Digging Deeper Into the State of the Practice for Domain Specific Research Software

Spencer Smith[1][0000−0002−0760−0987] and Peter Michalski[1]

McMaster University, 1280 Main Street West, Hamilton ON L8S 4K1, Canada
`smiths@mcmaster.ca`
`http://www.cas.mcmaster.ca/~smiths/`

Abstract. We have developed a methodology for assessing the state of the software development practice for a given research software domain. Our methodology prescribes the following steps: i) Identify the domain; ii) Identify a list of candidate software packages; iii) Filter the list to a length of about 30 packages; iv) Collect repository related data on each software package, like number of stars, number of open issues, number of lines of code; v) Fill in the measurement template (the template consists of 108 questions to assess 9 qualities (including the qualities of installability, usability and visibility)); vi) Rank the software using the Analytic Hierarchy Process (AHP); vii) Interview developers (the interview consists of 20 questions and takes about an hour); and, viii) Conduct a domain analysis. The collected data is analyzed by: i) comparing the ranking by best practices against the ranking by popularity; ii) comparing artifacts, tools and processes to current research software development guidelines; and, iii) exploring pain points. We estimate the time to complete an assessment for a given domain at 173 person hours. The method is illustrated via the example of Lattice Boltzmann Solvers, where we find that the top packages engaged in most of recommended best practices, but still show room for improvement with respect to providing API documentation, a roadmap, a code of conduct, programming style guide, uninstall instructions and continuous integration.

Keywords: research software · software quality · empirical measures · analytic hierarchy process · software artifacts · developer pain points · Lattice Boltzmann Method

1 Introduction

Research software is critical for tackling problems in areas as diverse as manufacturing, financial planning, environmental policy and medical diagnosis and treatment. However, developing reliable, reproducible, sustainable and fast research software to address new problems is challenging because of the complexity of the physical models and the nuances of floating point and parallel computation. The importance of research software and the difficulty with its development have prompted multiple researchers to investigate how developing this software differs from other classes of software. Previous studies have focused on surveying developers [6,12,14], developer interviews [9] and case studies [2,17]. A valuable source of information that has received less attention is the data in publicly available software repositories. We propose digging deeper into how well projects are applying best practices by using a new methodology that combines manual and automated techniques to extract repository-based information.

The surveys used in previous studies have tended to recruit participants from all domains of research software, possibly distinguishing them by programming language (for instance, R developers [14]), or by the role of the developers (for instance postdoctoral researchers [11]). Case studies, on the other hand, have focused on a few specific examples at a time, as is the nature of case studies. For our new methodology, we propose a scope between these two extremes. Rather than focus on all research software, or just a few examples, we will focus on one domain at a time. The practical reason for this scope is that digging deep takes time, making a broad scope infeasible. We have imposed a practical constraint of one person month of effort per domain.[1] Focusing on one domain at a time has more than just practical advantages. By restricting ourselves to a single domain we can bring domain knowledge and domain experts into the mix. The domain customized insight provided by the assessment has the potential to help a specific domain as they adopt and develop new practices. Moreover, measuring multiple different domains facilitates comparing and contrasting domain specific practices.

Our methodology is built around 10 research questions. Assuming that the domain has been identified (Section 2.1), the first question is:

RQ1: What software projects exist in the domain, with the constraint that the source code must be available for all identified projects? (Sections 2.2, 2.3)

We next wish to assess the representative software to determine how well they apply current software development best practices. At this point in the process, to remove potential user/developer bias, we will base our assessment only on publicly available artifacts, where artifacts are the documents, scripts and code that we find in a project's public repository. Example artifacts include requirements, specifications, user manuals, unit test cases, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code. Following best practices does not guarantee popularity, so we will also compare our ranking to how the user community itself ranks the identified projects.

RQ2: Which of the projects identified in RQ1 follow current best practices, based on evidence found by experimenting with the software and searching the artifacts available in each project's repository? (Sections 2.4)

RQ3: How similar is the list of top projects identified in RQ2 to the most popular projects, as viewed by the scientific community? (Section 3)

[1] A person month is considered to be 20 working days (4 weeks in a month, with 5 days of work per week) at 8 person hours per day, or $20 \cdot 8 = 160$ person hours.

To understand the state of the practice we wish to learn the frequency with which different artifacts appear, the types of development tools used and the methodologies used for software development. With this data, we can ask questions about how the domain software compares to other research software.

