The

STATE OF PRACTICE FOR MEDICAL IMAGING SOFTWARE

ASSESSING THE CURRENT STATE OF THE PRACTICE FOR MEDICAL IMAGING

SOFTWARE

redundant

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: ix, 114

Abstract

We present a method to assess the state of the practice for software in Scientific Computing (SC) domains and apply the method to the Medical Imaging (MI) domain. This method guided us to select 29 MI software projects from 48 candidates, assess nine software qualities by answering 103 questions for each software, and interview eight of the 29 development teams. The results helped us with revealing the current status of MI software development, such as the merits, drawbacks, and pain points during the development process and the software qualities. Based on our findings, we suggested guidelines and recommendations for future SC software development.

Keywords: Medical Imaging, Scientific Computing, software engineering, software quality, Analytic Hierarchy Process, developer interview

Acknowledgments

acknowledgements here

Abbreviations and Acronyms

symbol	description
2D	Two-Dimensional
3D	Three-Dimensional
AHP	Analytic Hierarchy Process
API	application programming interface
CI/CD	continuous integration and continuous delivery
L	Linux
LOC	lines of code
M	macOS
MI	Medical Imaging
MTTF	mean time to failure
NOC	number of contributors
MVC	Model, View, and Controller
OS	operating system
OSS	Open Source software
PACS	Picture Archiving and Communication System
Rlsd	release date
SC	Scientific computing
scc	Sloc Cloc and Code
VM	virtual machine
V	Visualization
W	Windows

Handwritten annotation: Capitalize the letter that is part of the acronym. Comment applies throughout this Table

Contents

Chapter 1

Introduction

This report 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] [91]. MI is an essential part of collecting accurate information during clinical diagnosis [95]. MI ~~computing and~~ software aim to visualize and process medical images and produce clinically meaningful information [90].

We define SC as "the use of computer tools to analyze or simulate mathematical models of real world systems of engineering or scientific importance so that we can better understand and predict the system's behaviour" [77]. Software development in SC depends on ~~the~~ knowledge of three areas: engineering or ~~science~~ domain knowledge, mathematical algorithm knowledge, and computational algorithm knowledge. 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

[Handwritten margin notes:]
Introduce SC & the measurement methodology first then introduce MI.
First introduce SC, then add that this is an important class of software that isn't always necessarily developed using modern software development techniques, tools & methods. Therefore, we ...develop a methodology for assessing the state of the practise practise for SC soft... We apply this process to MI soft.
include a citation for this
three field of
scientific
citations.

lists our research questions, and Section 1.3 ~~explains the~~ present ~~domain analysis of~~ scope of the MI software in our research.

~~scope of our research~~.

1.1 Motivation

Most scientists think developing and using SC software play a significant roles in their research [32]. They spend a substantial proportion of their working hours on SC software development [32] [65], and this proportion of time has increased over the years [32]. *from*

Developing SC software requires solid knowledge in specific domains [92]. Many ~~of~~ *developer* ~~them~~ learn software engineering skills by themselves or from their peers, instead of proper training [32]. Hannay et al. [32] *observe* ~~also pointed out~~ that many scientists showed ignorance and indifference to standard software engineering concepts. According to a survey by Prabhu et al. [65], 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 [92].

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. We build and refine our methodology, based on our previous work in scientific domains such as oceanography [80], mesh generation [84], geographic information systems [83], psychology [81] and seismology [82].

We want to understand whether problems like those mentioned above occur in all SC domains, or whether the state of the practice varies between domains.

1.2 Research Questions

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

[Handwritten annotations: "Why. aren't these questions in the same order and with the same wording as in Peter's report? We should standardize to use the same order & wording as Peter." "(So that we can more easily compare the two different domains.)" "Define pain points" with a checkmark]

RQ1. What are the pain points for developers working on MI software projects? What solutions the developers have tried to address the pain points?

RQ2. What *have* artifacts the projects generated?

RQ3. What role does documentation play in the projects? What are the developers' attitudes toward it?

RQ4. What principles, processes, methodologies, and tools the projects used?

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 the 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 [90], 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. [49] 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 (TK) libraries VTK [76] and ITK [57], developers build software with Visualization and Analysis functions; Picture Archiving and Communication System (PACS) helps users to economically store and conveniently access images [15]. 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.
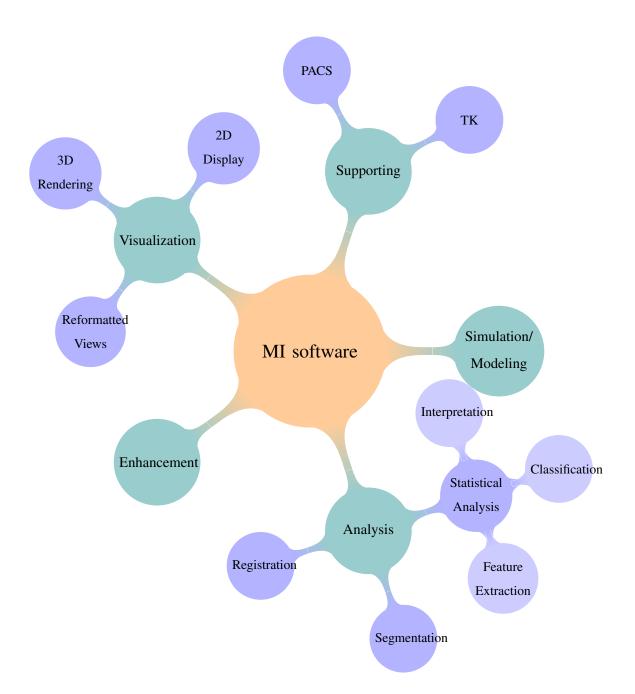
Figure 1.1: Major functions of MI software

Chapter 2

Background

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 Analytic Hierarchy Process (AHP) (Section 2.3).

