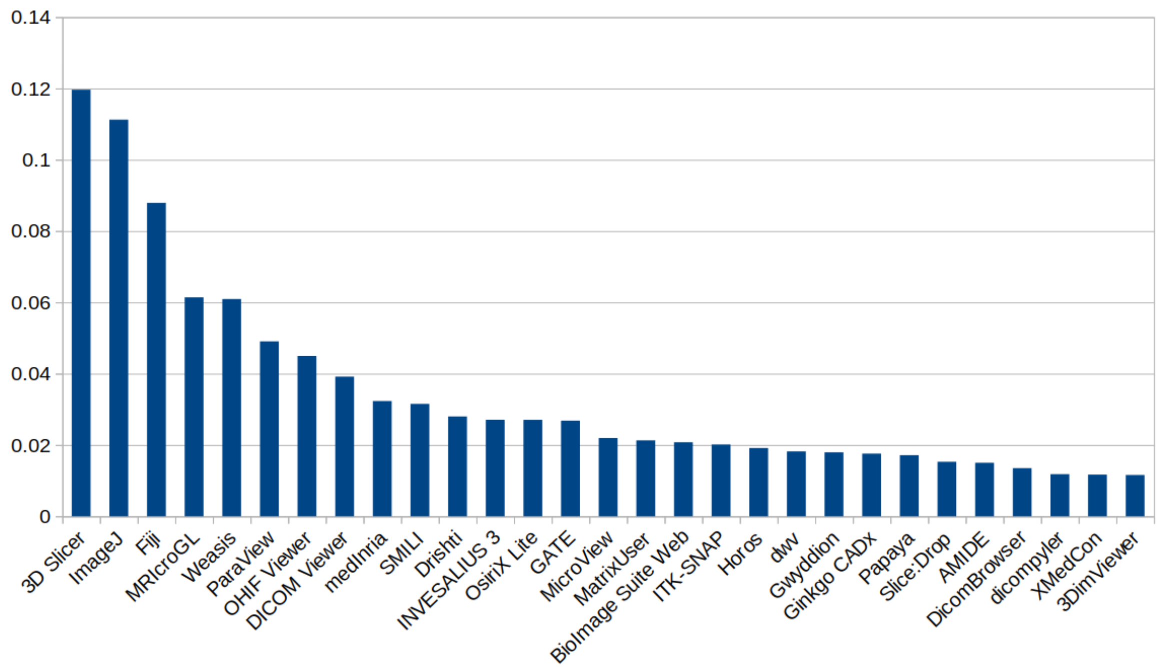

Figure 4.11: AHP visibility/transparency scores

Table 4.4 shows the projects which had documents for the development process, project status, development environment, and release notes.

Software	Dev process	Proj status	Dev env	Rls notes
3D Slicer	X	X	X	X
Weasis [57]			X	X
MRIcroGL [40]				X
SMILI [12]				X
ImageJ [58]	X	X	X	X
Fiji [63]	X	X	X	
Horos [31]				X
OsiriX Lite [62]				X
dwv [45]				X
Drishti [42]				X
BioImage Suite Web			X	
OHIF Viewer [80]			X	X
GATE [35]				X
ITK-SNAP [78]				X
ParaView [3]		X		
MatrixUser [43]				X
DICOM Viewer [2]			X	X
INVESALIUS 3 [5]				X
medInria [21]			X	X
MicroView [33]				X
Gwyddion [50]				X

Table 4.4: Software with the visibility/transparency documents

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. We decided to make all nine qualities equally important, so the score of each quality affects the overall scores on the same scale.

Figure 4.12 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.

