State of the Practice for Lattice Boltzmann Method Software

Anonymous

Abstract

We analyze the state of software development practice for Lattice Boltzmann solvers by quantitatively and qualitatively measuring and comparing 24 software packages for 10 software qualities (installability, correctness/ verifiability, reliability, robustness, usability, maintainability, reusability, understandability, visibility/transparency and reproducibility). Our reproducible analysis method employs a measurement template (containing 108 measures that are manually and automatically extracted from the project repository) and developer interviews (consisting of 20 questions). After measuring, we rank the software using the Analytic Hierarchy Process (AHP). Our ranking is roughly consistent with GitHub stars ranking, suggesting at least a partial correlation between the use of best practices and popularity. We find the state of the practice to be healthy, with 67% of the measured packages ranking in the top five for at least one quality, the majority of LBM generated artifacts corresponding to general recommendations from research software developers, common use of version control (67% of packages) and the adoption of a quasi-agile development process. Areas of best practice to potentially improve include adoption of continuous integration, API documentation and enforcement of programming style guides. We interviewed four developers to gain insight into their current pain points. Identified challenges include lack of development time, lack of funding, and difficulty with ensuring correctness. Developers are addressing these pain points by designing for change, circumventing the oracle problem and prioritizing documentation and usability. For future improvements we suggest the following: employing linters, conducting rigorous peer reviews, writing and submitting more papers on software, growing the number of contributors by following current recommendations for open source projects, and augmenting the theory manuals to include more requirements specification relevant information.

Keywords: Lattice Boltzmann Method (LBM), research software, software engineering, software quality, Analytic Hierarchy Process (AHP)

1. Introduction

We analyze the development of Computational Fluid Dynamics (CFD) software packages that use the Lattice Boltzmann Method (LBM). LBM packages form a family of algorithms for simulating single-phase and multiphase fluid flows, often incorporating additional physical complexities (Chen and Doolen, 1998), such as reflective and non-reflective boundaries. LBM solvers consider the behaviour of a collection of particles as a single unit at the mesoscopic scale, which lies between the nanoscopic and microscopic scales. These solvers predict the positional probability of a collection of particles moving through a lattice structure following a two-step process: i) streaming, where the particles move along the lattice via links; and, ii) colliding, where particles transfer energy and momentum when they collide (Bao and Meskas, 2011). As an example of an important application of LBM, Figure 1 shows the flow in a defective aorta pre- and post-medical intervention. LBM has several advantages over conventional CFD methods, including a simple calculation procedure, improved parallelization, and robust handling of complex geometries (Ganji and Kachapi, 2015).

A small sample of safety and mission-critical applications of LBM includes the following: designing fuel cells (Zhang et al., 2018), modelling groundwater flow (Anwar and Sukop, 2009), and analyzing the flow of blood in the cardiovascular system (Sadeghi et al., 2022a,b, 2020). As these examples illustrate, engineers and scientists use LBM for tackling problems that impact such areas as environmental policy, manufacturing, and health and safety. Given the critical applications of LBM, users worry about software qualities like reliability, robustness, reproducibility, performance, correctness, and verifiability. Since their time is valuable, developers of LBM libraries have additional quality concerns, including maintainability, reusability, and understandability. With these quality considerations in

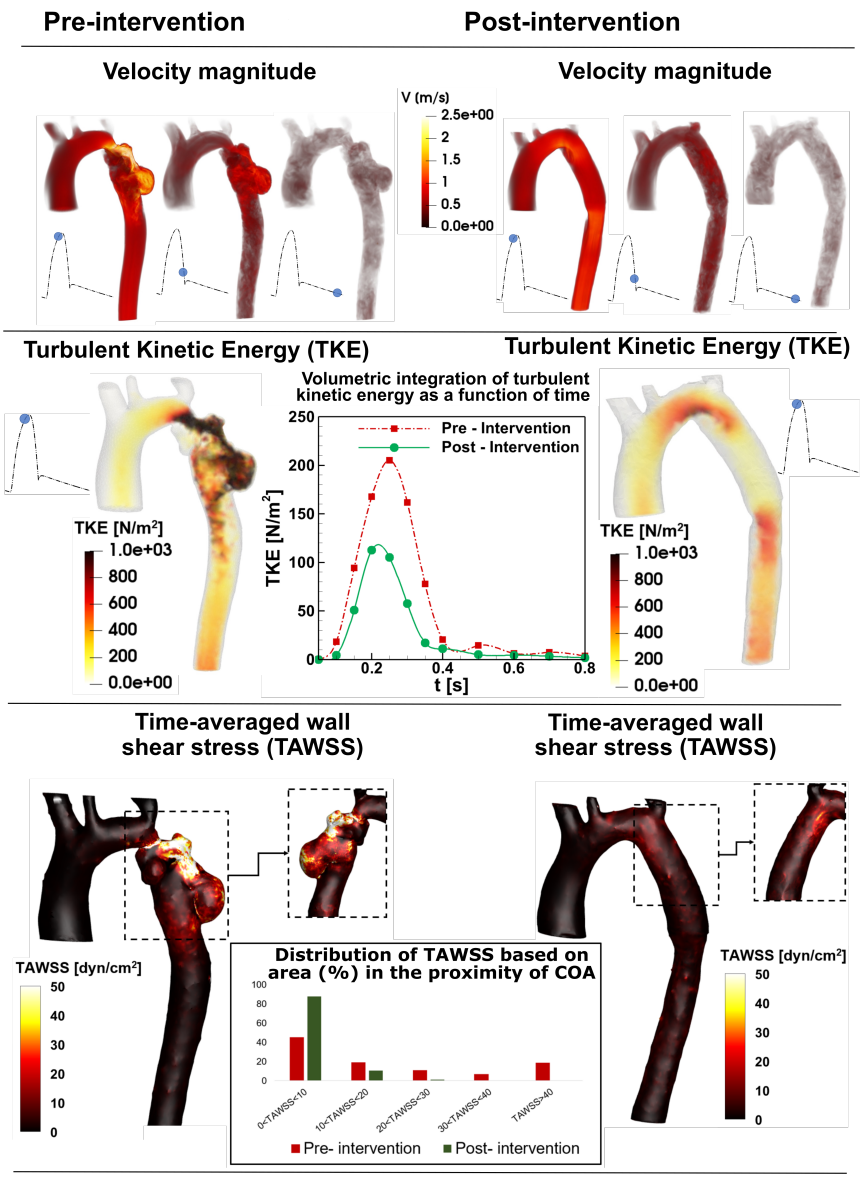

Figure 1: LBM (and Lattice Poisson-Boltzmann) computed flow (velocity magnitude, turbulent kinetic energy and time averaged wall shear stress) in an aorta with a coarctation (narrowing of the artery) and a major aneurysm (a ballooning of the artery wall) downstream of the coarctation, pre and post intravascular stent intervention (Sadeghi et al., 2020)

.

mind, and the potentially overwhelming number of choices for existing LBM libraries, the time is right to assess the state of the practice. Therefore, we analyze the current state of LBM software development and compare it to the development practices employed for research software in general. We want to highlight success stories that LBM developers can share amongst their community, and with the broader research software community, while at the same time looking for areas for potential future improvement.

We use 10 research questions to structure and focus our discussion. These questions drive our methodology. For

each question below, we point to the section that contains our answer. We start with identifying the examples of LBM software where we have access to the source code:

RQ1: What LBM software projects exist, with the constraint that the source code must be available for all identified projects? (Section 2)

We next wish to assess the representative software to determine how well they apply current software development best practices. By *best practices*, we mean methods, techniques, and tools that are generally believed to improve software development, like testing and documentation. We categorize our best practices around software qualities (Section 2.1). At this point in the process, to remove potential user/developer bias, we 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 also compare our ranking to how the user community 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? (Section 3)

RQ3: How similar is the list of top projects identified in RQ2 to the most popular projects, as viewed by the scientific community? (Section 4)

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 LBM software compares to other research software. For cases where data is available, we can compare to other specific domains of research software, like medical imaging (Dong, 2021) and ocean modelling (Jung et al., 2022). However, most of the literature on the state of the practice does not focus on specific domains, but rather on all research software. We will refer to studies and guidelines that do not distinguish an application domain as *research software in general*. By comparing LBM software to research software in general we can determine how LBM follows the trends from other developer communities and how it is different.

RQ4: How do LBM projects compare to research software in general with respect to the artifacts present in their repositories? (Section 5)

RQ5: How do LBM projects compare to research software in general with respect to the use of tools (Section 6) for:

RQ5.a development; and,

RQ5.b project management?

RQ6: How do LBM projects compare to research software in general with respect to principles, processes, and methodologies used? (Section 7)

Only so much information can be gleaned by looking at software repositories. To gain additional insight, we need to interview developers. We need to learn their concerns and how they deal with them; we need to understand their pain points. We wish to know what practices top LBM projects use, so that others can potentially emulate these practices. Part of our goal is to identify new practices that LBM developers can adopt to improve their software in the future. The question below cover these points:

RQ7: What are the pain points for developers working on LBM software projects? (Section 8)

RQ8: How do the pain points of developers from LBM compare to the pain points for research software in general? (Section 8)

RQ9: For LBM developers what specific best practices address their pain points and software quality concerns? (Section 9)

RQ10: What research software development practices could potentially address the pain point concerns identified in RQ7. (Section 10)

We investigate the research questions by applying the general methodology summarized in Anonymous (xxxx)[1]. The specific application of the methodology to LBM is reviewed in Section 2, with the full details in Anonymous (xxxx)[1]. Our methodology updates an approach used in prior assessments of domains like Geographic Information Systems (Smith et al., 2018b), Mesh Generators (Smith et al., 2016b), Oceanographic Software (Smith et al., 2015), Seismology software (Smith et al., 2018e), statistical software for psychology (Smith et al., 2018d) and medical image analysis software (Dong, 2021).