RQ4: How do domain projects compare to research software in general with respect to the artifacts present in their repositories? (Section 4)
RQ5: How do domain projects compare to research software in general with respect to the use of tools (Section 5) for:
RQ5.a development; and,
RQ5.b project management?
RQ6: How do domain projects compare to research software in general with respect to the processes used? (Section 6)

Only so much information can be gleaned by digging into software repos. To gain additional insight, we need to interview developers (Section 2.5) to learn:

RQ7: What are the pain points for developers working on domain software projects? (Section 7)
RQ8: How do the pain points of domain developers compare to the pain points for research software in general? (Section 7)

Our methodology answers the research question through inspecting repositories, using the Analytic Hierarch Process (AHP) for ranking software, interviewing developers and interacting with at least one domain expert. We leave the measurement of the performance, for instance using benchmarks, to other projects [8]. The current methodology updates the approach used in prior assessments of domains like Geographic Information Systems [22], Mesh Generators [21], Seismology software [24], and statistical software for psychology [23]. Initial tests of the new methodology have been done for medical image analysis software [3] and for Lattice Boltzmann Method (LBM) software [10]. The LBM example will be used throughout this paper to illustrate the steps in the methodology. The paper ends with a summary of potential threats to validity (Section 8) and our conclusions (Section 9).

2 Methodology

The assessment is conducted via the following steps. The steps depend on interaction with a Domain Expert partner, as discussed in Section 2.6.

1. Identify the domain of interest. (Section 2.1)
2. List candidate software packages for the domain. (Section 2.2)
3. Filter the software package list. (Section 2.3)
4. Gather the source code and documentation for each software package.
5. Collect quantitative measures from the project repositories. (Section 2.4)
6. Measure using the measurement template. (Section 2.4)

7. Use AHP to rank the software packages. (Section 2.4)
8. Interview the developers. (Section 2.5)
9. Domain analysis. (Section 2.7)
10. Analyze the results and answer the research questions. (Sections 3—7)

We estimate 173 hours to complete the assessment of a given domain [20], which is close our goal of 160 person hours. This estimate assumes the domain has been decided, the Domain Expert has been recruited, and the pre-existing template spreadsheet and AHP tool [20], will be utilized.

2.1 How to Identify the Domain?

To be applicable for the methodology described in this document, the chosen domain must have the following properties:

1. The domain must have well-defined and stable theoretical underpinning.
2. There must be a community of people studying the domain.
3. The software packages must have open source options.
4. A preliminary search, or discussion with experts, suggests that there will be around 30 or more candidate software packages.

2.2 How to Identify Candidate Software from the Domain?

The candidate software to answer RQ1 should be found through search engine queries, GitHub, swMATH and scholarly articles. The Domain Expert (Section 2.6) should also be engaged in selecting the candidate software. The following properties are considered when creating the list:

1. The software functionality must fall within the identified domain.
2. The source code must be viewable.
3. The repository based measures should be available.
4. The software must be complete.

2.3 How to Filter the Software List?

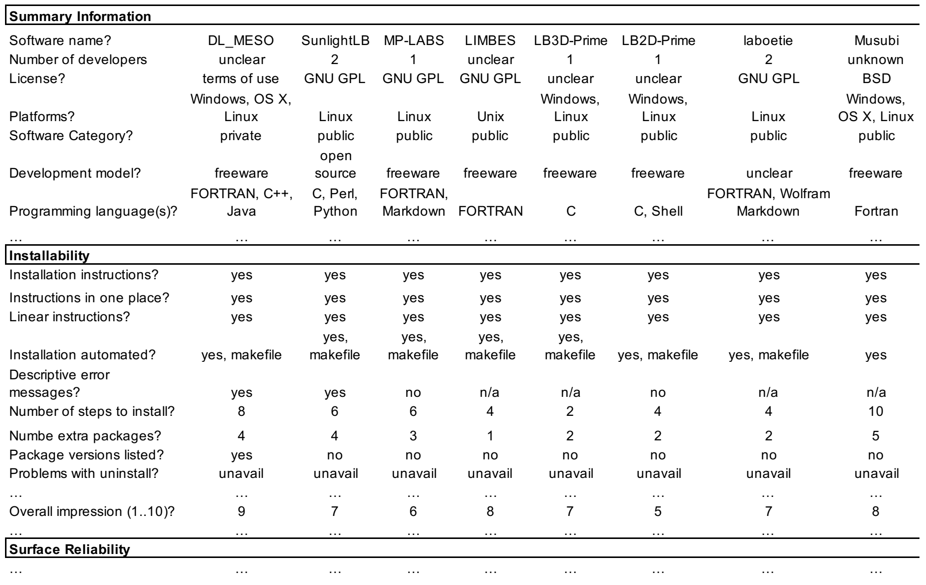
If the list of software is too long (over around 30 packages), then filters are applied in the priority order listed. Copies of both lists, along with the rationale for shortening the list, should be kept for traceability purposes.