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 A, we need the definitions of qualities and the AHP.

2.1 Software Categories

[handwritten note: add a sentence to distinguish categories from the Scope discussion]

We target specific software categories to narrow down the scope when selecting software packages for measuring. In this section, we discuss three common software categories: 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 [18]. 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 [66]. 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" [27]. 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 [27]. 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 [92].

2.2   Software Quality Definitions

This section lists the definitions of 10 software qualities, which are from Smith et al. [78]. 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 A. 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 [41] [51].

- **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 [25].

- **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) [60] [25].

- **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 [24] [11].

9

- **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" [42] [43].

- **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 [36] [11].

- **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" [47].

- **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" [40].

- **Visibility/Transparency** The extent to which all of the steps of a software development process and the current status of it are conveyed clearly [24].

- **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" [9].

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 [86]. AHP organizes multiple criteria factors in a hierarchical structure and uses pairwise comparisons between alternatives to calculate relative ratios [71].

When Comparing

For a project with m criteria, we can use an $m \times m$ 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 $1/A_{ij}$ [71],

- $A_{ij} = 1$ 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 [71],

$$Aw = \lambda_{max}w, \qquad (2.1)$$

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

In this project, w is approximated with the classic *mean of normalized values* approach [39],

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

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 for each of the m criteria. The software are evaluated with the grading template in Appendix A and a subjective score from one

11

to ten is given for each quality for each package. To convert the results into pair-wise scores, we apply the following steps: for each quality, for a pair of packages i and j, such that $score_i >= score_j$, the pairwise comparison result of i versus j is converted from $min((score_i - score_j) + 1, 9)$.

I asked in the last set of comments for an explanation for this equation. Why is the equation used?

Chapter 3

Methodology

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

Our method involves several steps, shown as follows:

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, 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 [79] 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 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 needs [79].

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 [79].

3.2.2 Filter the Software List

The goal is to build a software list with a length of about 30 [79].

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 try to apply them in the following priority order:

1. The functions and purpose 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.

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 [79].

3. The age of 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 [79]. 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 the top 10 software in the domain. Then, we cross-compare the two lists and discuss the commons and

differences. In addition, we ask them to vet our filtered list. They provide opinions on whether the list is reasonable to them. We also 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 103 questions that we use for grading software products. Figure 3.1 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, 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.

16

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 3.1: Grading template example

- **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 specifica-

17

tions, 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, 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 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 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, 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 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 display outputs in the format of web pages [26]; the other one is Sloc Cloc and Code (as known as *scc*) [12], 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, 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 VM 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 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.

3.4.2 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 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. We operate the invitation according to our ethics approval, such as the one in Appendix D. 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 1.1. We also included Dr. Michael Noseworthy 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 [10] [13] [28], online articles related to the domain [22] [33] [59], forum discussions related to the domain [72], etc. Appendix B shows all the 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 ones 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 3.1). We compared two lists and found six common ones. We did not include the rest in our final list, because their major function was not Visualization.

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 3.1: Top software by the MI domain experts

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 *MRIcron*, we chose *MRIcroGL*, because *MRIcron* development had moved to *MRIcroGL* [69].

After vetting our filtered list, the domain experts believed it was reasonable and did not identify any problem. Thus, as shown in Appendix B, 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 D). Going through the interview process in Section 3.4, we contacted all of the 29 teams. As a result, developers/architects from 8 teams have participated in our interviews so far.

Chapter 4

Measurement Results

As discussed in Section 3.5, we applied our method to the MI domain. 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.

Table 4.1 shows the 29 software packages that we measured. We found the initial release dates (Rlsd) in their documents for most projects and marked the rest two 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) 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 [48]	1998	2020-08	X	100	501451	X	X	X	
Ginkgo CADx [93]	2010	2019-05		3	257144	X	X	X	
XMedCon [62]	2000	2020-08		2	96767	X	X	X	
Weasis [68]	2010	2020-08		8	123272	X	X	X	
MRIcroGL [50]	2015	2020-08		2	8493	X	X	X	
SMILI [14]	2014	2020-06		9	62626	X	X	X	
ImageJ [70]	1997	2020-08	X	18	9681	X	X	X	
Fiji [75]	2011	2020-08	X	55	10833	X	X	X	
DicomBrowser [7]	2012	2020-08		3	5505	X	X	X	
3DimViewer [85]	?	2020-03	X	3	178065	X	X		
Horos [35]	?	2020-04		21	561617		X		
OsiriX Lite [74]	2004	2019-11		9	544304		X		
dwv [56]	2012	2020-09		22	47815	X	X	X	X
Drishti [53]	2012	2020-08		1	268168	X	X	X	
BioImage Suite Web [64]	2018	2020-10	X	13	139699	X	X	X	X
OHIF Viewer [96]	2015	2020-10		76	63951	X	X	X	X
Slice:Drop [29]	2012	2020-04		3	19020	X	X	X	X
GATE [44]	2011	2020-10		45	207122		X	X	
ITK-SNAP [94]	2006	2020-06	X	13	88530	X	X	X	
ParaView [3]	2002	2020-10	X	100	886326	X	X	X	X
MatrixUser [54]	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 [23]	2009	2020-11		21	148924	X	X	X	
dicompyler [63]	2009	2020-01		2	15941	X	X		
MicroView [38]	2015	2020-08		2	27470	X	X	X	
Papaya [67]	2012	2019-05		9	71831	X	X	X	
AMIDE [55]	2006	2017-01		4	102827	X	X	X	
Gwyddion [61]	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.
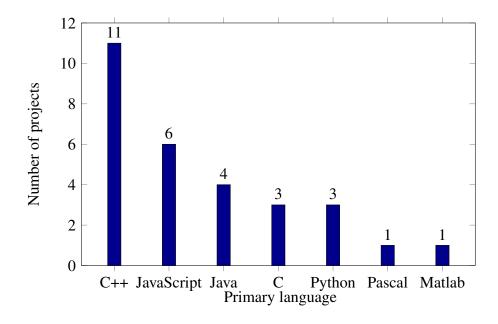
Figure 4.1: Primary languages versus number of projects using them