We start by identifying 24 LBM software packages. We then approximately measure the application of best practices for each package by filling in a grading template. Our analysis points to areas where some LBM software packages fall short of current best practices. Our intention is not to criticize any existing packages, especially since not every project is actively maintained. Moreover, we recognize that not every project needs to achieve the highest possible quality. However, rather than delve into the nuances of which software can justify compromising which practice we conducted our comparison under the ideal assumption that every project can have sufficient resources to match best practices.

Compared with our previous methodology (Anonymous, xxxx)[1], measuring the application of best practices now also includes repository based metrics, such as the number of files, number of lines of code, percentage of issues that are closed, etc. We rank the software with the Analytic Hierarchy Process (AHP), using the quantitative data from the grading template. After this, as another addition to our previous methodology, we interview some development teams to understand the status of their development processes. We next answer the research questions using the collected quantitative and qualitative data (Sections 2 to 10). Finally, we summarize the threats to validity (Section 11) and the final conclusions and recommendations (Section 12).

2. Methodology

We apply our methodology for evaluating the state of the practice of research software to the specific domain of LBM software. Since we group best practices around conventional software qualities, the first section below provides definitions for the qualities of interest (Section 2.1), along with information on how to measure these qualities. The following section (Section 2.2) provides an overview of the steps we used to select, measure and compare LBM software. The remaining sections (Sections 2.3 to 2.9) provide details on each step in the methodology.

2.1. Measuring Software Qualities

To ensure our assessment of best practices is organized and complete, we centre our measurements around software qualities. For completeness, we use the list of software engineering qualities from Ghezzi et al. (2003, p. 17–33), excluding interoperability, productivity, and timeliness. We excluded interoperability because our focus is on LBM software alone, not its interaction with other software. Our exclusion of the other two qualities is because they are process qualities, and our focus is on product qualities. We added installability to our list since potential users will ignore software they cannot install. Since the scientific method depends on reproducibility, we also added this quality. Below we provide information on each quality, including details on how they are measured.

Installability Installability is measured by the effort required for the installation, uninstallation, or reinstallation of a software product in a specified environment (ISO/IEC, 2011; Lenhard et al., 2013). In our case, the effort includes the following: time spent finding and understanding the installation instructions, the person-time and resources spent performing the installation procedure, and the absence or ease of overcoming system compatibility issues. An installability score improves if the software has means to validate the installation and instructions for uninstallation.

[1] Anonymous for double-blind review

Correctness A software program is correct if it behaves according to its stated specifications (Ghezzi et al., 2003, p. 17). Therefore, measuring correctness requires that a specification is available. However, research software is unlikely to have an explicit specification. As a consequence, correctness often cannot be measured directly. We assess correctness indirectly by looking for the following: availability of a requirements specification, reference to domain theory, and explicit use of tools and/or techniques for building confidence, such as documentation generators and software analysis tools.

Verifiability An artifact (like code, or a theory manual) is verifiable if, and only if, every statement therein is verifiable. A statement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that each statement is correct (adapted from recommended practice for requirements specifications (IEEE, 1998)). Similar to correctness, verifiability increases with the presence of a specification and with references and pointers to domain knowledge. A good measure of verifiability results from the following: availability of well-written tutorials that include expected output, software unit tests, and evidence of Continuous Integration (CI). Our process measures correctness and verifiability together.

Reliability Reliability is measured by the probability of failure-free operation of a computer program in a specified environment for a specified time (Ghezzi et al., 2003, p. 357), (Musa et al., 1987). The absence of errors during installation and use increases the reliability score. Recoverability from errors also improves reliability.

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 software inputs violate the requirements specification assumptions (Boehm, 2007), (Ghezzi et al., 2003, p. 19). A good measure of robustness correlates with a reasonable reaction to unexpected input, including data of the wrong type, empty input, or missing files or links. Examples of reasonable reactions include an appropriate error message and the ability to recover the system.

Performance Performance is measured by the degree to which a system or component accomplishes its designated functions within given constraints, such as speed (database response times, for instance), throughput (transactions per second), capacity (concurrent usage loads), and timing (hard real-time demands) (IEEE, 1991; Wiegers, 2003). Our state of the practice assessment does not directly quantitatively measure performance. Instead, we check the documentation for each software package looking for information alluding to consideration of performance, such as testing results or parallelization instructions.

Usability Usability is measured by the extent to which a software product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use (Nielsen, 2012). We assume that high usability correlates with the presence of documentation, including tutorials, manuals, defined user characteristics, and user support. Preferably the user support model will have avenues to contact developers and report issues.

Maintainability A measure of maintainability is the effort with which a software system or component can be modified to correct faults, improve performance or other attributes, and satisfy new requirements (IEEE, 1991; Boehm, 2007). In the current work, we measure maintainability by the quality of the documentation and the presence of version control and issue tracking. There are many documentation artifacts that can improve maintainability, including user and developer manuals, specifications, README files, change logs, release notes, publications, forums, and instructional websites. We also ask developers if they considered the ease of future changes when developing their software.

Reusability Reusability refers to the extent to which components of a software package can be used with or without adaptation in other software packages (Kalagiakos, 2003). A good indicator of reusability is many easily reusable components. Therefore, we look for high modularization, which we define as the presence of small components with well-defined interfaces. For this state of the practice assessment, we assume that a good measure of reusability correlates with a high number of code files, and the availability of API documentation.

Understandability Understandability is measured by the capability of the software package to enable the user to understand its suitability and function (ISO/IEC, 2001). It is an artifact-dependent quality. Understandability

is different for the user interface, source code, and documentation. In this state of the practice analysis, understandability focuses on the source code. We measure it by the consistency of formatting style, the extent of modularization, the explicit identification of coding standards, the presence of meaningful identifiers, and the clarity of comments.

Visibility and Transparency Visibility and transparency refer to the extent to which all the steps of a software development process, and its current status, are conveyed clearly (Ghezzi et al., 2003, p. 32). In this state of the practice assessment a good measure of visibility and transparency correlates with a well-defined development process, documentation of the development process and environment, and software version release notes.

Reproducibility Software achieves reproducibility if another developer can take the original code and input data, run it, and obtain the original output (Benureau and Rougier, 2017). We measure reproducibility qualitatively by asking developers if they have any concerns that their computational results are not reproducible, and if they have taken any steps to ensure reproducibility.

2.2. Overall Process

Our assessment process consists of the following steps:

1. List candidate software packages for the domain. (Section 2.3)
2. Filter the software package list. (Section 2.4)
3. Gather the source code and documentation for each software package.
4. Collect quantitative measures from the project repositories. (Section 2.5)
5. Measure using the measurement template. The full measurement template can be found in Anonymous (xxxx)[1]. (Section 2.5)
6. Rank the software using AHP. (Section 2.6)
7. Interview the developers. (Section 2.7)
8. Analyze the domain. (Section 2.8)
9. Analyze the results and answer the research questions.

The above steps depend on interaction with a Domain Expert partner, as discussed in Section 2.9.

2.3. Identify Candidate Software

To answer RQ1 we identify existing LBM projects through search engine queries targeting authoritative lists. We find the software in scholarly articles, GitHub and swMATH. The Domain Expert (Section 2.9) also helps with selecting the candidate software. We build our list based on the following criteria:

1. The software functionality must fall within the identified domain.
2. The source code must be viewable.
3. The empirical measures should be available, which implies a preference for GitHub-style repositories.
4. The software cannot be marked as incomplete, or in an initial state of development.

Our initial list had 45 packages, including some that we later found to be unusable because of incompleteness, or an absence of publicly available source code.

2.4. Filter the Software List

To reduce the length of the list and to answer RQ1, we filter the initial list of 45 packages. We apply the following filters in the priority order listed.

1. Scope: We remove software by narrowing what functionality we consider within the scope of the domain.
2. Usage: We eliminate software if its installation procedure is missing or unclear.
3. Age: We eliminate the older software packages (with age measured by the last date of a change), except when an older software package appears to be highly recommended and currently in use.

For the third item in the above filter, we categorize software packages as 'alive' if documentation updates happened within the last 18 months, and 'dead' otherwise.

Name	Released	Updated	Relevant Publication
DL_MESO (LBE)	unclear	2020 Mar	(Seaton et al., 2013)
ESPResSo	2010 Nov	2020 Jun	(Weik et al., 2019)
ESPResSo++	2011 Feb	2020 Apr	(Halverson et al., 2013)
lbmpy	unclear	2020 Jun	(Bauer et al., 2021b)
lettuce	2019 May	2020 Jul	(Bedrunka et al., 2021)
Ludwig	2018 Aug	2020 Jul	(Desplat et al., 2001)
LUMA	2016 Nov	2020 Feb	(Harwood et al., 2018)
MechSys	2008 Jun	2021 Oct	(Galindo-Torres, 2013)
Musubi	2013 Sep	2019 Aug	(Hasert et al., 2014)
OpenLB	2007 Jul	2019 Oct	(Heuveline and Krause, 2010)
Palabos	unclear	2020 Jul	(Latt et al., 2021)
pyLBM	2015 Jun	2020 Jun	
Sailfish	2012 Nov	2019 Jun	(Januszewski and Kostur, 2014)
TCLB	2013 Jun	2020 Apr	(Rokicki and Laniewski-Wollk, 2016)
waLBerla	2008 Aug	2020 Jul	(Bauer et al., 2021a)

Table 1: Alive Software Packages (Sorted Alphabetically)

Filtering by scope, usage, and age decreased the size of the list to 23 packages. We removed 22 packages for the following reasons: we cannot install them, they are incomplete, source code is not publicly available, we are not licensed to use them, or the project is out of scope. Michalski (2021) lists the eliminated software packages. Of the remaining 23 packages, we kept some despite their 'dead' status because of their presence on authoritative LBM software lists and because of their surface excellence. After the initial measurement exercise, which took place in mid-2020, we added one more package in January 2022: Musubi. We found Musubi after our initial search, but we felt obliged to include it since it clearly fits our criteria. Time constraints meant we could not update the previous manual data collection, but we did redo the automated data collection.