1. Scope: Software is removed by narrowing what functionality is considered to be within the scope of the domain.
2. Usage: Software packages are eliminated if their installation procedure is missing or not clear and easy to follow.
3. Age: Older software packages (age being measured by the last date when a change was made) are eliminated, except in the cases where an older software package appears to be highly recommended and currently in use. (The Domain Expert should be consulted on this question as necessary.)

For the LBM example the initial list had 46 packages [10]. This list was filtered by scope, usage, and age to decrease the length to 24.

2.4 Quantitative Measures

We rank the projects by how well they follow best practices (RQ2) via a measurement template [20]. For each software package (each column in the template), we fill-in the rows. This process takes about 2 hours per package, with a cap of 4 hours. An excerpt of the template is shown in Figure 1. In keeping with scientific transparency, all data should be made available in a public repository.



Summary Information								
Software name?	DL_MESO	SunlightLB	MP-LABS	LIMBES	LB3D-Prime	LB2D-Prime	laboetie	Musubi
Number of developers	unclear	2	1	unclear	1	1	2	unknown
License?	terms of use	GNU GPL	GNU GPL	GNU GPL	unclear	unclear	GNU GPL	BSD
Platforms?	Windows, OS X, Linux	Linux	Linux	Unix	Windows, Linux	Windows, Linux	Linux	Windows, OS X, Linux
Software Category?	private	public	public	public	public	public	public	public
Development model?	freeware	open source	freeware	freeware	freeware	freeware	unclear	freeware
Programming language(s)?	FORTRAN, C++, Java	C, Perl, Python	FORTRAN, Markdown	FORTRAN	C	C, Shell	FORTRAN, Wolfram Markdown	Fortran
...	...	...	...	...	...	...	...	...
Installability								
Installation instructions?	yes	yes	yes	yes	yes	yes	yes	yes
Instructions in one place?	yes	yes	yes	yes	yes	yes	yes	yes
Linear instructions?	yes	yes	yes	yes	yes	yes	yes	yes
Installation automated?	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes, makefile	yes
Descriptive error messages?	yes	yes	no	n/a	n/a	no	n/a	n/a
Number of steps to install?	8	6	6	4	2	4	4	10
Numbe extra packages?	4	4	3	1	2	2	2	5
Package versions listed?	yes	no	no	no	no	no	no	no
Problems with uninstall?	unavail	unavail	unavail	unavail	unavail	unavail	unavail	unavail
...	...	...	...	...	...	...	...	...
Overall impression (1..10)?	9	7	6	8	7	5	7	8
...	...	...	...	...	...	...	...	...
Surface Reliability								
...	...	...	...	...	...	...	...	...

Fig. 1. Excerpt of the Top Sections of the Measurement Template

The full template [20] consists of 108 questions categorized under 9 qualities: (i) installability; (ii) correctness and verifiability; (iii) surface reliability; (iv) surface robustness; (v) surface usability; (vi) maintainability; (vii) reusability; (viii) surface understandability; and, (ix) visibility/transparency.

The questions were designed to be unambiguous, quantifiable and measurable with limited time and domain knowledge. The measures are grouped under headings for each quality, and one for summary information (Figure 1). The summary section provides general information, such as the software name, number of developers, etc. Several of the qualities use the word "surface". This is to highlight that, for these qualities, the best that we can do is a shallow measure. For instance, we do not conduct experiments to measure usability. Instead, we are looking for an indication that usability was considered by looking for cues in the documentation, such as getting started instructions, a user manual or documentation of expected user characteristics.

Tools are used to find some of the measurements, such as the number of files, number of lines of code (LOC), percentage of issues that are closed, etc. The tool GitStats is used to measure each software package's GitHub repository for the number of binary files, the number of added and deleted lines, and the number of commits over varying time intervals. The tool Sloc Cloc and Code (scc) is used to measure the number of text based files as well as the number of total, code, comment, and blank lines in each GitHub repository.