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*.
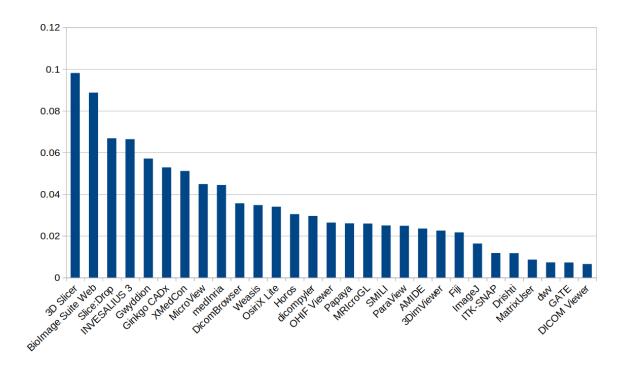
Figure 4.2: AHP installability scores

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 auto-

mated 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 [31]. 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.
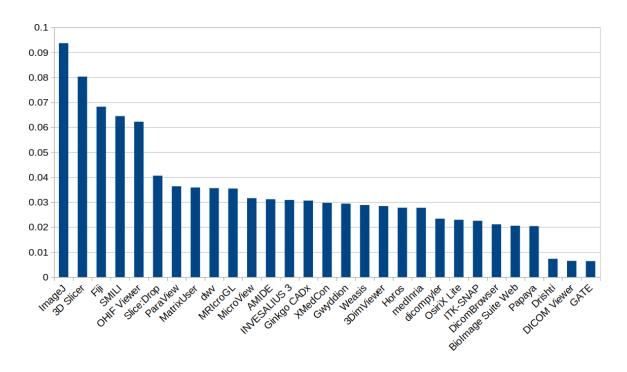
Figure 4.4: AHP surface reliability scores

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

33

response several times.

4.4   Surface Robustness

Figure 4.5 presents the scores for *surface robustness*.
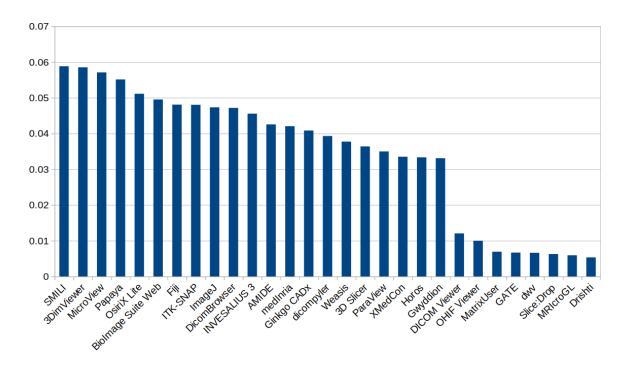
Figure 4.5: AHP surface robustness scores

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 unex-

pected/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*.
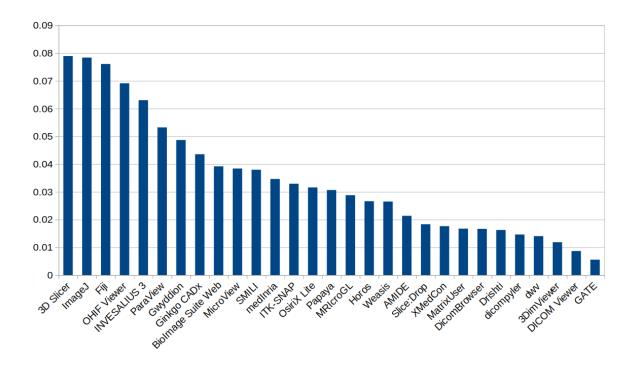
Figure 4.6: AHP surface usability scores

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*.
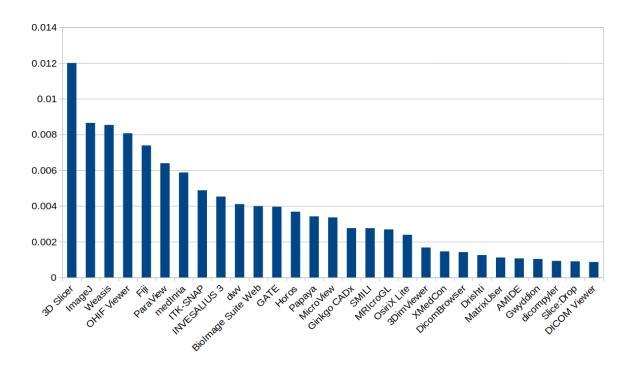
Figure 4.7: AHP maintainability scores