The final list of 24 software packages is in two tables. Table 1 lists packages that fall into the 'alive' category as of mid-2020 (January 2022 for Musubi), and Table 2 lists packages that are 'dead'. The tables include a hyperlink to each project and, when available, citations for relevant publications. The final list of software packages shows considerable variation in purpose, size, user interfaces, and programming languages. For example, the OpenLB software package is predominantly a C++ package that uses hybrid parallelization and addresses many CFD problems (Heuveline et al., 2009). The software package pyLBM is an all-in-one Python language package for numerical simulations (Graille and Gouarin, 2017). ESPResSo consists of extensible C++ and Python code specifically for research on soft matter (Weik et al., 2019). The HemeLB package simulates fluid flow in several medical domains and is written predominantly in C, C++, and Python (Mazzeo and Coveney, 2008).

Name	Released	Updated	Relevant Publication
HemeLB	2007 Jun	2018 Aug	(Mazzeo and Coveney, 2008)
laboetie	2014 Nov	2018 Aug	(Levesque et al., 2013)
LatBo.jl	2014 Aug	2017 Feb	
LB2D-Prime	2005	2012 Apr	
LB3D	unclear	2012 Mar	(Schmieschek et al., 2017)
LB3D-Prime	2005	2011 Oct	
LIMBES	2010 Nov	2014 Dec	
MP-LABS	2008 Jun	2014 Oct	
SunlightLB	2005 Sep	2012 Nov	

Table 2: Dead Software Packages (Sorted Alphabetically)

2.5. *Quantitative Measures*

We use a measurement template to rank the projects by how well they follow best practices (RQ2), as described in Anonymous (xxxx)[1]. For each software package (each column in the template), we fill in the rows of the template. This process takes about two hours per package, with a cap of four hours. The time constraint is necessary so that the workload is feasible for a team as small as one, given our aim to cap the measurement phase at 160 person-hours (Anonymous, xxxx)[1]. An excerpt of the template, in spreadsheet form, is shown in Figure 2.

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								
…	…	…	…	…	…	…	…	…

Figure 2: Excerpt of the Top Sections of the Measurement Template

The full template consists of 108 questions categorized under nine qualities (Section 2.1): (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.

We designed the questions to be unambiguous, quantifiable, and measurable with limited time and domain knowledge. We categorize them by quality, with an extra category for summary information (Figure 2). The summary section provides general data, such as the software name, number of developers, etc. Gewaltig and Cannon (2012) provide us with our software categories. Public means software available to all, while private means software available only to a specific group. The concept category defines software written to demonstrate algorithms or concepts. The three categories of development models are open source, where source code is freely available under an open-source license; free-ware, which freely provides a binary or executable, but not source code; and commercial, where the user must pay.

Several of the qualities use the word "surface" 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 directly. Instead, we look for usability considerations by searching for cues in the documentation, such as the presence of getting started instructions, a user manual, or documentation of expected user characteristics.

We used tools to automatically measure the number of files, number of lines of code (LOC), percentage of closed issues, etc. GitStats (Gieniusz, 2019) measures each 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. Sloc Cloc and Code (scc) (Boyter, 2021) measures the number of text files and the number of total, code, comment, and blank lines in each GitHub repository.

We used Virtual Machines (VMs) to provide an optimal testing environment for each package. We used VMs to create fresh environments, eliminating any worries about existing libraries and conflicts. Moreover, when the tests are complete, the VM can be deleted, without impacting the host operating system. The most significant advantage of using VMs is that they level the playing field. Every software install starts from a clean slate, which removes "works-on-my-computer" errors.

2.6. Analytic Hierarchy Process

Once we measure each package, we still need to rank them to answer RQ2. We do this via the Analytic Hierarchy Process (AHP), a decision-making technique that compares multiple options by multiple criteria. Our AHP algorithm uses the overall impression score, gathered via the measurement template, to compare and rank the LBM software. AHP performs a pairwise analysis using a matrix and generates an individual quality score for each software package. Anonymous (xxxx)[1] shows the details on how AHP ranks software based on quality measures.

We use our own simple tool for conducting the AHP analysis. The tool includes a sensitivity analysis to ensure that the software package rankings are appropriate with respect to the uncertainty of the quality scores. For the sensitivity analysis, we modified the score by 10% for each package and verified that the overall ranking was stable.

2.7. Interview Developers

Two of the research question (RQ7 and RQ9) require going beyond the quantitative data from the measurement template. To gain insight, we interview developers using a list of 20 questions from Anonymous (xxxx)[1]. 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 focus on specific software qualities, such as maintainability, understandability, usability, and reproducibility. The interviews are semi-structured based on the question list; we ask follow-up questions when necessary. Based on our experience, the interviewees often raise exciting and unexpected ideas.

We requested interviews with developers for all 24 packages, except Musubi, since we added Musubi after our ethics approval expired. We found the developers from project websites, code repositories, publications, and biographic information on institution webpages. Four developers agreed to participate in our study. We met with them online, using either Zoom or Teams to facilitate recording and automatic transcription. Each meeting lasted about an hour.

2.8. Domain Analysis

Since the LBM domain has a reasonably small scope, we can view the software as constituting a program family. Parnas (1976) defines a program family as "a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members." Domain analysis is the term for studying the common properties within a family of related programs.

The domain analysis (also called a commonality analysis) consists of understanding the commonalities, variabilities, and common terminology for a program family (Weiss, 1997). Commonalities are goals, theories, models, definitions, and assumptions that are common between family members. Variabilities are goals, theories, models, definitions, and assumptions that differ between family members. Associated with each variability are parameters of variation, which summarize the possible values for that variability, along with their potential binding time. The binding time is when a variability is assigned a value. Binding time options include specification time, build time (when the program compiles) and run time (when the code executes). For the current assessment, the packages all have the commonality of using LBM techniques to simulate the motion of fluids. Variabilities for LBM packages include the following: the dimension of problems they simulate, whether (and how) they support parallel computation, whether compressibility can be considered, whether reflexive boundaries can be employed, whether the simulation handles multi-fluids, whether the simulation handles turbulence, and whether complex geometries can be simulated.

To focus the current presentation, we only present the results of our shallow domain analysis. We distinguish programs by their variabilities, which for research software are often assumptions. Table 3, which is sorted alphabetically, lists the variabilities as columns. Michalski (2021) provides a more detailed analysis of the commonalities and variabilities between LBM packages.

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 3: 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))) (Sorted Alphabetically)

2.9. Interaction With Domain Expert

We partnered with a Domain Expert to vet our list of projects (RQ1) and our ranking (RQ2). The Domain Expert is a crucial member of the state of the practice assessment team. Pitfalls exist if non-experts attempt to acquire an authoritative list of software or try to rank the software. Non-experts rely on online information, which has the following drawbacks: i) the online resources could have false or inaccurate information, and ii) the online resources could leave out relevant information that is too ingrained with experts for them to explicitly record it. For the current assessment, our Domain Expert (and paper co-author) is Dr. Zahra Keshavarz-Motamed, Assistant Professor of Mechanical Engineering at McMaster University, Hamilton, Ontario, Canada.

The Domain Expert has a central role in verifying the list of packages. In advance of the first meeting with the Domain Expert, we ask them to create a list of top software packages in the domain. We do this to help the expert get in the right mindset for our meeting. Moreover, by doing the exercise in advance, we avoid the potential pitfall of the expert approving the discovered list of software without giving it adequate thought. We ask the Domain Expert to vet the collected data and analysis. In particular, we ask them to vet the proposed list of software packages and the AHP ranking.

3. Ranking Projects Based on Quality Measures

We answer RQ2 by using AHP to rank the identified LBM software against best practices. The sections below summarize how the software packages compare based on the qualities discussed in Section 2.1, as measured via the measurement template (Section 2.5). The last subsection combines all the qualities into one "global" ranking. The full LBM data is available publicly on Mendeley Data (Anonymous, xxxx)[1].

3.1. Installability

All 24 software packages have installation instructions. Most have their installation instructions located in one place, often in an instruction manual or a webpage. Sometimes, like with Ludwig, we found incomplete installation instructions in one place, but complete instructions elsewhere. Installability of the software, and maintainability and correctness of the instructions themselves, are improved if all the instructions are in one location.

All 24 packages are installable on a Unix-like system. Eight packages are also installable on Windows, and five on macOS. Twenty packages list their operating system requirements, but only four (ESPResSo, LUMA, Mechsys, Sailfish) list compatible operating system versions. Ubuntu provided the testing platform for all packages, except for TCLB, which we tested on CentOS, as suggested by its installation instructions.

All but one of the software packages (LatBo.jl) have automated at least part of the installation process. Most packages, such as waLBerla and SunlightLB, use Make to automate the installation, and a few of them, like lbmpy, use custom scripts.

In most of the cases with installation errors, descriptive error messages allowed us to proceed. Systems that provided vague error messages, such as messages that did not specify which action or file was at fault, were harder to troubleshoot. Only three software packages (HemeLB, LB3D, lbmpy) that displayed a descriptive error message were not recoverable, and most of these instances were due to hardware and operating system incompatibility, such as the requirement of CUDA. Fourteen software packages broke during installation. Some packages, such as LB2D-Prime and LB3D-Prime did not provide a definitive message of the success or failure of the installation. In these instances, validating the installation required performing a tutorial or running a script, if these were available.

About half of the installation instructions assume the user is already aware of, and has already installed, dependent packages. Many packages, like ESPResSo++, Ludwig and LUMA, do not list all their dependencies. Sometimes only an error message during the installation process informs the user of a required dependency. In cases like this, we suggest a detailed rewrite of the installation instructions to accommodate a clean operating system. A virtual machine can provide an environment for installability testing.