Virtual machines (VMs) are used to provide an optimal testing environments for each package [21] because with a fresh VM there are no worries about conflicts with existing libraries. Moreover, when the tests are complete the VM can be deleted, without any impact on the host operating system. The most significant advantage of using VMs is that every software install starts from a clean slate, which removes "works-on-my-computer" errors.

Once we have measured each package, we still need to rank them to answer RQ2. To do this, we used the Analytical Hierarchy Process (AHP), a decision-making technique that uses pair-wise comparisons to compare multiple options by multiple criteria [15]. In our work AHP performs a pairwise analysis between each of the 9 quality options for each of the (approximately) 30 software packages. This results in a matrix, which is used to generate an overall score for each software package for the given criteria [21].

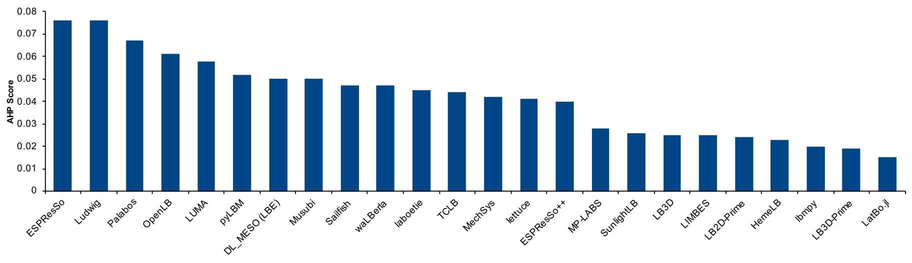


Fig. 2. AHP Overall Score

Figure 2 shows the overall ranking of the LBM software packages. In the absence of other information on priorities, the overall ranking was calculated by assuming an equal weighting between all qualities. The LBM data is available on-line.

2.5 Interview Developers

Several of the research question (RQ5, RQ6 and RQ7) require going beyond the quantitative data from the measurement template. To gain the required insight, we interview developers using a list of 20 questions [20]. The questions cover the background of the development teams, the interviewees, and the software itself. We ask the developers how they organize their projects and about their understanding 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 of, and the current situation for, documentation. A few questions are about specific software qualities, such as maintainability, usability, and reproducibility. The interviews are semi-structured based on the question list. Each interview should take about 1 hour.

The interviewees should follow standard ethics guidelines for consent, recording, and including participant details in the report. The interview process presented here was approved by the McMaster University Research Ethics Board under the application number MREB#: 5219. For LBM we were able to recruit 4 developers to participate in our study.

2.6 Interaction With Domain Expert

Our methodology relies on engaging a Domain Expert to vet the list of projects (RQ1) and the AHP ranking (RQ2). The Domain Expert is an important member of the assessment team. Pitfalls exist if non-experts attempt to acquire an authoritative list of software, or try to definitively rank software. Non-experts have the problem that they can only rely on information available on-line, which has the following drawbacks: i) the on-line resources could have false or inaccurate information; and, ii) the on-line resources could leave out relevant information that is so in-grained with experts that nobody thinks to explicitly record it. Domain experts may be recruited from academia or industry. The only requirements are knowledge of the domain and a willingness to be involved.

2.7 Domain Analysis

For the current methodology time constraints necessitate a shallow domain analysis. For each domain a table should be constructed that distinguishes the programs under study by their variabilities. In research software the variabilities are often assumptions. Table 1 shows the variabilities for LBM software [10].

3 Comparison to Community Ranking

To address RQ3 we need to compare the ranking by best practices to the community's ranking. Our best practices ranking comes from the AHP ranking (Section 2.4). We estimate the community's ranking by repository stars and watches. The comparison will provide insight on whether best practices are rewarded by popularity. However, inconsistencies between the AHP ranking and the community's ranking are inevitable for the following reasons: i) the overall quality ranking via AHP makes the unrealistic assumption of equal weighting between quality factors; ii) stars are known to not be a particularly good measure of popularity, because of how people use stars and because young projects have less time to accumulate stars [25]; iii) and, as for consumer products, there are more factors influencing popularity than just quality.