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*.
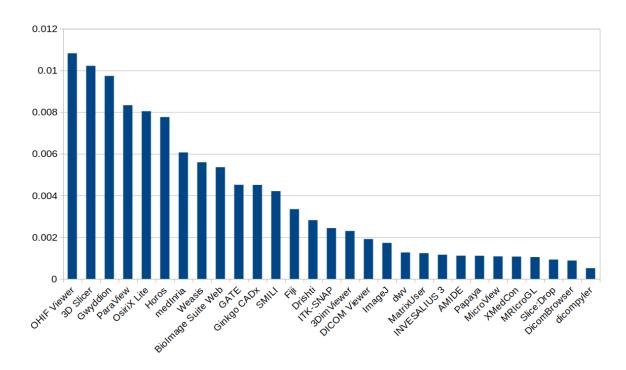
Figure 4.8: AHP reusability scores

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*.
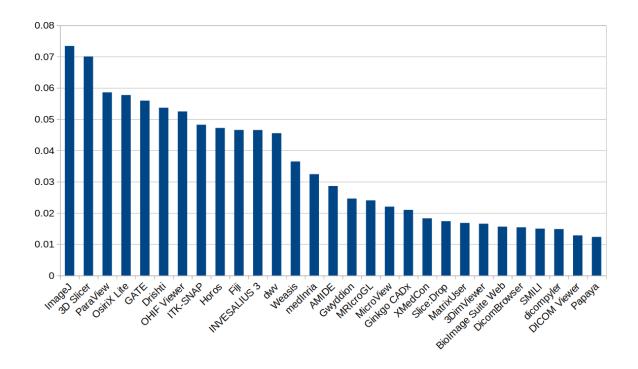
Figure 4.10: AHP surface understandability scores

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*.

41

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.
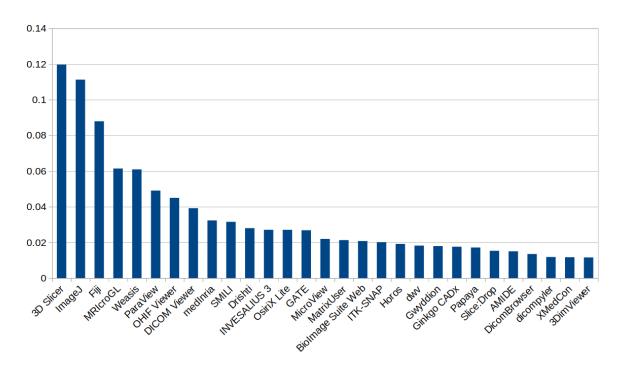
Figure 4.11: AHP visibility/transparency scores

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 [68]			X	X
MRIcroGL [50]				X
SMILI [14]				X
ImageJ [70]	X	X	X	X
Fiji [75]	X	X	X	
Horos [35]				X
OsiriX Lite [74]				X
dwv [56]				X
Drishti [53]				X
BioImage Suite Web			X	
OHIF Viewer [96]			X	X
GATE [44]				X
ITK-SNAP [94]				X
ParaView [3]		X		
MatrixUser [54]				X
DICOM Viewer [2]			X	X
INVESALIUS 3 [5]				X
medInria [23]			X	X
MicroView [38]				X
Gwyddion [61]				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.
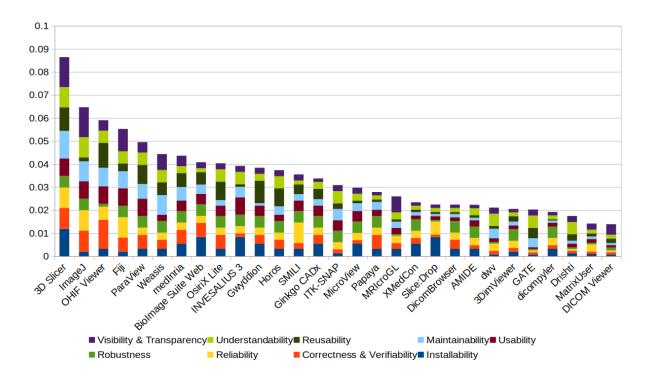
Figure 4.12: Overall AHP scores for all 9 software qualities

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

We applied our method to the MI domain (Section 3.5) and summarize some answers from the interviews with developers in this section. We only discuss the answers that we think are the most exciting and essential in this section, and summarize the rest in Appendix C.

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 can be 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. 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.

Grp	Obstacle	Num ans.	
		current	past
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
Oths	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 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 5.1.1, 5.1.2, and 5.1.3 include further discussion about the three major groups of pain points and their solutions.

5.1.1 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:

- 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.

48

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   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, performance, 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 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 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 in Table 5.3.

	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 5.3: Advantages and disadvantages of native application and web application

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;

51

- 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;

- 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:

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 5.4 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	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 preferable than documentation	2
Users help each other by referring to documentation	1

Table 5.4: Opinions on documentation by the numbers of interviewees with the answers

Table 5.5 lists some of the tools and methods mentioned by the interviewees, which they used for documentation.

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

5.3   Contribution Management and Project Management

We tried to understand how the teams managed the contributions and their projects by asking the 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?

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 5.6 shows the number of times the interviewees mentioned the methods of receiving contributions.