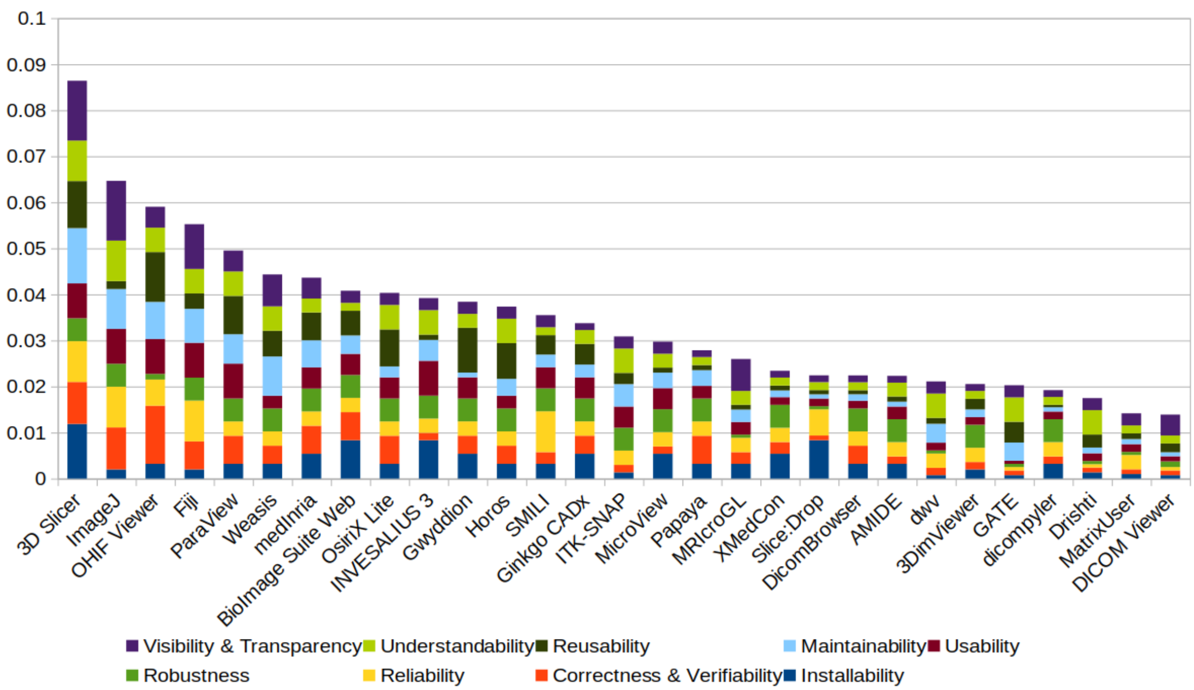

Figure 4.12: Overall AHP scores for all 9 software qualities

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.

Chapter 5

Interviews with Developers

This section summarizes some answers from the interviews with developers. 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 he wrote down the answers and sent them to us.

In this section, we only discuss the answers which we think are the most exciting and essential. We summarize the rest part of the interviews in Appendix C. The interviewees may have provided multiple answers to each question. When counting the number of interviewees for each response, we count every interviewee with this answer.

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.

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

Group	Obstacle	Number of interviewees with the answer	
		as a current obstacle	as a past obstacle
1	Lack of fundings	3	
	Lack of time to devote to the project	2	1
2	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
3	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	
	Codebase is so big and complicated and only a few people fully understand it	1	
	Hard to transfer to new technologies		2
	Hard to understand users' needs		1
	Hard to maintain good documentations		1

Table 5.1: Current and past obstacles by the numbers of interviewees with the answers

The interviewees provided some potential and proven solutions for the problems in Table 5.1. We group these pain points into three major groups, and put the less mentioned ones into the group *Others*. Sections 5.1.2, 5.1.2, and 5.1.3 include further discussion about the three major groups of pain points and their solutions.

5.1.1 Pain Points in Group 1

We summarize the pain points in Group 1 as **the lack of fundings and time.**

We also summarize the potential and proven solutions as follows.

Potential solutions from interviewees:

- shifting from development mode toward maintenance mode when developers are not enough;
- 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 researches. The interviewee from the *ITK-SNAP* team thought enough fundings was a way to solve to problem of lacking time, because they could hire more dedicated developers. On the other hand, the interviewee from the *ITK-SNAP* 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 solve the funding problem fundamentally. However, they provided ideas to save time, such as better documentation, third-party plugins, and good CI/CD tools.