Name	Dim	Pll	Com	Rflx	MFl	Turb	CGE	OS
DL_MESO (LBE)	2, 3	MPI/OMP	Y	Y	Y	Y	Y	W, M, L
ESPResSo	1, 2, 3	CUDA/MPI	Y	Y	Y	Y	Y	M, L
ESPResSo++	1, 2, 3	MPI	Y	Y	Y	Y	Y	L
HemeLB	3	MPI	Y	Y	Y	Y	Y	L
laboetie	2, 3	MPI	Y	Y	Y	Y	Y	L
LatBo.jl	2, 3	-	Y	Y	Y	N	Y	L
LB2D-Prime	2	MPI	Y	Y	Y	Y	Y	W, L
LB3D	3	MPI	N	Y	Y	Y	Y	L
LB3D-Prime	3	MPI	Y	Y	Y	Y	Y	W, L
lbmpy	2, 3	CUDA	Y	Y	Y	Y	Y	L
lettuce	2, 3	CUDA	Y	Y	Y	Y	Y	W, M, L
LIMBES	2	OMP	Y	Y	N	N	Y	L
Ludwig	2, 3	MPI	Y	Y	Y	Y	Y	L
LUMA	2, 3	MPI	Y	Y	Y	Y	Y	W, M, L
MechSys	2, 3	-	Y	Y	Y	Y	Y	L
MP-LABS	2, 3	MPI/OMP	N	Y	Y	N	N	L
Musubi	2, 3	MPI	Y	Y	Y	Y	Y	W, L
OpenLB	1, 2, 3	MPI	Y	Y	Y	Y	Y	W, M, L
Palabos	2, 3	MPI	Y	Y	Y	Y	Y	W, L
pyLBM	1, 2, 3	MPI	Y	Y	N	Y	Y	W, M, L
Sailfish	2, 3	CUDA	Y	Y	Y	Y	Y	M, L
SunlightLB	3	-	Y	Y	N	N	Y	L
TCLB	2, 3	CUDA/MPI	Y	Y	Y	Y	Y	L
waLBerla	2, 3	MPI	Y	Y	Y	Y	Y	L

Table 1. Features of Software Packages (Dim for Dimension (1, 2, 3), Pll for Parallel (CUDA, MPI, OpenMP (OMP)), Com for Compressible (Yes or No), Rflx for Reflexive Boundary Condition (Yes or No), MFl for Multi-fluid (Yes or No), Turb for Turbulent (Yes or No), CGE for Complex Geometries (Yes or No), OS for Operating System (Windows (W), macOS (M), Linux (L)))

Table 2 compares the AHP ranking of the LBM package to their popularity in the research community. Nine packages do not use GitHub, so they do not have a measure of stars. Looking at the stars of the other 15 packages, we can observe a pattern where packages that have been highly ranked by our assessment tend to have more stars than lower ranked packages. The best ranked package by AHP (ESPResSo) has the second most stars, while the ninth ranked package (Sailfish) has the highest number of stars. Although the AHP ranking and the community popularity estimate are not perfect measures, they do suggest a correlation between best practices and popularity.

4 Comparison of Artifacts to Other Research Software

We answer RQ4 by comparing the artifacts that we observe to those observed and recommended for research software in general. While filling in the measurement

Name	Our Ranking	Repository Stars	Repository Star Rank	Repository Watches	Repository Watch Rank
ESPResSo	1	145	2	19	2
Ludwig	2	27	8	6	7
Palabos	3	34	6	GitLab	GitLab
OpenLB	4	No Git	No Git	No Git	No Git
LUMA	5	33	7	12	4
pyLBM	6	95	3	10	5
DL_MESO (LBE)	7	No Git	No Git	No Git	No Git
Musubi	8	No Git	No Git	No Git	No Git
Sailfish	9	186	1	41	1
waLBerla	10	20	9	GitLab	GitLab
laboetie	11	4	13	5	8
TCLB	12	95	3	16	3
...	...	...	...	...	...
LatBo.jl	24	17	10	8	6

Table 2. Excerpt from Repository Ranking Metrics [10]

template (Section 2.4), the domain software is examined for the presence of artifacts, which are then categorized by frequency as: common (more than 2/3 of projects), less common (between 1/3 and 2/3), and rare (less than 1/3). The observed frequency of artifacts should then be compared to the artifacts recommended by research software guidelines, as summarized in Table 3.

The majority of LBM generated artifacts (summarized below) correspond to general recommendations from research software developers. For LBM, a union of the three categories mostly corresponds to general research software recommendations (Table 3). Areas where LBM developers could improve include providing: API documentation, a roadmap, a code of conduct, programming style guide, and uninstall instructions.