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 5.6: 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 5.7 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	Num ans.
Undefined/self-directed	3
Similar to Agile	2
Similar to Waterfall	1
Agile	1
Waterfall	1

Table 5.7: Software development models by the numbers of interviewees with the answers

Some interviewees mentioned the project management tools they used, which are in Table 5.8. 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	Num ans.
GitHub	3
Issue trackers	1
Bug trackers	1
Jira	1
Wiki page	1
Google Doc	1
Confluence	1

Table 5.8: 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?". We concluded that a yes-or-no question is not very informative here. 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.

5.4 Discussions on Software Qualities

Questions 15, 16, 17, 18, and 20 are about software qualities *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.

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

Threat to correctness	Num ans.
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 5.9: Threats to correctness by the numbers of interviewees with the answers

Table 5.10 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	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 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 5.10: Strategies to ensure correctness by the numbers of interviewees with the answers

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 5.11 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 [46].

Strategy to ensure maintainability	Num ans.
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 5.11: Strategies to ensure maintainability by the numbers of interviewees with the answers

5.4.3 Understandability

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

Table 5.12 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	Num ans.
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 5.12: Threats to understandability by the numbers of interviewees with the answers

Table 5.13 shows the strategies to ensure *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 5.13: Strategies to ensure understandability by the numbers of interviewees with the answers

5.4.4 Usability

Q17. What, if any, actions were taken to address usability issues?

Table 5.14 shows the strategies to ensure *usability* 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 5.14: Strategies to ensure usability by the numbers of interviewees with the answers

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 5.15 shows the threats to *reproducibility* 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
When reverse-engineering how manufacturers create medical images, the team may misinterpret it.	1

Table 5.15: Threats to reproducibility by the numbers of interviewees with the answers

Table 5.16 shows the strategies to ensure *reproducibility* by the numbers of intervie-wees 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	Num ans.
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 5.16: Strategies to ensure reproducibility by the numbers of interviewees with the answers

Chapter 6

Answers to Research Questions

This section answers our research questions in Section 1.2. The answers are based on our quality measurements and developer interviews. Sections 6.1, 6.2, 6.3, 6.4, 6.5, and 6.6 summarize the answers to the six questions respectively. Section 6.7 lists the threats to the validity of our research.

6.1 Pain Points and Solutions

RQ1. What are the pain points for developers working on MI software projects? What solutions have the developers tried to address the pain points?

We answer this question primarily by interviewing the developers. As described in Section 5.1, the top three pain points are as follows,

- 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.

Section 5.1 also lists potential and proven solutions from the developers.

6.2   Artifacts in the Projects

RQ2. What artifacts the projects generated?

We answer this question by searching and scanning the source code and documentation of the projects.

Part of the answer is in Section 4. Table 4.3 shows which projects contained a project plan, development manual, or API documentation, and Table 4.4 lists the projects with a documented development process, project status, development environment, or release notes.

Table 6.1 shows the frequency of some artifacts in the 29 MI projects.

Artifact	Number of projects
Source code	29
Version Control	29
README	29
License	28
Bug tracker	28
Change request	28
User Manual	22
Release notes	22
Build file	18
Tutorials	18
Makefile	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 6.1: Artifacts by their frequency in the 29 MI projects

6.3 Documentation in the Projects

RQ3. What role does documentation play in the projects? What are the developers' attitudes toward it?

We answer this question by interviewing the developers. As shown in Section 5.2, most 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 Principles, Processes, Methodologies, and Tools in the Projects

RQ4. What principles, processes, methodologies, and tools the projects used?

We answer this question by measuring the qualities and interviewing the developers.

Table 4.3, 4.4, and 6.1 reveal part of the answer, such as which and how many teams used a project plan, API documentation, release notes, version control, etc.

We summarize the other part of the answer from the interviews. Sections 5.2 and 5.3 list some principles, processes, methodologies, and tools for documentation, contribution management, and project management. As shown in Section 5.4, the developers also shared some of their principles, processes, methodologies, and tools to improve *correctness*, *maintainability*, *understandability*, *usability*, and *reproducibility*.

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?

We answer this question by measuring the qualities and interviewing the developers.

Section 4 contains our measurement results for *Installability*, *Correctness & Verifiability*, *Reliability*, *Robustness*, *Usability*, *Maintainability*, *Reusability*, *Understandability*, and *Visibility/Transparency*. We applied (surface) measurements to the nine qualities, and hope that the grading scores represent their current status. We believe that our assessments also revealed some of the development teams' practices addressing the qualities.

Section 5.4 shows findings from the interviews for *correctness*, *maintainability*, *understandability*, *usability*, and *reproducibility*. The interviewees expressed their thoughts on these five qualities. We discussed the current or past threats to them, and found out what actions they believed would improve the qualities.

6.6 Our Ranking versus the Community Ratings

RQ6. How does the software quality ranking generated by our methods compare with the ratings from the community?

We answer this question by grading the qualities of the software, then comparing the ranking with the community ratings on GitHub, such as GitHub stars, number of forks, and number of people watching the projects.

Table 6.2 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.

Generally speaking, most of the top-ranking MI software projects also received greater attention and popularity on GitHub. 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.