Seventeen software packages require installation of less than 10 dependencies. All software package (except for LatBo.jl) require less than 20 dependencies. Some packages may automatically install additional dependencies in the background. Twenty of the software packages do not explicitly indicate the versions of software dependencies. Some software package installation issues could have been avoided by providing a list of dependencies. Sixteen software packages do not have detailed instructions for installing dependencies. Sixteen software packages have less than 10

manual installation steps. If one command installs dependencies, then none of the software packages take more than 20 steps to install. The average number of steps is eight, and the fewest is two (LB3D-Prime).

All but six (ESPResSo, HemeLB, laboetie, LB3D-Prime, lbmpy, waLBerla) of the software packages have a way to verify the installation. Most have tutorial examples for the user to run. Some other approaches to installation validation include validation scripts (LB2D-Prime, lettuce, Ludwig, LUMA), automatic validation after the installation (LatBo.jl), and instructions to manually review the file system (LIMBES). Only one software package (pyLBM) has uninstallation instructions.

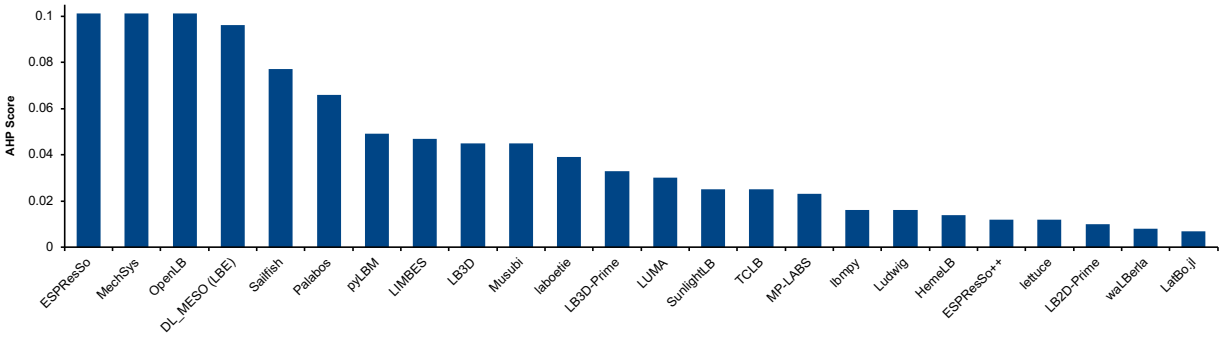

Figure 3: AHP Installability Score

Figure 3 shows the installability ranking of the software packages. Software packages with a higher score (ESPResSo, MechSys, OpenLB, DL_MESO, Palabos) tend to have one set of linear installation instructions, written as if the person doing the installation has none of the dependencies installed. The instructions often list compatible operating system versions and include instructions for installing dependencies. The top-ranked packages often incorporate some automation of the installation process. The number of dependencies a package has does not correlate with a higher score. The ability to validate the installation process, say through tutorials or test cases, leads to a higher score. Apparently active work on a project improves installability since the top seven ranked packages are all classified as alive.

3.2. Surface Correctness and Verifiability

Seventeen software packages explicitly reference domain theory, but a complete and rigorous requirements specification, in the software engineering sense (IEEE, 1998; Robertson and Robertson, 1999; ESA, 1991), is absent from all projects. Software packages that include a subset of a requirements specification, such as DL_MESO (LBE), keep the information brief and include it within other documents, such as a user manual, webpage, or a publication. In the latter case tracking down the knowledge may take significant time.

Thirteen software packages explicitly use document generation tools. Eight of them use Sphinx and the same number use Doxygen. Several packages use both.

Tutorials are available for 19 of the packages. Generally they are linearly written and easy to follow. However, only eight tutorials provide expected outputs, making it difficult to verify the correctness of their output. In these cases the user may need to assume correctness if there are no visible errors. waLBerla provides a particularly detailed tutorial, including expected output.

Unit tests are only explicitly available for two of the software packages, Ludwig and Musubi. Code modularization of most packages allows for users to create tests with varying degrees of effort. These tests allow developers and users to verify the correctness of fragments of the source code, and in doing so better assess the correctness of the entire package.

The use of Continuous Integration (CI) tools and techniques alludes to a refined development process that quickly isolates and fixes faults. Only three of the packages (ESPResSo, Ludwig, Musubi) mentioned applying the practice of CI in their development process.

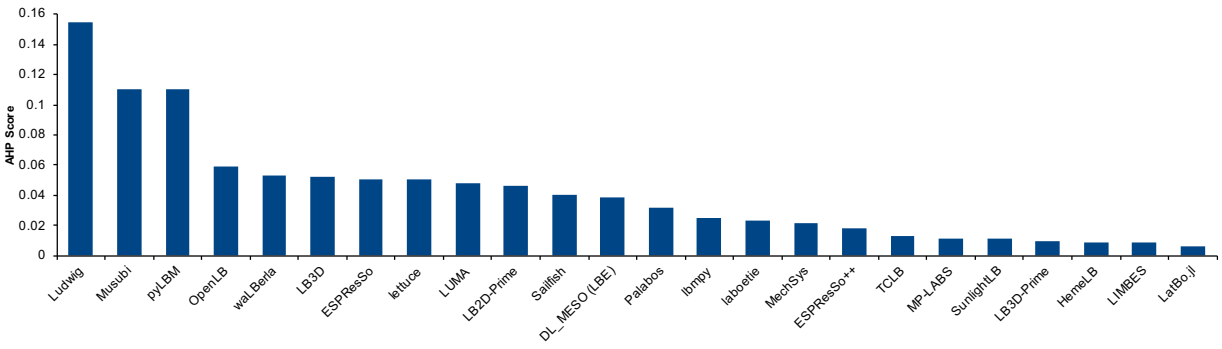

Figure 4: AHP Surface Correctness and Verifiability Score

Figure 4 shows the ranking for surface correctness and verifiability. Software packages with a higher score tend to have theory documentation. They also explicitly use at least one document generation tool that helps with verification, thus building confidence in correctness. The top ranked software packages all include getting started tutorials, and most of these include expected output. The two top ranked packages, Ludwig and Musubi, indicate the use of unit testing in their documentation. They also incorporated CI in the development process. Furthermore, eight of the top 10 ranked packages are alive.

3.3. Surface Reliability

The analysis of surface reliability focuses on package installation and tutorials. Errors occurred when installing 16 of the software packages. Every instance prompted an error message. These messages indicated unrecognized commands (even when following the installation guide), missing links, missing dependencies, and syntax errors in code files. In some instances the error messages were vague. Several automatic installation processes could not find or load dependencies. In these instances the installation tried to access outdated external repositories. We recovered and verified seven of the installations, and we assumed that one of the installations (LB3D-Prime) was recovered, since there was no way to test the installation. The installation of eight of the software packages could not be recovered. Most of these broken installations could not find external dependencies, encountered system incompatibilities, or displayed vague error messages.

Of the 14 software packages that installed correctly and also have tutorials, four (pyLBM, ESPResSo++, LIMBES, Ludwig) broke during tutorial testing. All of these instances resulted in the display of an error message. One error (pyLBM) was due to a missing tutorial dependency, another (Ludwig) was due to an invalid command despite following the tutorial, and the final two errors were vague execution errors. Of the four broken tutorial instances, only the one due to a missing dependency was recoverable.

Figure 5 shows the surface reliability ranking of the software packages. Software packages with a high score either did not break during installation, or the broken installation was recoverable. All the top five ranked packages have tutorials. The package pyLBM broke during tutorial testing, but a descriptive error message helped in recovery. Nine of the top 10 packages are alive.

3.4. Surface Robustness

Robustness testing includes handling unexpected input, such as incorrect data types, empty input, and missing files or links. A reasonable response, including appropriate error messages and an absence of unrecoverable system failures, means success.

Figure 6 shows the surface robustness ranking of the software packages. Software packages with a high score behaved reasonably in response to unexpected input as described above. All the software packages that installed correctly passed this test. They did not crash and usually output descriptive error messages. Software packages with a lower surface robustness score did not install correctly, so their robustness score may not be a true reflection of their runtime robustness. All successfully installed software packages that require a plain text input file correctly handled