Common: Developer List, Issue Tracker, Dependency List, Installation Guide, Theory Notes, Related Publications, Build Files, README File, License, Tutorial, Version Control

Less Common: Change Log, Design Doc., Functional Spec., Performance Info., Test Cases, User Manual

Rare: API Doc., Developer Manual, FAQ, Verification Plan, Video Guide, Requirements Spec.

5 Comparison of Tools to Other Research Software

Software tools are used to support the development, verification, maintenance, and evolution of software, software processes, and artifacts [4, p. 501]. Development tools support the development of end products, but do not become part of

	[27]	[16]	[1]	[29]	[18]	[7]	[26]	[5]	[13]
LICENSE	✓	✓	✓	✓	✓		✓	✓	✓
README		✓	✓	✓	✓		✓	✓	✓
CONTRIBUTING		✓	✓	✓	✓		✓	✓	✓
CITATION				✓				✓	✓
CHANGELOG		✓		✓	✓		✓		
INSTALL					✓		✓	✓	✓
Uninstall								✓	
Authors							✓	✓	✓
Getting started					✓		✓	✓	✓
User manual			✓				✓		
Tutorials							✓		
FAQ							✓	✓	✓
Dependency List			✓		✓			✓	
Issue Track		✓	✓		✓	✓	✓		✓
Version Control		✓	✓	✓	✓	✓	✓	✓	✓
Requirements		✓				✓			✓
Design		✓	✓		✓		✓	✓	✓
API Doc.					✓		✓	✓	✓
Build Scripts		✓		✓	✓	✓	✓		✓
Unit Tests	✓	✓	✓		✓	✓	✓	✓	✓
Test Plan		✓				✓			
Integ. Tests		✓	✓					✓	✓
System Tests		✓	✓			✓		✓	✓
Acceptance Tests		✓							
Regression Tests			✓			✓			✓
Code Style Guide		✓					✓	✓	✓
Release Info.		✓				✓	✓		
Product Roadmap						✓	✓	✓	
Code of Conduct							✓		
Acknowledgements							✓	✓	✓

Table 3. Commonly Recommended Artifacts in Software Development Guidelines

them, unlike dependencies that remain in the application once it is released [4, p. 506]. To answer RQ5 we summarize tools that are visible in the repositories and that are mentioned during the developer interviews. As an example, the tools found for LBM software packages are as follows:

Development Tools: Continuous Integration, Code Editors, Development Environments, Runtime Environments, Compilers, Unit Testing Tools, Correctness Verification Tools

Dependencies: Build Automation Tools, Technical Libraries, Domain Specific Libraries

Project Management Tools: Collaboration Tools, Email, Change Tracking Tools, Version Control Tools, Document Generation Tools

Once the data on tools is collected, the use of two specific tools should be compared to research software norms: version control and continuous integration. A little over 10 years ago version control was estimated to be used in only 50% of research software projects [12], but even at that time [12] noted an increase from previous usage levels. More recently, version control usage rates for active mesh generation, geographic information system and statistical software packages were close to 100% [19]. Almost every software guide cited in Table 3 includes the advice to use version control. For LBM packages 67% use version control (GitHub, GitLab or CVS). The high usage of version control tools in LBM software matches the trend in research software in general.

Continuous integration is rarely used in LBM (3 of 24 packages or 12.5%). This contrasts with the frequency with which continuous integration is recommended in research software development guidelines [1,5,26]. In this case, it seems likely that the recommendations are ahead of common practice.

6 Comparison of Processes to Other Research Software

The interview data on development processes is used to answer research question RQ6. This data should be contrasted with the development process used by research software in general. The literature suggests that scientific developers naturally use an agile philosophy [2,17], or an amethododical process [9]. Another point of comparison should be on the use of peer review, since peer review is frequently recommended [7,13,27].

The LBM example confirms an informal, agile-like, process. The development process is not explicitly indicated in the artifacts. However, during interviews one developer (ESPResSo) told us their non-rigorous development model is like a combination of agile and waterfall. Employing a loosely defined process makes sense for LBM software, given that the teams are generally small and self-contained. One of the developers (ESPResSo) also noted that they use an ad hoc peer review process.

7 Developer Pain Points

To answer RQ7, we ask developers about their pain points and compare their responses to the literature [28,14]. Pain points to watch for include: cross-platform compatibility, scope bloat, lack of user feedback, dependency management, data handling concerns, reproducibility, and software scope.