Software	Our ranking	Stars/yr	Watch/yr	Forks/yr
3D Slicer	1	284.25	18.75	128.25
ImageJ	2	83.58	9.37	30.00
OHIF Viewer	3	277.04	19.30	223.83
Fiji	4	44.20	4.98	20.68
ParaView	5	66.76	7.11	28.36
Weasis	6	36.00	5.10	18.90
medInria	7	7.04	3.35	6.35
BioImage Suite Web	8	17.85	4.62	6.77
OsiriX Lite	9	34.14	8.62	23.77
INVESALIUS 3	10	39.66	4.11	17.37
Gwyddion	11	n/a	n/a	n/a
Horos	12	48.62	9.23	18.00
SMILI	13	3.04	0.91	1.52
Ginkgo CADx	14	18.83	4.43	5.72
ITK-SNAP	15	9.10	0.97	3.59
MicroView	16	1.48	0.82	0.82
Papaya	17	44.70	4.84	19.71
MRIcroGL	18	23.63	3.38	3.38
XMedCon	19	n/a	n/a	n/a
Slice:Drop	20	9.73	2.16	4.97
DicomBrowser	21	n/a	n/a	n/a
AMIDE	22	n/a	n/a	n/a
dwv	23	123.56	11.59	51.36
3DimViewer	24	n/a	n/a	n/a
GATE	25	18.90	5.79	25.79
dicompyler	26	35.18	4.77	13.64
Drishti	27	16.04	3.78	4.43
MatrixUser	28	2.00	0.33	0.33
DICOM Viewer	29	42.86	5.71	8.57

Table 6.2: Software ranking versus GitHub metrics

6.7   Threats 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.

6. As mentioned in Section 6.6, we gave *dwv* much lower scores than its GitHub popularity. We might underestimate its rank due to uncommon technical issues.

Chapter 7

Recommendations

This section presents our recommendations on SC software development. We give these suggestions to the development in all SC domains, unless specifically mentioning that some guidelines are 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 to tackle 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**

    - 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**

  - – test-driven development with unit tests, integration tests, nightly tests, etc.

  - – two stage development process with stable release & nightly builds;

  - – CI/CD;

  - – documents of requirements specifications and theory manuals.

- **Reliability**

  - – test-driven development with unit tests, integration tests, nightly tests, etc.

  - – two stage development process with stable release & nightly builds;

  - – descriptive error messages.

- **Robustness**

  - – designing with exceptions and make the software failures elegant;

  - – descriptive error messages.

- **Usability**

  - – 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**

  – using GitHub;

  – modular approach;

  – documentation for developers, such as project plan, developer's manual, and API documentation.

- **Reusability**

  – modular approach;

  – API documentation.

- **Understandability**

  – modular approach;

  – consistent codding style;

  – clear comments;

  – description of used algorithms;

  – documentation;

  – communication between developers and users via GitHub issues, mailing lists, forums etc.

  – graphical user interface.

- **Visibility/Transparency**

  – documents for the development process, project status, development environment, and release notes.

- **Reproducibility**

  - test-driven development with unit tests, integration tests, nightly tests, etc.

  - 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 to 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 by 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 the bugs show.

Identifying and fixing bugs can cause 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 difficulty to maintain. 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 fix bugs and design problems in the past. The software qualities may also 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 to 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 with harnessing 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 back-end 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, good documents such as 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 these data, which can form a good dataset 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 their voluntary supports for testing. Ultimately, perhaps the community can establish a nonprofit organization as a third-party, which maintains large datasets, tests OSS in the domain, and protects privacy.

Chapter 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, that we list in Section 1.2.

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.

With the measurement and interview results in Section 4 and 5, we reveal some current status of SC software development and qualities in the MI domain. The interview results also show some merits, drawbacks, and pain points within the development process. With these findings, we answer our six research questions in Section 6. In addition, Section 7 presents our recommendations on SC software development, such as the ones on improving software qualities and easing pain points during development.

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

- we can further automate the measurements on the grading temple in Appendix A. For example, with automation scripts and a GitHub API, we may save significant time on retrieving the GitHub metrics;

- 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. For example, in question 14, "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?"

- we can better organize the interview questions. Since we use audio conversion tools to transcribe the answers, we should aim to 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.

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] F. Benureau and N. Rougier. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *ArXiv e-prints*, August 2017.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

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

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

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

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

[25] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

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

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

[28] 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.

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

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

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

[32] 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.

[33] 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].

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

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

[36] IEEE. Ieee standard glossary of software engineering terminology. Standard, IEEE, 1991.

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

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

[39] 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.

[40] ISO. Iec 9126-1: Software engineering-product quality-part 1: Quality model. *Geneva, Switzerland: International Organization for Standardization*, 21, 2001.

[41] ISO/IEC. Systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models. Standard, International Organization for Standardization, Mar 2011.

[42] ISO/TR. Ergonomics of human-system interaction — usability methods supporting human-centred design. Standard, International Organization for Standardization, June 2002.

[43] ISO/TR. Ergonomics of human-system interaction — part 11: Usability: Definitions and concepts. Standard, International Organization for Standardization, March 2018.

[44] 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.

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

[46] 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.

[47] Panagiotis Kalagiakos. The non-technical factors of reusability. In *Proceedings of the 29th Conference on EUROMICRO*, page 124. IEEE Computer Society, 2003.

[48] 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.

[49] 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.

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

[51] Jörg Lenhard, Simon Harrer, and Guido Wirtz. Measuring the installability of service orchestrations using the square method. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 118–125. IEEE, 2013.

[52] 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.

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

[54] 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.

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

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

[57] 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.

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

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

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

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

[62] 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.

[63] 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.

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

[65] 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.

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

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

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

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

[70] 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.

[71] 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.

[72] 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].

[73] 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].

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

[75] 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.

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

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

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

[79] 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.

[80] Spencer Smith, Yue Sun, and Jacques Carette. State of the practice for developing oceanographic software. Technical report, 04 2015. CAS-15-02-SS. McMaster University, Department of Computing and Software.