5.1.2 Pain Points in Group 2

We summarize the pain points in Group 2 as **the difficulty to balance between four factors: cross-platform compatibility, convenience to development & maintenance, per-**

formance, and security. They are also related to the choice between native application and web application.

We also summarize the potential and proven solutions as follows.

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 fundings;
- 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 5.2 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.

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 5.2: Teams' choices between native application and web application

The interviewees talked about the advantages and disadvantages of the two choices. We summarize the opinions from the interviewees as follows.

Native application:

- Advantages
  - higher performance
- Disadvantages
  - hard to achieve cross-platform compatibility
  - more complicated build process

Web application:

- Advantages

  - easy to achieve cross-platform compatibility
  - simpler build process

- Disadvantages
  - lower performance (without a backend)
  - may not meet the requirements of protecting patients' privacy (with a backend)
  - extra cost for backend servers (with a backend)

The interviewees did not conclude any perfect solutions.

5.1.3 Pain Points in Group 3

The pain point in Group 3 is **the lack of access to real-world datasets for testing.**

We also summarize the potential and proven solutions as follows.

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 to;
- if the team has access to MRI scanners, self-building MI datasets;

- if the team has connections with MI equipment manufacturers, asking for their help on data format problems.

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

By asking questions 11 and 19, we tried to understand the interviewees' opinions on documentation. We also aimed to find out the quality of documentations in their projects.

Questions 11 and 19 are as follows,

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 5.3 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. However, most of them saw the need to improve their documentation, and only three of them thought that their documentations conveyed information clearly enough.

Opinion on documentation	Number of interviewees with the answer
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 preferable than documentation	2
Users help each other by referring to documentation	1

Table 5.3: Opinions on documentation by the numbers of interviewees with the answers

Table 5.4 lists some of the tools and methods mentioned by the interviewees, which they used for documentation.

Tool or method for documentation	Number of interviewees with the answer
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 5.4: Tools and methods for documentation by the numbers of interviewees with the answers

5.3 Contribution Management and Project Management

By asking questions 5, 13, and 14, we tried to understand how the teams managed the contributions and their projects.

Questions 5, 13, and 14 are as follows,

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?

Maybe some team had a documented process for accepting new contributions, but none 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 5.5 shows the number of times the interviewees mentioned the methods of receiving contributions.

Method of receiving contributions	Number of interviewees with the answer	
	as a current method	as a past method
GitHub with pull requests	8	
Code contributions from emails		3
Code contributions from forums		1
Sharing the git repository by email		1

Table 5.5: Methods of receiving contributions by the numbers of interviewees with the answers

Additionally, 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 5.6 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.

Software development model	Number of interviewees with the answer
Undefined/self-directed	3
Similar to Agile	2
Similar to Waterfall	1
Agile	1
Waterfall	1

Table 5.6: Software development models by the numbers of interviewees with the answers

Some interviewees mentioned the project management tools they used, which are in Table 5.7. Generally speaking, they talked about two types:

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

Project management tools	Number of interviewees with the answer
GitHub	3
Issue trackers	1
Bug trackers	1
Jira	1
Wiki page	1
Google Doc	1
Confluence	1

Table 5.7: Project management tools by the numbers of interviewees with the answers

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 question "Do you think improving this process can tackle the current problem?", but the interviewee from the *OHIF* team gave a positive answer.

Chapter 6

Threat to Validity

This section lists all the potential threats to the validity of our research.

1. 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 not be thorough in revealing their status fully.

2. We used the grading template in Appendix A 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.

3. 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.

4. 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.

5. 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.

Chapter 7

Recommendations

I think the recommendations can originate from both parts - measurements and interviews.

Chapter 8

Conclusions

No clues yet. Should be started at a later stage.

Bibliography

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

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

[3] J. Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *Visualization Handbook*, 01 2005.

[4] Aliza. Aliza ms dicom viewer. https://www.aliza-dicom-viewer.com/, 2021. [Online; accessed 27-May-2021].

[5] Paulo Amorim, Thiago Franco de Moraes, Helio Pedrini, and Jorge Silva. Invesalius: An interactive rendering framework for health care support. page 10, 12 2015.