An example pain point noted for LBM is a lack of development time. A developer of pyLBM noted that their small development team did not have enough time to implement new features. Small development teams are common for LBM software packages (as shown in Figure 1).

8 Threats To Validity

The measures in the measurement template [20] may not be broad enough to accurately capture some qualities. For example, there are only two measures of surface robustness. Similarly, reusability is assessed by the number of code files and LOC per file, assuming that a large number of relatively small files implies modularity. Furthermore, the measurement of understandability relies on 10 random source code files. It is possible that the 10 files that were chosen to represent a software package may not be representative.

Another risk to the validity of the proposed approach is missing or incorrect data. Some software package data may have been missed due to technology issues like broken links. For the LBM example, this issue arose with the measurement of Palabos, which had a broken link to its user manual. Some pertinent data may not have been specified in public artifacts, or may be obscure within an artifact or web-page. For the LBM example, the use of unit testing and continuous integration was mentioned in the artifacts of only three (ESPResSo, Ludwig, Musubi) packages. However, interviews suggested a more frequent use of both unit testing and continuous integration in the development processes.

9 Concluding Remarks

To improve the development of research software, both in terms of productivity, and the resulting software quality, we need to understand the current state of the practice. An exciting strategy to approach this goal is to assess one domain at a time, collecting data from developer interviews and digging deeply into their code repositories. By providing feedback specific to their domain, the developer community can be drawn into a dialogue on how to best make improvements going forward. Moreover, they can be encouraged to share their best practices between one another, and with other research software domains.

We have outlined a methodology for assessing the state of the practice for any given research software domain based on a list of about 30 representative projects. In addition to interviews, we use manual and automated inspection of the artifacts in each project's repositories. The repository data is collected by filling in a 108 question measurement template, which requires installing the software on a VM, running simple tests (like completing the getting started instructions (if present)), and searching the code, documentation and test files. Using AHP the projects are ranked for 9 individual qualities (installability, correctness and verifiability, reliability, robustness, usability, maintainability, reusability, surface understandability, visibility/transparency) and for overall quality. Perspective and insight is shared with the user community via the following: i) comparing the ranking by best practices against an estimate of the community's ranking of popularity; ii) comparing artifacts, tools and processes to current research software development guidelines; and, iii) exploring pain points via developer interviews. Using our methodology, spreadsheet templates and AHP tool, we estimate (based on our experience with using the process) the time to complete an assessment for a given domain at 173 person hours.

For the running example of LBM we found that the top packages engaged in most of recommended best practices, including examples of practising peer review. However, we did find room for improvement with respect to providing API documentation, a roadmap, a code of conduct, programming style guide, and uninstall instructions. In addition, the community could likely benefit by increased use of continuous integration.

Building from the LBM example we can create a wealth of data on multiple domains. The next step is a meta-analysis, where we look at how the different domains compare to answer new research questions like: What lessons from one domain could be applied in other domains? What (if any) differences exist in the pain points between domains? Are there differences in the tools, processes, and documentation between domains?

References

1. Brett, A., Cook, J., Fox, P., Hinder, I., Nonweiler, J., Reeve, R., Turner, R.: Scottish covid-19 response consortium. `https://github.com/ScottishCovidResponse/modelling-software-checklist/blob/main/software-checklist.md` (August 2021)
2. Carver, J.C., Kendall, R.P., Squires, S.E., Post, D.E.: Software development environments for scientific and engineering software: A series of case studies. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. pp. 550–559. IEEE Computer Society, Washington, DC, USA (2007). `https://doi.org/http://dx.doi.org/10.1109/ICSE.2007.77`
3. Dong, A.: Assessing the State of the Practice for Medical Imaging Software. Master's thesis, McMaster University, Hamilton, ON, Canada (September 2021)
4. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edn. (2003)
5. van Gompel, M., Noordzij, J., de Valk, R., Scharnhorst, A.: Guidelines for software quality, CLARIAH task force 54.100. `https://github.com/CLARIAH/software-quality-guidelines/blob/master/softwareguidelines.pdf` (September 2016)
6. Hannay, J.E., MacLeod, C., Singer, J., Langtangen, H.P., Pfahl, D., Wilson, G.: How do scientists develop and use scientific software? In: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering. pp. 1–8. SECSE '09, IEEE Computer Society, Washington, DC, USA (2009). `https://doi.org/10.1109/SECSE.2009.5069155`
7. Heroux, M.A., Bieman, J.M., Heaphy, R.T.: Trilinos developers guide part II: ASC softwar quality engineering practices version 2.0. `https://faculty.csbsju.edu/mheroux/fall2012_csci330/TrilinosDevGuide2.pdf` (April 2008)
8. Kågström, B., Ling, P., Van Loan, C.: Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. ACM Transactions on Mathematical Software (TOMS) **24**(3), 268–302 (1998)
9. Kelly, D.: Industrial scientific software: A set of interviews on software development. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research. pp. 299–310. CASCON '13, IBM Corp., Riverton, NJ, USA (2013), `http://dl.acm.org/citation.cfm?id=2555523.2555555`