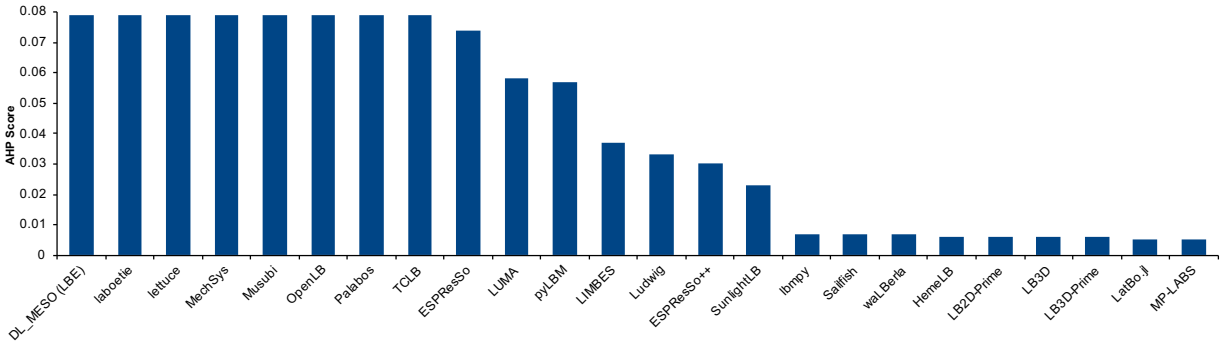

Figure 5: AHP Surface Reliability Score

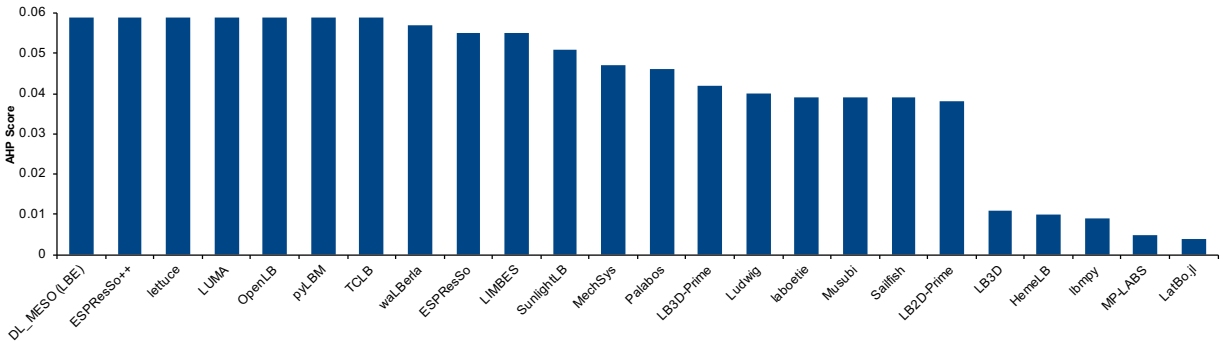

Figure 6: AHP Surface Robustness Score

an unexpected change to these input files, including a replacement of new lines with carriage returns. Following the trend of the other qualities, nine of the top 10 ranked packages are alive.

3.5. Surface Performance

Although the software packages all apply LBM to solve scientific computing problems, they differ in scope and capabilities, as shown in Table 3. Therefore, tests to compare run-time performance are not appropriate. Instead, we look through each software package's artifacts for evidence of considering performance. The artifacts of 18 software packages mentioned parallelization. This included GPU processing and the CUDA parallel computing platform, which were mentioned in the artifacts of six packages (ESPResSo, lbmpy, lettuce, pyLBM, Sailfish, TCLB). When a high degree of parallelization is possible, GPUs provide superior speed compared to CPUs. The software package TCLB is implemented in a highly efficient multi-GPU code to achieve performance suitable for model optimization (Rutkowski et al., 2020). The Ludwig package uses a so-called mixed mode approach implementing fine-grained parallelism on the GPU. MPI is used for even larger scale parallelism (Gray and Stratford, 2013). While one software package (Sailfish) required CUDA and GPU processing, some (ESPResSo, lbmpy, lettuce, pyLBM, TCLB) have the option of using either the GPU or the CPU. In general, the packages that require GPU and CUDA have better performance at the expense of installability and surface reliability.

3.6. Surface Usability

We review software repositories for the presence of a tutorial, a user manual, documented user characteristics, and a user support model. In total 19 software packages have a tutorial, 13 have a user manual, and 11 have both. The tutorials vary in scope and substance, and eight include expected output. Most user manuals are in the form of a file that

can be downloaded, while some are rendered on a webpage. Some packages (waLBerla) do not have a user manual, but do have useful documentation distributed throughout their webpages. Five software packages (laboetie, LIMBES, Ludwig, Musubi, Palabos) document expected user characteristics. Users are typically scientists or engineers. Their background is often physics, chemistry, biophysics, or mathematics. All projects (except LIMBES) have a user support model, and many of them have multiple avenues of user support. The most popular avenue of support is Git, followed by email and forums. One software package (OpenLB) has an FAQ (Frequently Asked Questions) page.

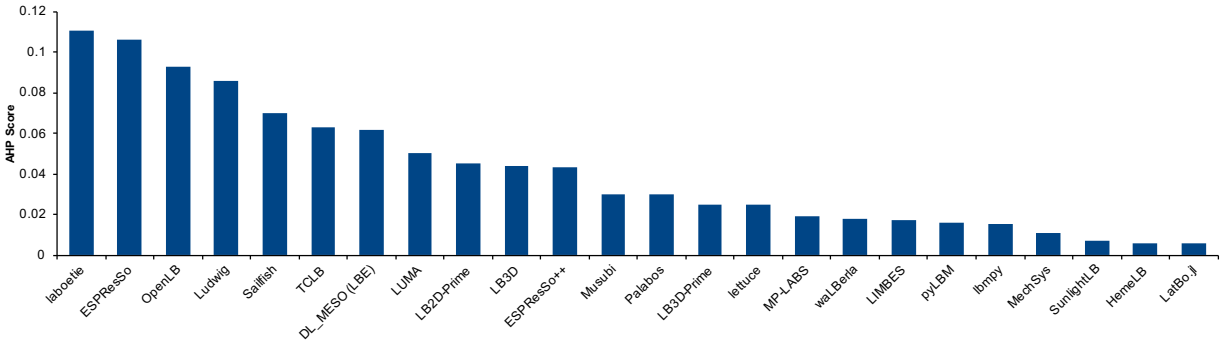

Figure 7: AHP Surface Usability Score

Figure 7 shows the surface usability ranking of the software packages. Software packages with a high score have a tutorial and user manual, sometimes have documented user characteristics, and have at least one user support model. Many packages have several user support models. Furthermore, four of the top five packages are alive.

3.7. Maintainability

We review software packages for the presence of artifacts and recorded every type of artifact that is not a code file. (The artifacts will be discussed further in Section 5.) We also reviewed the software packages for software release and documentation version numbers. This information can be used to troubleshoot issues and organize documentation. All but three software packages (LatBo.jl, LB3D-Prime, MechSys) have source code release and documentation version numbers.

We look for evidence that teams review code and have information on how users can contribute. In total, 11 software packages have this information, found in various artifacts, including in developer guides, contributor guides, user guides, developer webpages, and README files.

Twenty-three packages use issue tracking, 15 of which use GitHub or GitLab to host their git repository, seven use email, and one (SunlightLB) uses SourceForge. Most software packages that use GitHub or GitLab have the majority of their issues closed, and only three (laboetie, lettuce, Sailfish) have less than 50% of their issues closed. All the top five ranked packages have most of their issues closed. Alive packages (11 use issue tracking) have 64% of their issues closed, while dead packages (3 use issue tracking) have 71% of their issues closed. Table 4 presents this information. With respect to the host for the git repository, 13 packages use GitHub and two (Palabos, waLBerla) use GitLab. Of the other packages, one package (SunlightLB) uses CVS for issue tracking and version control, and seven do not appear to use any issue tracking system. Assuming the use of GitHub, GitLab or CVS implies the use of version control, 16 of 24 (67%) of projects use version control.

Table 4 shows the percentage of code that is comments, in decreasing order. We assume maintainability improves for packages with a higher percentage of comments. Comments represent more than 10% of code files in 16 packages, and the average percentage of code comments is about 14%. All the top five overall ranked packages have more than the average. LUMA has only 0.2% comments, the fewest of any package. However, this package also has the most lines of source code, with over four million. The next largest package is ESPResSo++ with one million.

Figure 8 shows the maintainability ranking of the software packages. Software packages with a high score provide version numbers on documents and source code releases, have an abundance of high quality artifacts, and use an issue tracking tool and version control system. These packages also appear to manage their issue tracking, having most of

Name	% Code Comments	% Issues Closed	Status
MP-LABS	26.67	100.00	Dead
Musubi	24.19	Not Git	Alive
waLBerla	22.62	72.90	Alive
OpenLB	22.43	Not Git	Alive
ESPResSo	21.78	89.26	Alive
Ludwig	20.70	60.00	Alive
Palabos	17.76	89.47	Alive
SunlightLB	17.67	Not Git	Dead
LIMBES	17.39	Not Git	Dead
ESPResSo++	17.10	66.28	Alive
HemeLB	16.68	No Issues	Dead
pyLBM	16.12	66.67	Alive
MechSys	15.11	Not Git	Alive
LB3D-Prime	14.34	Not Git	Dead
LB3D	13.76	Not Git	Dead
LB2D-Prime	13.61	Not Git	Dead
Sailfish	9.26	22.22	Alive
lettuce	8.19	33.33	Alive
DL_MESO (LBE)	8.06	Not Git	Alive
TCLB	6.02	60.32	Alive
laboetie	2.47	18.75	Dead
lbmpy	2.03	58.33	Alive
LatBo.jl	0.40	93.33	Dead
LUMA	0.20	85.71	Alive

Table 4: Git Repository Data, Sorted by Percentage of Code that is Comments (Sorted by % Code Comments)

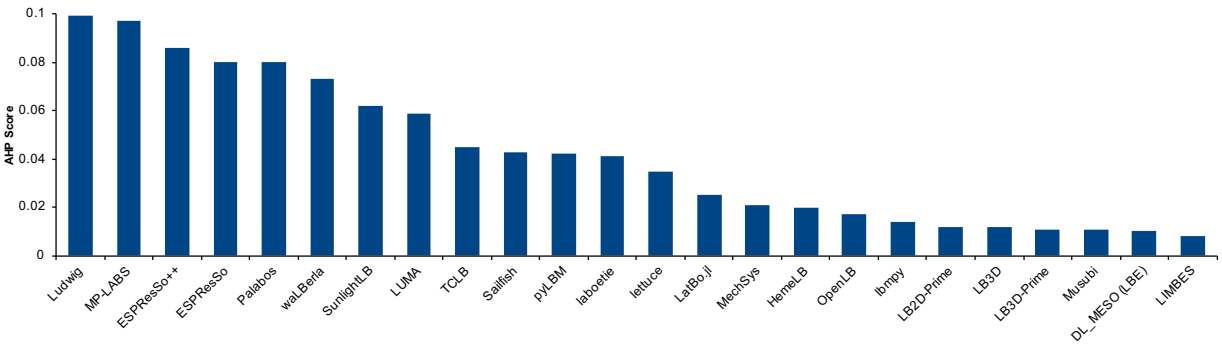

Figure 8: AHP Maintainability Score

their issues closed. Maintainable packages by our ranking have more than 10% of the code as comments. Four of the top five ranked packages are alive.