[6] S. Angenent, Eric Pichon, and Allen Tannenbaum. Mathematical methods in medical image processing. *Bulletin (new series) of the American Mathematical Society*, 43:365–396, 07 2006.

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

[8] Isaac N. Bankman. Preface. In Isaac N. Bankman, editor, *Handbook of Medical Imaging*, Biomedical Engineering, pages xi – xii. Academic Press, San Diego, 2000.

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

[10] Ben Boyter. Sloc cloc and code. `https://github.com/boyter/scc`, 2021. [Online; accessed 27-May-2021].

[11] Andreas Brühschwein, Julius Klever, Anne-Sophie Hoffmann, Denise Huber, Elisabeth Kaufmann, Sven Reese, and Andrea Meyer-Lindenberg. 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.

[12] Shekhar Chandra, Jason Dowling, Craig Engstrom, Ying Xia, Anthony Paproki, Ales Neubert, David Rivest-Hénault, Olivier Salvado, Stuart Crozier, and Jurgen Fripp. A lightweight rapid application development framework for biomedical image analysis. *Computer Methods and Programs in Biomedicine*, 164, 07 2018.

[13] Robert Choplin, J Boehme, and C Maynard. Picture archiving and communication systems: an overview. *Radiographics : a review publication of the Radiological Society of North America, Inc*, 12:127–9, 02 1992.

[14] CIT. Mipav. `https://mipav.cit.nih.gov`, 2020. [Online; accessed 27-May-2021].

[15] ClearCanvas. Clearcanvas. https://github.com/ClearCanvas/ClearCanvas, 2015. [Online; accessed 27-May-2021].

[16] James Edward Corbly. The free software alternative: Freeware, open source software, and libraries. *Information Technology and Libraries*, 33(3):65–75, Sep. 2014.

[17] cornerstone. cornerstone. https://github.com/cornerstonejs/cornerstone, 2020. [Online; accessed 27-May-2021].

[18] dcm4che. dcm4che. https://github.com/dcm4che/dcm4che, 2021. [Online; accessed 27-May-2021].

[19] DCMTK. Dcmtk. https://github.com/DCMTK/dcmtk, 2021. [Online; accessed 27-May-2021].

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

[21] Pierre Fillard, Nicolas Toussaint, and Xavier Pennec. Medinria: Dt-mri processing and visualization software. 04 2012.

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

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

[24] Daniel Haak, Charles-E Page, and Thomas Deserno. A survey of dicom viewer software to integrate clinical research and medical imaging. *Journal of digital imaging*, 29, 10 2015.

[25] Daniel Haehn. Slice:drop: collaborative medical imaging in the browser. pages 1–1, 07 2013.

[26] Daniel Haehn, Nicolas Rannou, Banu Ahtam, Patricia Grant, and Rudolph Pienaar. Neuroimaging in the browser using the x toolkit. volume 8, 10 2012.

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

[28] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, 2009.

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

[30] Frank Heckel, Michael Schwier, and Heinz-Otto Peitgen. Object-oriented application development with mevislab and python. pages 1338–1351, 01 2009.

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

[32] Raster Images. Oviyam. `http://oviyam.raster.in/index.html`, 2021. [Online; accessed 27-May-2021].

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

[34] Alessio Ishizaka and Markus Lusti. How to derive priorities in ahp: A comparative study. *Central European Journal of Operations Research*, 14:387–400, 12 2006.

[35] 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. Gate: a simulation toolkit for pet and spect. *Physics in medicine and biology*, 49:4543–61, 11 2004.

[36] Sébastien Jodogne. The orthanc ecosystem for medical imaging. *Journal of Digital Imaging*, 31, 05 2018.

[37] Alark Joshi, Dustin Scheinost, Hirohito Okuda, Dominique Belhachemi, Isabella Murphy, Lawrence Staib, and Xenophon Papademetris. Unified framework for development, deployment and robust testing of neuroimaging algorithms. *Neuroinformatics*, 9:69–84, 03 2011.