[81] Spencer Smith, Yue Sun, and Jacques Carette. Statistical software for psychology: Comparing development practices between cran and other communities, 2018.

[82] Spencer Smith, Zheng Zeng, and Jacques Carette. Seismology software: state of the practice. *Journal of Seismology*, 22, 05 2018.

[83] W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of the practice for gis software, 2018.

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

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

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

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

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

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

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

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

[92] 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.

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

[94] 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.

[95] 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.

[96] 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 ex-

tensible 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

Table A.1: Measurement Template

Summary Information
Software name? (string)
URL? (URL)
Affiliation (institution(s)) (string or N/A)
Software purpose (string)
Number of developers (all developers that have contributed at least one commit to the project) (use repo commit logs) (number)
How is the project funded? (unfunded, unclear, funded∗) where ∗ requires a string to say the source of funding
Initial release date? (date)
Last commit date? (date)
Status? (alive is defined as presence of commits in the last 18 months) (alive, dead, unclear)
License? (GNU GPL, BSD, MIT, terms of use, trial, none, unclear, other∗) ∗ given via a string
Platforms? (set of Windows, Linux, OS X, Android, other∗) ∗ given via string
Software Category? The concept category includes software that does not have an officially released version. Public software has a released version in the public domain. Private software has a released version available to authorized users only. (concept, public, private)
Development model? (open source, freeware, commercial, unclear)
Publications about the software? Refers to publications that have used or mentioned the software. (number or unknown)
Source code URL? (set of url, n/a, unclear)
Programming language(s)? (set of FORTRAN, Matlab, C, C++, Java, R, Ruby, Python, Cython, BASIC, Pascal, IDL, unclear, other∗) ∗ given via string
Is there evidence that performance was considered? Performance refers to either speed, storage, or throughput. (yes∗, no)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Installability (Measured via installation on a virtual machine.)

Are there installation instructions? (yes, no)

Are the installation instructions in one place? Place referring to a single document or web-page. (yes, no, n/a)

Are the installation instructions linear? Linear meaning progressing in a single series of steps. (yes, no, n/a)

Are the instructions written as if the person doing the installation has none of the dependent packages installed? (yes, no, unclear)

Are compatible operating system versions listed? (yes, no)

Is there something in place to automate the installation (makefile, script, installer, etc)? (yes∗, no)

If the software installation broke, was a descriptive error message displayed? (yes, no, n/a)

Is there a specified way to validate the installation? (yes∗, no)

How many steps were involved in the installation? (Includes manual steps like unzipping files) Specify OS. (number, OS)

What OS was used for the installation? (Windows, Linux, OS X, Android, other∗)
∗given via string

How many extra software packages need to be installed before or during installation? (number)

Are required package versions listed? (yes, no, n/a)

Are there instructions for the installation of required packages / dependencies? (yes, no, n/a)

Run uninstall, if available. Were any obvious problems caused? (yes∗ , no, unavail)

Overall impression? (1 .. 10)

Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Correctness and Verifiability
Any reference to the requirements specifications of the program or theory manuals? (yes∗ , no, unclear)
What tools or techniques are used to build confidence of correctness? (literate programming, automated testing, symbolic execution, model checking, assertions used in the code, Sphinx, Doxygen, Javadoc, confluence, unclear, other∗) ∗ given via string
If there is a getting started tutorial? (yes, no)
Are the tutorial instructions linear? (yes, no, n/a)
Does the getting started tutorial provide an expected output? (yes, no∗, n/a)
Does your tutorial output match the expected output? (yes, no, n/a)
Are unit tests available? (yes, no, unclear)
Is there evidence of continuous integration? (for example mentioned in documentation, Jenkins, Travis CI, Bamboo, other) (yes∗, no, unclear)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Surface Reliability
Did the software "break" during installation? (yes∗ , no)
If the software installation broke, was the installation instance recoverable? (yes, no, n/a)
Did the software "break" during the initial tutorial testing? (yes∗, no, n/a)
If the tutorial testing broke, was a descriptive error message displayed? (yes, no, n/a)
If the tutorial testing broke, was the tutorial testing instance recoverable? (yes, no, n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Robustness
Does the software handle unexpected/unanticipated input (like data of the wrong type, empty input, missing files or links) reasonably? (a reasonable response can include an appropriate error message.) (yes, no∗)
For any plain text input files, if all new lines are replaced with new lines and carriage returns, will the software handle this gracefully? (yes, no∗, n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Usability
Is there a getting started tutorial? (yes, no)
Is there a user manual? (yes, no)
Are expected user characteristics documented? (yes, no)
What is the user support model? FAQ? User forum? E-mail address to direct questions? Etc. (string)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Maintainability

What is the current version number? (number)

Is there any information on how code is reviewed, or how to contribute? (yes∗, no)

Are artifacts available? (List every type of file that is not a code file – for examples please look at the 'Artifact Name' column of https://gitlab.cas.mcmaster.ca/SEforSC/se4sc/-/blob/git-svn/GradStudents/Olu/ResearchProposal/Artifacts_MiningV3.xlsx) (yes∗, no, unclear) ∗list via string

What issue tracking tool is employed? (set of Trac, JIRA, Redmine, e-mail, discussion board, sourceforge, google code, git, BitBucket, none, unclear, other∗) ∗ given via string

What is the percentage of identified issues that are closed? (percentage)

What percentage of code is comments? (percentage)

Which version control system is in use? (svn, cvs, git, github, unclear, other∗) ∗ given via string