3.8. Reusability

We measure the total number of source code files for each project. We assume that numerous source files increases reusability, since more files implies increased modularization. Some packages have more features than others. We

assume this contributes to reusability, since they have more source code for potential reuse. We reviewed the software packages for the presence of API documentation, which suggests that the developers considered potential future reuse.

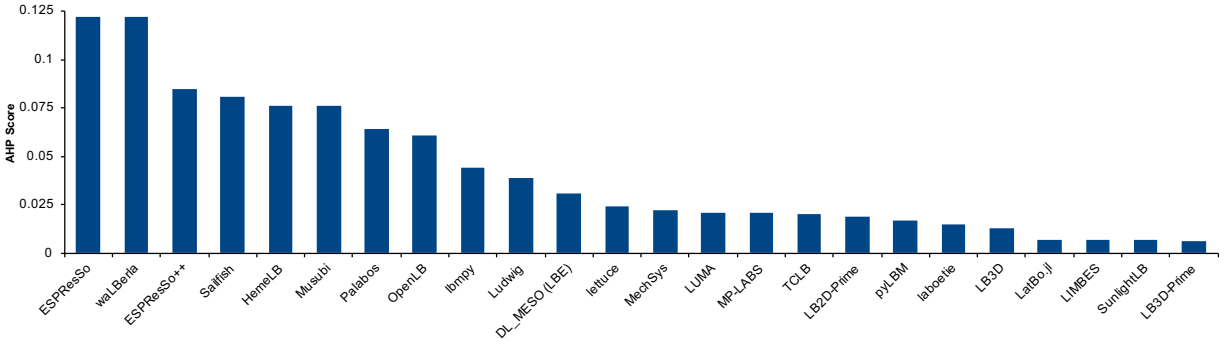

Figure 9: AHP Reusability Score

Figure 9 shows the reusability ranking of the software packages. Software packages with a high score have thousands of source code files and API documentation. The highest scoring packages, ESPResSo and waLBerla, have extensive functionality, including graphical visualizations and non-LBM modelling. For this reason, a comparison with other software packages is not on a level field. Nine of the top 10 ranked packages are alive.

Name	Binary Files	LOC	Text Files	Avg. LOC / Text File
LUMA	19	4399723	314	14011
LatBo.jl	0	42172	41	1029
LB2D-Prime	19	54755	82	668
LB3D-Prime	6	12944	23	563
DL_MESO (LBE)	51	170223	310	549
LB3D	76	39766	99	402
laboetie	1	48403	133	364
waLBerla	69	873988	2643	331
MechSys	3	95707	333	287
Palabos	71	563841	1974	286
lbmpy	29	61632	250	247
SunlightLB	1	7646	36	212
Musubi	1839	281879	1347	209
LIMBES	1	4872	26	187
Ludwig	32	162518	954	170
OpenLB	7	218406	1438	152
ESPResSo	83	195083	1390	140
MP-LABS	3	43124	307	140
pyLBM	108	37234	272	137
ESPResSo++	30	165194	1406	118
HemeLB	44	123806	1102	112
Sailfish	11	69398	632	110
lettuce	1	7660	73	105
TCLB	7	49156	594	83

Table 5: Module Data (Sorted by Avg. LOC / Text File)

Table 5 shows file and Lines Of Code (LOC) data for the software packages. Packages with a high reusability score do not have as many LOC per text file, generally having a few hundred lines or fewer. We assume that relatively small files implies modularization.

The package waLBerla scored high on reusability because of its focus on modularity. The modularity in the waLBerla framework enhances productivity, reusability, and maintainability (Bauer et al., 2021a). The design of waLBerla has enabled it to be successfully applied as a basis for extension in several projects (Bauer et al., 2021a).

3.9. Surface Understandability

To measure understandability, we review 10 random source code files per package. Using only 10 files may incorrectly estimate understandability, but for practical considerations we have to limit our time measuring each package.

All the packages appear to have consistent indentation and formatting. Only HemeLB, LUMA, and Musubi explicitly identify coding standards that are used during development. Generally, the software packages use consistent, distinctive, and meaningful code identifiers. Only four packages (LB2D-Prime, LB3D-Prime, LIMBES, MP-LABS) appear to use vague identifiers, such as single letters for variables. Thirteen source code files use symbolic constants for various parameters, mathematical constants, and matrix definitions. The code includes appropriate comments for all packages, with the comments clearly indicating what is being done (as opposed to how it is being done). The source code of 11 packages include notes on domain algorithms. Table 5 suggests that the software packages are modular to various degrees. When observing the source code files, 14 of the packages have a consistent style and order of function parameters.

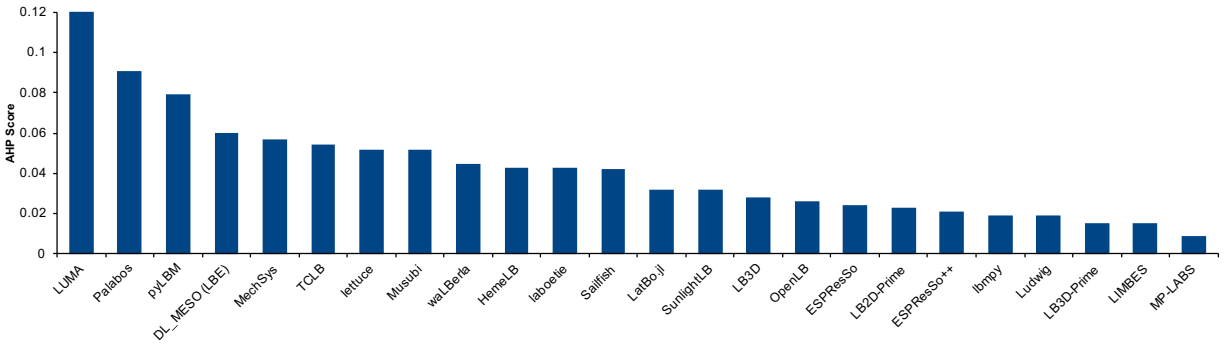

Figure 10: AHP Understandability Score

Figure 10 shows the surface understandability ranking. Software packages with a high score have a consistent indentation and formatting style, and consistent, distinctive, and meaningful code identifiers. They also have symbolic constants, and explicitly identify mathematical and LBM algorithms. Their comments are clear and indicate what is being done in the source code. The source code is well modularized and structured. All the top five ranked packages are alive.

3.10. Visibility and Transparency

We review the software artifacts for process documentation, a development environment description, and release notes. The packages tend to not explicitly use well-known development models (see also Section 7). The development teams of these packages are fairly small and easily organized without the need for such processes. Seven of the software packages did have some artifacts outlining the general development process, how to contribute, and the status of the package or its components. Eight of the packages have artifacts that note the development environment. While this information could help developers, and would improve transparency, the small close-knit nature of the development teams make explicitly specifying this information practically unnecessary. Nine of the software packages included version release notes.

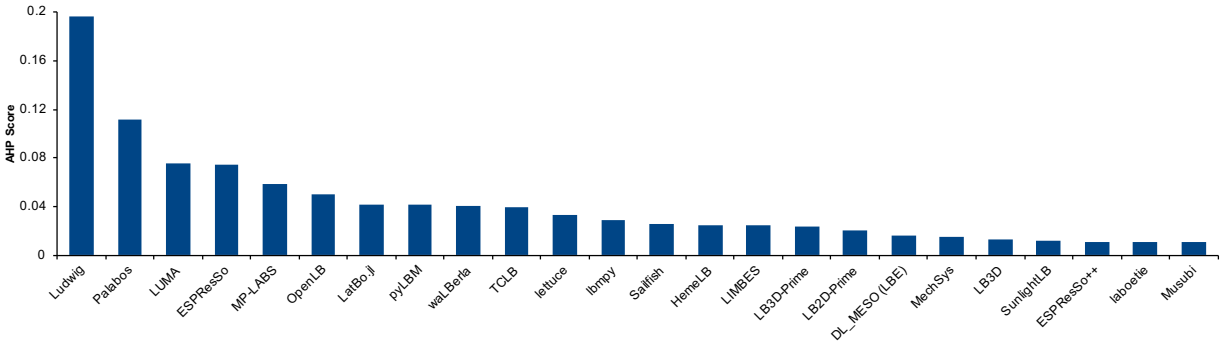

Figure 11: AHP Visibility and Transparency Score

Figure 11 shows the visibility and transparency ranking of the software packages. Software packages with a high score have an explicit development model and defined development process. They also have detailed and easy to access software release notes. Four of the top five ranked packages are alive.

3.11. Overall Quality

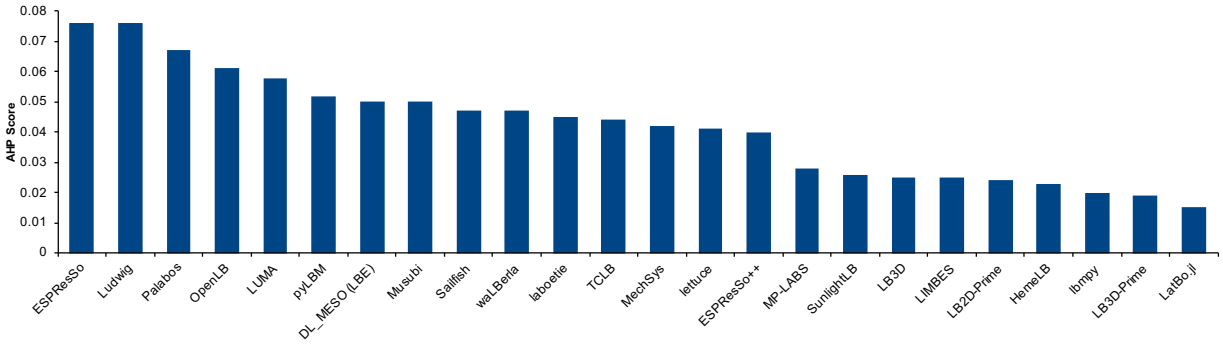

Figure 12: AHP Overall Score

Figure 12 shows the overall ranking. In the absence of other information on priorities, we calculated the ranking by assuming an equal weighting between all qualities. Software packages with an overall high score ranked high in at least several of the individual qualities.