[38] Ron Kikinis, Steve Pieper, and Kirby Vosburgh. *3D Slicer: A Platform for Subject-Specific Image Analysis, Visualization, and Clinical Support*, volume 3, pages 277–289. 01 2014.

[39] Tae-Yun Kim, Jaebum Son, and Kwanggi Kim. The recent progress in quantitative medical image analysis for computer aided diagnosis systems. *Healthcare informatics research*, 17:143–9, 09 2011.

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

[41] Xiangrui Li, Paul Morgan, John Ashburner, Jolinda Smith, and Chris Rorden. The first step for neuroimaging data analysis: Dicom to nifti conversion. *Journal of Neuroscience Methods*, 264, 03 2016.

[42] Ajay Limaye. Drishti, a volume exploration and presentation tool. volume 8506, page 85060X, 10 2012.

[43] Fang Liu, Julia Velikina, Walter Block, Richard Kijowski, and Alexey Samsonov. Fast realistic mri simulations based on generalized multi-pool exchange tissue model. *IEEE Transactions on Medical Imaging*, PP:1–1, 10 2016.

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

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

[46] Matthew McCormick, Xiaoxiao Liu, Julien Jomier, Charles Marion, and Luis Ibanez. Itk: Enabling reproducible research and open science. *Frontiers in neuroinformatics*, 8:13, 02 2014.

[47] Hemant Kumar Mehta. *Mastering Python scientific computing: a complete guide for Python programmers to master scientific computing using Python APIs and tools*. Packt Publishing, 2015.

[48] MicroDicom. Microdicom. `https://www.microdicom.com/`, 2021. [Online; accessed 27-May-2021].

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

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

[51] E Nolf, Tony Voet, Filip Jacobs, R Dierckx, and Ignace Lemahieu. (x)medcon * an opensource medical image conversion toolkit. *European Journal of Nuclear Medicine and Molecular Imaging*, 30:S246, 08 2003.

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

[53] Xenophon Papademetris, Marcel Jackowski, Nallakkandi Rajeevan, Robert Constable, and Lawrence Staib. Bioimage suite: An integrated medical image analysis suite. 1, 01 2005.

[54] 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. A survey of the practice of computational science. SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

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

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

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

[58] Curtis Rueden, Johannes Schindelin, Mark Hiner, Barry Dezonia, Alison Walter, and Kevin Eliceiri. Imagej2: Imagej for the next generation of scientific image data. *BMC Bioinformatics*, 18, 11 2017.

[59] Thomas L. Saaty. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9–26, 1990. Desicion making by the analytic hierarchy process: Theory and applications.

[60] Ravi Samala. Can anyone suggest free software for medical images segmentation and volume? `https://www.researchgate.net/post/Can_anyone_suggest_free_software_for_medical_images_segmentation_and_volume`, 03 2014. [Online; accessed 31-January-2020].

[61] Santesoft. Sante dicom viewer lite. `https://www.santesoft.com/win/sante-dicom-viewer-pro/sante-dicom-viewer-pro.html`, 2021. [Online; accessed 27-May-2021].

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

[63] 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. Fiji: An open-source platform for biological-image analysis. *Nature methods*, 9:676–82, 06 2012.

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

[65] Spencer Smith. Systematic development of requirements documentation for general purpose scientific computing software. pages 205–215, 09 2006.

[66] Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. Quality definitions of qualities. Manuscript in preparation, 2020.

[67] Spencer Smith, Jacques Carette, Olu Owojaiye, Peter Michalski, and Ao Dong. Methodology for assessing the state of the practice for domain x. Manuscript in preparation, 2021.

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

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

[70] Marcel van Herk. Conquest. `https://github.com/marcelvanherk/Conquest-DICOM-Server`, 2021. [Online; accessed 27-May-2021].

[71] Open Dicom Viewer. Open dicom viewer. `https://sourceforge.net/p/opendicomviewer/code/HEAD/tree/`, 2011. [Online; accessed 27-May-2021].