10. Michalski, P.: State of The Practice for Lattice Boltzmann Method Software. Master's thesis, McMaster University, Hamilton, Ontario, Canada (September 2021)
11. Nangia, U., Katz, D.S.: Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research. pp. 1–6. Zenodo (Jun 2017). https://doi.org/10.5281/zenodo.814220, https://doi.org/10.5281/zenodo.814220
12. Nguyen-Hoan, L., Flint, S., Sankaranarayana, R.: A survey of scientific software development. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 12:1–12:10. ESEM '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1852786.1852802, http://doi.acm.org/10.1145/1852786.1852802
13. Orviz, P., García, Á.L., Duma, D.C., Donvito, G., David, M., Gomes, J.: A set of common software quality assurance baseline criteria for research projects (2017). https://doi.org/10.20350/digitalCSIC/12543
14. Pinto, G., Wiese, I., Dias, L.F.: How do scientists develop and use scientific software? an external replication. In: Proceedings of 25th IEEE International Conference on Software Analysis, Evolution and Reengineering. pp. 582–591 (Feb 2018). https://doi.org/10.1109/SANER.2018.8330263
15. Saaty, T.L.: The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation. McGraw-Hill Publishing Company, New York, New York (1980)
16. Schlauch, T., Meinel, M., Haupt, C.: Dlr software engineering guidelines (Aug 2018). https://doi.org/10.5281/zenodo.1344612, https://doi.org/10.5281/zenodo.1344612
17. Segal, J.: When software engineers met research scientists: A case study. Empirical Software Engineering **10**(4), 517–536 (Oct 2005). https://doi.org/10.1007/s10664-005-3865-y, http://dx.doi.org/10.1007/s10664-005-3865-y
18. Smith, B., Bartlett, R., Developers, x.: xsdk community package policies (Dec 2018). https://doi.org/10.6084/m9.figshare.4495136.v6, https://figshare.com/articles/journal_contribution/xSDK_Community_Package_Policies/4495136/6
19. Smith, W.S.: Beyond software carpentry. In: 2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18). pp. 1–8 (2018)
20. Smith, W.S., Carette, J., Michalski, P., Dong, A., Owojaiye, O.: Methodology for assessing the state of the practice for domain X. https://arxiv.org/abs/2110.11575 (October 2021)
21. Smith, W.S., Lazzarato, A., Carette, J.: State of practice for mesh generation software. Advances in Engineering Software **100**, 53–71 (Oct 2016)
22. Smith, W.S., Lazzarato, A., Carette, J.: State of the practice for GIS software. https://arxiv.org/abs/1802.03422 (Feb 2018)
23. Smith, W.S., Sun, Y., Carette, J.: Statistical software for psychology: Comparing development practices between CRAN and other communities. https://arxiv.org/abs/1802.07362 (2018), 33 pp.
24. Smith, W.S., Zeng, Z., Carette, J.: Seismology software: State of the practice. Journal of Seismology **22**(3), 755–788 (May 2018)
25. Szulik, K.: Don't judge a project by its github stars alone. https://blog.tidelift.com/dont-judge-a-project-by-its-github-stars-alone (December 2017)
26. Thiel, C.: EURISE network technical reference. https://technical-reference.readthedocs.io/en/latest/ (2020)
27. USGS: USGS software plannning checklist. https://www.usgs.gov/media/files/usgs-software-planning-checklist (December 2019)

28. Wiese, I.S., Polato, I., Pinto, G.: Naming the pain in developing scientific software. IEEE Software pp. 1–1 (2019). `https://doi.org/10.1109/MS.2019.2899838`
29. Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., Teal, T.K.: Good enough practices in scientific computing. CoRR **abs/1609.00037** (2016), `http://arxiv.org/abs/1609.00037`