Looking at the top three ranked packages: ESPResSo achieved a relatively high score in installability, surface usability, maintainability, reusability, and visibility and transparency. Ludwig scored high in surface correctness and verifiability, surface robustness, surface usability, maintainability, and visibility and transparency. Palabos scored high in installability, surface reliability, surface robustness, maintainability, understandability, and visibility and transparency.

4. Comparison to Community Ranking

To address RQ3, Table 6 compares our LBM software rankings against their popularity in the research community, as estimated by repository stars and watches. Nine packages do not use GitHub, so they do not have a measure of repository stars. Looking at the repository stars of the other 15 packages, we observe a pattern where packages that have been highly ranked by our assessment tend to have more stars than lower ranked packages. Our best ranked package (ESPResSo) has the second most stars, while our ninth ranked package (Sailfish) has the highest number of

stars. The repository watch column shows the same correlation, although this column contains less data, since two of the packages (Palabos, waLBerla) use GitLab, which does not track watches. Consistent with the star measure, our ninth ranked package (Sailfish) has the highest number of watches and our best ranked package (ESPResSo) has the second most watches. Packages designated as lower quality often do not use GitHub or GitLab, or have only a few stars and watches. Although the details of the rankings differ, our measures show that following best practices tends to correlate with increased popularity in the scientific community.

Notable exceptions to this trend occur for three projects that rank high by star count, but relatively low in the "follow best practices" ranking: TCLB (star rank 3, our rank 12), lettuce (star rank 4, our rank 14) and ESPResSo++ (star rank 5, our rank 15). Part of the discrepancy is because our process ranks all projects, but the star measure does not apply to all projects. Also, our process equally weights all qualities (Section 3.11), but for real users some qualities will be more important than others. Discrepancies between our measures and the star measures may also be due to inaccuracy with using stars to approximate popularity, because of how people use stars, because young projects have less time to accumulate stars, and because stars represent the community's feeling in the past more than current preferences (Szulik, 2017). Older packages (like ESPResSo++ and TCLB) have had more time to accumulate stars and watches. Young projects have less time to accumulate stars, like HemeLB, which only has 12 stars (at the time of measurement) having recently moved from a different GitHub repository. Lettuce is a young project (released in 2019) with a relatively high number of stars. The discrepancy in rankings may be due to a mismatch between the time we manually measured lettuce and when we counted the stars. The late addition of Musubi means a gap between when we measured projects and the time when our automated data (like stars) was collected. A final reason for inconsistencies between our ranking and the community's ranking is that, as for consumer products, more factors influence popularity than just quality.

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	N/A	N/A	N/A	N/A
LUMA	5	33	7	12	4
pyLBM	6	95	3	10	5
DL_MESO (LBE)	7	N/A	N/A	N/A	N/A
Musubi	8	N/A	N/A	N/A	N/A
Sailfish	9	186	1	41	1
waLBerla	10	20	9	GitLab	GitLab
laboetie	11	4	13	5	8
TCLB	12	95	3	16	3
MechSys	13	N/A	N/A	N/A	N/A
lettuce	14	48	4	5	8
ESPResSo++	15	35	5	12	4
MP-LABS	16	12	11	2	9
SunlightLB	17	N/A	N/A	N/A	N/A
LB3D	18	N/A	N/A	N/A	N/A
LIMBES	19	N/A	N/A	N/A	N/A
LB2D-Prime	20	N/A	N/A	N/A	N/A
HemeLB	21	12	11	12	4
lbmpy	22	11	12	2	9
LB3D-Prime	23	N/A	N/A	N/A	N/A
LatBo.jl	24	17	10	8	6

Table 6: Repository Ranking Metrics (Sorted by Our Ranking)

Although our ranking and the estimate of the community's ranking are not perfect measures, they do suggest a correlation between best practices and popularity. We do not know which comes first, the use of best practices or popularity, but we do know that the top ranked packages tend to incorporate best practices. The next sections will explore how the practices of the LBM community compare to the broader research software community. We will also investigate the practices from the top projects that others within the LBM community and beyond can potentially adopt.

5. Comparison Between LBM and Other Research Software for Artifacts

As part of filling in the measurement template, we listed the artifacts for each package, and then categorized them by frequency. We group the artifacts into common, uncommon, and rare categories in Table 7. Common artifacts are in 16 to 24 (>63%) of the software packages. Uncommon artifacts are in 8 to 15 (30-63%) and rare artifacts are in 1 to 7 (<30%) of the packages. We answer RQ4 by comparing the artifacts that we observed in LBM repositories to those observed and recommended for research software in general.

Common	**Uncommon**	**Rare**
Authors / Developers List	Change Log / Release Notes	Acknowledgements
Getting Started	Citation	API Documentation
Installation Guide / Instructions	Contributing	Code Style Guide
Issue Tracker	Design Documentation	Developer / Contributor Manual
Library Dependency List	Functional Spec. / Notes	FAQ / Forum
License	Performance Information	Requirements Spec.
List of Related Publications	Test Plan / Script / Cases	Product Roadmap
Makefile / Build File	User Manual / Guide	Uninstall
README File		Verification and Validation Plan
Theory Notes		Video Guide (including YouTube)
Tutorial		
Version Control		

Table 7: Artifacts Present in LBM Packages, Classified by Frequency (Sorted Alphabetically)

Table 8 shows that MI artifacts generally match the recommendations found in nine current research software development guidelines:

- United States Geological Survey Software Planning Checklist (USGS, 2019),
- DLR (German Aerospace Centre) Software Engineering Guidelines (Schlauch et al., 2018),
- Scottish Covid-19 Response Consortium Software Checklist (Brett et al., 2021),
- Good Enough Practices in Scientific Computing (Wilson et al., 2016),
- xSDK (Extreme-scale Scientific Software Development Kit) Community Package Policies (Smith et al., 2018a),
- Trilinos Developers Guide (Heroux et al., 2008),
- EURISE (European Research Infrastructure Software Engineers') Network Technical Reference (Thiel, 2020),
- CLARIAH (Common Lab Research Infrastructure for the Arts and Humanities) Guidelines for Software Quality (van Gompel et al., 2016), and
- A Set of Common Software Quality Assurance Baseline Criteria for Research Projects (Orviz et al., 2017).

In Table 8, each row corresponds to an artifact. For a given row, a checkmark in one of the columns means that the corresponding guideline recommends this artifact. The last column shows whether the artifact appears in the measured set of LBM software, either not at all (blank), commonly (C), uncommonly (U) or rarely (R). We did our best to interpret the meaning of each artifact consistently between guidelines and specific MI software, but the terminology and the contents of artifacts are not standardized. This challenge even exists for the ubiquitous README file. As illustrated by Prana et al. (2018), the content of README files shows significant variation between projects. Although some content is reasonably consistent, with 97% of README files containing at least one section describing the 'What' of the repository and 88.5 % offering some 'How' content, other categories are more variable. For instance, information on 'Contribution', 'Why', and 'Who', appear in 27.8%, 25.7% and 52.9% of the analyzed files, respectively (Prana et al., 2018).

The frequency of checkmarks in Table 8 indicates the popularity of recommending a given artifact, but it does not imply that the most commonly recommended artifacts are the most important artifacts. Moreover, just because a guideline does not explicitly recommend an artifact does not mean that the artifact has no value. The guideline authors may have excluded it because it is out of the scope of their recommendations or outside their experience. For instance, an artifact related to uninstall is only explicitly mentioned by van Gompel et al. (2016), but other guideline authors would likely see its value. They may simply feel that uninstall is implied by install, or they may have never asked themselves about the need for separate uninstall instructions.

The top four packages (ESPREsSo, Ludwig, Palabos, and OpenLB) include most of the common and uncommon artifacts, but not many of the rare artifacts. They collectively have all the common artifacts, except Palabos does not have theory notes, and OpenLB does not appear to use version control. The top four packages have most of the uncommon artifacts. Only one them (Ludwig) is missing design information, and only one (Palabos) does not have a user manual. However, for Palabos there was a broken link on the package website indicating that such an artifact might exist. The developers later fixed this broken link, but this is not reflected in our data because it was not present at the time of data collection (mid-2020). Despite the broken link, Palabos does have a detailed and informative website. The top four ranked packages do not have many of the rare artifacts. None of them have any explicit API documentation. Three of them (ESPREsSo, Ludwig, Palabos) have information on contributing to the project and two (OpenLB, Palabos) have an FAQ section or forum. One (OpenLB) has verification and validation notes, and a video guide for the software.

Three of the items that appear in Table 7 do not explicitly appear in the software development guidelines: i) list of related publications, ii) performance information and iii) video guides. The occurrence of the first two items in this list are likely more a consequence of how we collected our list of LBM artifacts, rather than being a unique practice of LBM developers. We highlighted publications and performance information because we were explicitly looking for both as part of our measurement template. Related publications and performance information certainly appear in other research software projects; the practices just are not highlighted in guidelines because they are implicit in other documentation recommendations, such as theory documentation and user guides. The observation of a video guide artifact is more novel. Fogel (2005) recommends videos for open source projects, and other projects likely use them, but the practice has apparently not yet made it into research software community's guidelines. Putting aside the debate of the effectiveness of learning from print versus video, video documentation is growing in popularity, with a majority of Generation Z learners preferring video to print (Genota, 2018).