[72] Visus. Jivex dicom viewer. `https://www.visus.com/en/downloads/jivex-dicom-viewer.html`, 2021. [Online; accessed 27-May-2021].

[73] Wikipedia contributors. Medical image computing — Wikipedia, the free encyclopedia, 2021. [Online; accessed 25-July-2021].

[74] Wikipedia contributors. Medical imaging — Wikipedia, the free encyclopedia, 2021. [Online; accessed 25-July-2021].

[75] 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. Best practices for scientific computing. *PLoS Biology*, 12:e1001745, 01 2014.

[76] 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. Best practices for scientific computing. *PLoS Biology*, 12:e1001745, 01 2014.

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

[78] Paul A. Yushkevich, Joseph Piven, Heather Cody Hazlett, Rachel Gimpel Smith, Sean Ho, James C. Gee, and Guido Gerig. User-guided 3D active contour segmentation of anatomical structures: Significantly improved efficiency and reliability. *Neuroimage*, 31(3):1116–1128, 2006.

[79] Xiaofeng Zhang, Nadine Smith, and Andrew Webb. 1 - medical imaging. In David Dagan Feng, editor, *Biomedical Information Technology*, Biomedical Engineering, pages 3–27. Academic Press, Burlington, 2008.

[80] Erik Ziegler, Trinity Urban, Danny Brown, James Petts, Steve D. Pieper, Rob Lewis, Chris Hafey, and Gordon J. Harris. 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):336–345, 2020. PMID: 32324447.

Appendix A

Full Grading Template

appendix here

Appendix B

Full Software List Before Filtering

Table B.1 lists the 48 software before filtering. We selected 29 of them to the final list, which are all open-source and in Visualization (V) sub-group. We found software packages in sub-groups Tool Kit (TK) and PACS but removed them from the final list. The table also shows the sources of identifying them.

Software	Final list	Open-source	Sub-group	Source						
				[9]	[11]	[24]	[20]	[29]	[49]	[60]
3D Slicer [38]	X	X	V			X	X	X		X
Ginkgo CADx [77]	X	X	V	X	X	X	X	X	X	
XMedCon [51]	X	X	V			X	X	X	X	
Weasis [57]	X	X	V			X	X	X	X	
MRIcroGL [40]	X	X	V			X	X	X		X
SMILI [12]	X	X	V				X	X	X	
ImageJ [58]	X	X	V			X	X	X		X
Fiji [63]	X	X	V							X
DicomBrowser [7]	X	X	V				X	X		

Software	Final list	Open-source	Sub-group	Source [9]	[11]	[24]	[20]	[29]	[49]	[60]
3DimViewer [68]	X	X	V	X			X	X		
Horos [31]	X	X	V	X	X					
OsiriX Lite [62]	X	X	V	X	X	X				
dwv [45]	X	X	V	X		X				
Drishti [42]	X	X	V	X						
BioImage Suite Web [53]	X	X	V			X				
OHIF Viewer [80]	X	X	V	Other source: [80]						
Slice:Drop [25]	X	X	V			X				
GATE [35]	X	X	V				X			
ITK-SNAP [78]	X	X	V							X
ParaView [3]	X	X	V							X
MatrixUser [43]	X	X	V					X		
DICOM Viewer [2]	X	X	V					X		
INVESALIUS 3 [5]	X	X	V					X		
medInria [21]	X	X	V			X				
dicompyler [52]	X	X	V						X	
MicroView [33]	X	X	V			X				
Papaya [56]	X	X	V				X	X	X	
AMIDE [44]	X	X	V				X	X	X	
Gwyddion [50]	X	X	V					X		
VTK [64]		X	TK							X
ITK [46]		X	TK							X