Overall impression? (1 .. 10)

Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Reusability

How many code files are there? (number)

Is API documented? (yes, no, n/a)

Overall impression? (1 .. 10)

Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Understandability (Based on 10 random source files)
Consistent indentation and formatting style? (yes, no, n/a)
Explicit identification of a coding standard? (yes*, no, n/a)
Are the code identifiers consistent, distinctive, and meaningful? (yes, no* , n/a)
Are constants (other than 0 and 1) hard coded into the program? (yes, no* , n/a)
Comments are clear, indicate what is being done, not how? (yes, no* , n/a)
Is the name/URL of any algorithms used mentioned? (yes, no* , n/a)
Parameters are in the same order for all functions? (yes, no* , n/a)
Is code modularized? (yes, no* , n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Visibility/Transparency
Is the development process defined? If yes, what process is used. (yes*, no, n/a)
Are there any documents recording the development process and status? (yes*, no))
Is the development environment documented? (yes*, no)
Are there release notes? (yes*, no)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Raw Metrics (Measured via git_stats)
Number of text-based files. (number)
Number of binary files. (number)
Number of total lines in text-based files. (number)
Number of total lines added to text-based files. (number)
Number of total lines deleted from text-based files. (number)
Number of total commits. (number)
Numbers of commits by year in the last 5 years. (Count from as early as possible if the project is younger than 5 years.) (list of numbers)
Numbers of commits by month in the last 12 months. (list of numbers)
Raw Metrics (Measured via scc)
Number of text-based files. (number)
Number of total lines in text-based files. (number)
Number of code lines in text-based files. (number)
Number of comment lines in text-based files. (number)
Number of blank lines in text-based files. (number)
Repo Metrics (Measured via GitHub)
Number of people watching this repo. (number)
Number of stars. (number)
Number of forks. (number)
Number of open pull requests. (number)
Number of closed pull requests. (number)
Number of months on GitHub. (number)
Accessed date. (date)

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 Picture Archiving and Communication System (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 [10]	[13]	[28]	[22]	[33]	[59]	[72]
3D Slicer [48]	X	X	V			X	X	X		X
Ginkgo CADx [93]	X	X	V	X	X	X	X	X	X	
XMedCon [62]	X	X	V			X	X	X	X	
Weasis [68]	X	X	V			X	X	X	X	
MRIcroGL [50]	X	X	V			X	X	X		X
SMILI [14]	X	X	V				X	X	X	
ImageJ [70]	X	X	V			X	X	X		X
Fiji [75]	X	X	V							X
DicomBrowser [7]	X	X	V				X	X		

Software	Final list	Open-source	Sub-group	[10]	[13]	[28]	[22]	[33]	[59]	[72]
3DimViewer [85]	X	X	V	X			X	X		
Horos [35]	X	X	V	X	X					
OsiriX Lite [74]	X	X	V	X	X	X				
dwv [56]	X	X	V	X		X				
Drishti [53]	X	X	V	X						
BioImage Suite Web [64]	X	X	V			X				
OHIF Viewer [96]	X	X	V	Other source: [96]						
Slice:Drop [29]	X	X	V			X				
GATE [44]	X	X	V				X			
ITK-SNAP [94]	X	X	V							X
ParaView [3]	X	X	V							X
MatrixUser [54]	X	X	V					X		
DICOM Viewer [2]	X	X	V					X		
INVESALIUS 3 [5]	X	X	V					X		
medInria [23]	X	X	V			X				
dicompyler [63]	X	X	V						X	
MicroView [38]	X	X	V			X				
Papaya [67]	X	X	V				X	X	X	
AMIDE [55]	X	X	V				X	X	X	
Gwyddion [61]	X	X	V					X		
VTK [76]		X	TK							X
ITK [57]		X	TK							X

Software	Final list	Open-source	Sub-group	Source [10]	[13]	[28]	[22]	[33]	[59]	[72]
DCMTK [21]		X	TK					X	X	
XTK [30]		X	TK	Other source: it is used by slice:drop						
dcm4che [20]		X	TK	X			X	X		
cornerstone [19]		X	TK			X				
dcm2niix [52]		X	TK					X		
orthanc [45]		X	PACS	X				X		
Conquest [87]		X	PACS	X						
ClearCanvas [17]		X	PACS, V	X						
Open Dicom Viewer [88]		X	V			X			X	
MicroDicom [58]			V	X					X	
Aliza [4]			V				X		X	
JiveX [89]			V			X			X	
MIPAV [16]			V			X				
Oviyam [37]			V			X				
MeVisLab [34]			V							X
Sante DICOM Viewer Lite [73]			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	Num ans.
PHD	4
Master	3
Bachelor	0
Unspecified academic degree	2

Table C.1: Interviewees' highest academic degrees

Major	Num ans.
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	Num ans.
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.

Years	Num ans.
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.

Num mbrs	Num ans.
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	Num ans.
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	1) search results on Google Scholar; 2) number of new posts per year on slicer.org; 3) 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	Only counted the active users from several Universities.
ITK-SNAP	10,000 plus	Number of downloads.
MRIcroGL	No estimation	The top 1 downloaded software on the NITRC list.
Weasis	10,000 incl. one-time users	Number of profiles.
OHIF	About 5000	Some platforms integrated *OHIF*; hard to estimate.

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	Num ans.
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

Appendix D

Ethics Approval

This project received ethics clearance from the McMaster Research Ethics Board on February 20, 2021.

Project Title: AIMSS - State of the Practice

MREB#: 5219