Software	Final list	Open-source	Sub-group	Source						
				[9]	[11]	[24]	[20]	[29]	[49]	[60]
DCMTK [19]		X	TK					X	X	
XTK [26]		X	TK	Other source: it is used by slice:drop						
dcm4che [18]		X	TK	X			X	X		
cornerstone [17]		X	TK			X				
dcm2niix [41]		X	TK					X		
orthanc [36]		X	PACS	X				X		
Conquest [70]		X	PACS	X						
ClearCanvas [15]		X	PACS, V	X						
Open Dicom Viewer [71]		X	V			X			X	
MicroDicom [48]			V	X					X	
Aliza [4]			V				X		X	
JiveX [72]			V			X			X	
MIPAV [14]			V			X				
Oviyam [32]			V			X				
MeVisLab [30]			V							X
Sante DICOM Viewer Lite [61]			V						X	
Navegatium DICOM Viewer			V						X	

Table B.1: Full software list before filtering

Appendix C

Other Interview Answers

We asked 20 interview questions to the nine interviewees from eight software projects. We discuss the answers to interview questions 5, 9, 10, 11, 12, 13, 14, and 19 in Section 5, and summarize the answers to the other questions in this section.

Q1. Interviewees' current position/title? degrees?

Six of the nine interviewees revealed their position/title, such as CEO of a company, endowed chair and professor in universities, software engineers in a commercial company and a hospital.

Most of them answered their backgrounds and degrees. Table C.1 shows the highest academic degrees the participants have, and Table C.2 shows what majors they studied. Many of the interviewees studied in multiple majors.

Highest degree	Number of interviewees
PHD	4
Master	3
Bachelor	0
Unspecified academic degree	2

Table C.1: Interviewees' highest academic degrees

Major	Number of interviewees
Computer Science	4
Physics	2
Biomedical Engineering	1
Neuroimaging	1
Geology (image analysis)	1
Media Arts and Sciences	1
Mechanical Engineering	1
Materials Engineering	1
Psychology	1

Table C.2: Interviewees' majors at university

Q2. Interviewees' contribution to/relationship with the software?

Table C.3 shows the interviewees' roles and responsibilities in the projects. One of the participants did not explicitly mention his role, but implicitly revealed that he was a primary contributor to the project.

Role in the projects	Number of interviewees
Chief Architect	2
Lead Developer	1
Core Developer	5
Unspecified	1

Table C.3: Interviewees' roles in the projects

Q3. Length of time the interviewee has been involved with this software?

Table C.4 shows the distribution of the lengths of time the interviewees had worked on the projects.

Length of time in the projects	Number of interviewees
0-1	1
2-5	0
6-10	2
11-15	3
16-20	2
21-25	1

Table C.4: Lengths of time that the interviewees worked in the projects

Q4. How large is the development group?

The size of each group grows and shrinks over the years. Most teams mentioned that the team members join and leave. Some teams said that when there was sufficient funding, they could afford more developers.

Table C.5 shows the numbers of active members at the time of interviews. The members include people working on development and project management.

Number of current members	Number of projects
1-3	5
4-6	3

Table C.5: Numbers of current members in the projects

As shown in the table, no team had a vast number of members. Some projects had more developers, such as *3D Slicer*; on the other hand, some teams such as *dwv* had only one primary developer, plus a maximum of two or three developers occasionally.

3D Slicer is a special case, because it supports third-party extensions. So there have been community members developing and maintaining these extensions. Table C.5 does not include these members.

Q6. What is the typical background of a developer?

Not all interviewees could clearly answer this question. Many of them talked about the backgrounds of members with who they were familiar. Table C.6 shows the number of times all interviewees mentioned a background.

Background of a developer	Number of interviewees with the answer
Computer Science, Information Technology, and Software Development	6
Imaging	2
Medical Imaging	1
Mathematics	1
Biomedical Engineering	1
Computer Aided Medical Procedures	1
Physician	1

Table C.6: Backgrounds of developers by the numbers of interviewees with the answers

Q7. What is your estimated number of users? How did you come up with that estimate?

None of the interviewees knew the exact number of users. Some of them provided estimations based on different facts. However, we do not think these numbers are comparable to each other.

Software	Rough estimation	Considered facts
3D Slicer	100,000	Search results on Google Scholar; number of new posts per year on slicer.org; number of downloads.
INVESALIUS 3	75,000	Number of random IDs created by new installation.
dwv	No estimation	About 20 companies integrated *dwv* in their products.
BioImage Suite Web	100 active users	The interviewee only counted the users from several Universities who were active users.
ITK-SNAP	10,000 plus	Number of downloads.
MRIcroGL	No estimation	It is the top 1 downloaded software on this NITRC list https://www.nitrc.org/top/toplist.php?type=downloads
Weasis	10,000 user used it as least once	Number of profiles.
OHIF	About 5000	Some platforms integrated *OHIF*, and it was hard to know the number of end users.

Table C.7: Rough Estimations for the Number of Users

Table C.7 shows the estimations and how the interviewees made them. It is clear that some estimated only the active users, and some counted users who had used only once. So we do not compare these numbers.

Q8. What is the typical background of a user?

All interviewees provided several different user backgrounds, and all of them mentioned medical researchers or medical professionals. Table C.8 shows the number of times all interviewees mentioned a background.

Background of a user	Number of interviewees with the answer
Medical Researchers	6
Doctors/Health care professionals/Surgeons	5
Student Researchers	4
Patients	3
Paleontologist	1
Biomechanical Engineers	1
Imaging Researchers	1
Mechanical Engineers	1

Table C.8: Backgrounds of users by the numbers of interviewees with the answers

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 C.9 shows the threats to *correctness* by the numbers of interviewees with the answers.

Threat to correctness	Number of interviewees with the answer
Clinical systems produce data in various formats (e.g. DICOM), which have complicated standards. The software needs to handle the complexity.	2
Different types of MI machines can create data in slightly different ways, adding more complexity for the software to handle.	2
Besides viewing, the software has additional functions, which lead to extra complexity.	1
There is lack of real word image data for testing.	1
The team cannot use private data for debugging, even when the data cause problems.	1
The software uses huge datasets for testing, so the tests are expensive and time-consuming.	1
It is hard to well manage releases.	1
The project has no unit tests.	1
The project has no dedicated quality assurance team.	1

Table C.9: Threats to correctness by the numbers of interviewees with the answers

Table C.10 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.

Strategy to ensure correctness	Number of interviewees with the answer
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 the beta versions of software to medical workers who can access the data and do the tests.	1
Collecting and maintaining a dataset of problematic images.	1

Table C.10: Strategies to ensure correctness by the numbers of interviewees with the answers

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 C.11 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 [37].

Strategy to ensure maintainability	Number of interviewees with the answer
Modular approach / thinking the software as reusable Lego bricks / 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 C.11: Strategies to ensure maintainability by the numbers of interviewees with the answers

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

Table C.12 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.

Threat to understandability	Number of interviewees with the answer
Not all users understand how to use some features of the software.	2
The team has no dedicated user experience (UX) designer.	1
The software does not make some important indicators 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 C.12: Threats to understandability by the numbers of interviewees with the answers

Table C.13 shows the strategies to ensure *understandability* by the numbers of interviewees with the answers.

Strategy to ensure understandability	Number of interviewees with the answer
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 C.13: Strategies to ensure understandability by the numbers of interviewees with the answers

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

Table C.14 shows the strategies to ensure *usability* by the numbers of interviewees with the answers.

Strategy to ensure usability	Number of interviewees with the answer
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 C.14: Strategies to ensure usability by the numbers of interviewees with the answers

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 C.15 shows the threats to *reproducibility* by the numbers of interviewees with the answers.

Threat to reproducibility	Number of interviewees with the answer
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
When reverse-engineering how manufacturers create medical images, the team may misinterpret it.	1

Table C.15: Threats to reproducibility by the numbers of interviewees with the answers

Table C.16 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*.

Strategy to ensure reproducibility	Number of interviewees with the answer
Regression tests / unit tests / having good tests.	6
Making code, data, and documentation available / making the software open-source / using open-source libraries.	5
Running same tests on all platforms.	1
A dockerized version of the software, insulating it from the operating system 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
Benchmark the software against other software with similar purposes.	1

Table C.16: Strategies to ensure reproducibility by the numbers of interviewees with the answers

Appendix D

Ethics Approval

appendix here