As Table 8 shows, the LBM community participates in most practices we found listed in the general research software guidelines; however, we did not observe, or rarely observed, some recommended practices. For instance, we rarely saw API documentation for LBM software, but it is a frequently recommended artifact (Smith et al., 2018a; Thiel, 2020; van Gompel et al., 2016; Orviz et al., 2017; Institute, 2022; Zadka, 2018). In addition to the items in the last column of Table 7, we can add the following community recommended items that we rarely, if ever, observed:

- **A roadmap** is considered by some to be a critical document for an organization to map out the vision and direction for a product offering (Münch et al., 2019). Roadmaps take different forms, but they all aim to address the following: i) Where are we now?, ii) Where are we going?, and iii) How can we get there? (Phaal et al., 2005). Developers can create a roadmap by the following steps: i) define and outline a strategic mission and product vision, ii) scan the environment, iii) revise and distill the product vision to write the product roadmap, and iv) estimate the product life cycle and evaluate the mix of planned development efforts (Vähäniitty et al., 2002). A product roadmap has the benefits of providing continuity of purpose, facilitating stakeholder

	USGS (2019)	Schlauch et al. (2018)	Brett et al. (2021)	Wilson et al. (2016)	Smith et al. (2018a)	Heroux et al. (2008)	Thiel (2020)	van Gompel et al. (2016)	Orviz et al. (2017)	LBM
LICENSE	✓	✓	✓	✓	✓		✓	✓	✓	C
README		✓	✓	✓	✓		✓	✓	✓	C
CONTRIBUTING		✓	✓	✓	✓		✓	✓	✓	U
CITATION				✓				✓	✓	U
CHANGELOG		✓		✓	✓		✓			U
INSTALL					✓		✓	✓	✓	C
Uninstall								✓		R
Dependency List			✓		✓			✓		C
Authors							✓	✓	✓	C
Code of Conduct							✓			
Acknowledgements							✓	✓	✓	R
Code Style Guide		✓					✓	✓	✓	R
Release Info.		✓				✓	✓			U
Prod. Roadmap						✓	✓	✓		R
Getting started					✓		✓	✓	✓	C
User manual			✓				✓			U
Tutorials							✓			C
FAQ							✓	✓	✓	R
Issue Track		✓	✓		✓	✓	✓		✓	C
Version Control		✓	✓	✓	✓	✓	✓	✓	✓	C
Build Scripts		✓		✓	✓	✓	✓		✓	C
Requirements		✓				✓			✓	
Design Doc.		✓	✓		✓		✓	✓	✓	U
API Doc.					✓		✓	✓	✓	R
Test Plan		✓				✓				U
Test Cases	✓	✓	✓		✓	✓	✓	✓	✓	U

Table 8: Comparison of Recommended Artifacts in Software Development Guidelines to Artifacts in LBM Projects (C for Common, U for Uncommon and R for Rare)

collaboration and assisting with prioritization (Pichler, 2012). Although we saw some elements of a roadmap in the LBM projects, we did not observe a specific artifact devoted to this purpose.

- **A code of conduct** documents expectations and standards of ethical behaviour and explicitly states how developers should treat one another (Tourani et al., 2017). A code of conduct outlines rules for communication, establishes enforcement mechanisms for violations and codifies the spirit of a community, such that anyone can contribute comfortably regardless of, for example, ethnicity, gender, or sexual orientation (Tourani et al., 2017). The top three most popular code's of conduct from Tourani et al. (2017) are: Ubuntu Code of Conduct, Contributor Covenant, and Django Code of Conduct. Singh et al. (2021) shows the value of adopting a code of conduct on the participation of women. Improved inclusivity means a wider pool of potential contributors. Moreover, a code of conduct facilitates meeting ethical obligations for developers that abide by a code of ethics, such as the IEEE Code of Ethics (IEEE-CS/ACM and Practices, 1999), or the Professional Engineers of Ontario code of ethics (Professional Engineers Act, 2021, p. 23–24). For the LBM software, we did not observe any codes of conduct, possibly because the number of developers for each project is small and because the idea has not yet occurred to them.

- **Code style guides** provide rules and guidance on writing code. Style guides standardize such elements as naming identifiers, formatting, code comments, best practices and dangers to avoid (Carty, 2020). An example of a ubiquitous coding standard is to use ALLCAPS for symbolic constants. Standardization improves understandability (Section 2.1), since developers spend more time on the content of the code, and less time distracted by its style. Three example style guides include: PEP8 Style Guide for Python, Google Java Style Guide and Google C++Style Guide. Tools like flake8 can be used to automatically check that coding styles, in this case PEP8, are enforced. For LBM software we saw code style advice as part of some developer guides, but only rarely.

- **Checklists** can be used to ensure that developers follow best practices. Some examples include checklists merging branches into master (Brown, 2015), checklists for saving and sharing changes to the project (Wilson et al., 2016), checklists for new and departing team members (Heroux and Bernholdt, 2018), checklists for processes related to commits and releases (Heroux et al., 2008) and checklists for overall software quality (Thiel, 2020; Institute, 2022). For LBM software, ESPResSo has a checklist for managing releases, but we did not see any other examples.

- **Uninstall instructions** tell a user how to remove the software and associated libraries from their system. Uninstall removes files that consume space and potentially pollute the user's build system with potential future compilation and linking conflicts. Only one software package (pyLBM) has uninstallation instructions.

- **API documentation** shows programmers the services or data provided by the software application (or library) through such resources as its methods or objects (Meng et al., 2018). API documentation is a critical factor for API understandability (Meng et al., 2018) (Section 2.1). Some API documentation can be generated using tools like Doxygen, but when learning a new API developers mostly use code examples and tutorials (Meng et al., 2018). API documentation is rare within our sample of LBM packages.

Where data is available, LBM software artifacts can be compared to other research software domains. Compared to ocean modelling software (Jung et al., 2022), LBM software seems to pay more attention to artifacts related to testing. Compared to Medical Imaging Software (Dong, 2021), the artifact frequency is similar, except LBM software is more likely to have developer related artifacts, like Contributing, Dependency list, and Design documentation.

Table 8 shows that many LBM projects fall short of recommended best practices for research software. However, LBM software is not alone in this. Many, if not most, research projects fall short of best practices. A gap exists in research software development practices and software engineering recommendations (Storer, 2017; Kelly, 2007; Owojaiye et al., 2021). Johanson and Hasselbring (2018) observe that the state-of-the-practice for research software in industry and academia does not incorporate state-of-the-art SE tools and methods. This causes sustainability and reliability problems (Faulk et al., 2009). Rather than benefit from capturing and reusing previous knowledge, projects waste time and energy "reinventing the wheel" (de Souza et al., 2019).

Software requirements documentation provides a good example of the common deviation between recommendations and practice. Although some guidelines recommend requirements documentation (Schlauch et al., 2018; Heroux et al., 2008; Smith and Koothoor, 2016), in practice research software developers often do not produce a proper requirements specification (Heaton and Carver, 2015). Sanders and Kelly (2008) interviewed 16 scientists from 10 disciplines and found that none of the scientists created requirements specifications, unless regulations in their field mandated such a document. Nguyen-Hoan et al. (2010) shows requirements are the least commonly produced type of documentation for research software in general. When looking at the pain points for research software developers, Wiese et al. (2019) state that software requirements and management is the software engineering discipline that most hurts scientific developers, accounting for 23% of the technical problems reported by study participants. The lack of support for requirements is likely due to the perception that up-front requirements are impossible for research software (Carver et al., 2007; Segal and Morris, 2008), but when we drop the insistence on "up-front" requirements, allowing the requirements to be written iteratively and incrementally, requirements are feasible (Smith, 2016).

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

Developers use software tools to support the development, verification, maintenance, and evolution of software, software processes, and artifacts (Ghezzi et al., 2003, p. 501). Table 9 summarizes the observed tools, subdivided

into development tools, dependencies, and project management tools. In this section we answer RQ5 by comparing aspects of tool usage in LBM software packages to their use in the research software community in general.

Development Tools	**Dependencies**	**Project Management Tools**
Code Editors	Build Automation Tools	Change Tracking Tools
Compilers	Domain Specific Libraries	Collaboration Tools
Continuous Integration	Technical Libraries	Document Generation Tools
Correctness Verification Tools		Email
Development Environments		Version Control Tools
Runtime Environments		
Unit Testing Tools		

Table 9: Observed development tools, dependencies and project management tools (Sorted Alphabetically)

Development tools support the development of end products, but do not become part of them, unlike dependencies that remain in the application once it is released (Ghezzi et al., 2003, p. 506). Although not shown in Table 9, developers also likely use debuggers. Only three (ESPResSo, Ludwig and Musubi) packages mention CI tools, like Travis CI. Several packages explicitly note code editors and compilers, but likely all packages use them. One of the packages (Ludwig) explicitly notes the use of proprietary unit testing code written in C. Likewise, one of the developers of pyLBM mentions the use of proprietary code for verifying the correctness of output. Developers of other packages likely use similar tools.

For the dependency tools (Table 9), we observe that most of the software packages use some sort of build automation tools, most commonly Make. They all use various technical and domain specific libraries. Technical libraries include visualization (e.g., Matplotlib, ParaView, Pygame, VTK), data analysis (e.g., Anaconda, Torch), and message passing libraries (e.g., MPICH, Open MPI, PyZMQ). Domain specific libraries include research software libraries (e.g., SciPy).

Many of the software packages have teams of two or more. Their work needs to be coordinated and managed. Table 9 shows the types of project management tools that were explicitly noted in the artifacts, webpages, or interviews with the developers. As with development tools and dependencies, developers may use other types of project management tools, but they were not visible in the artifacts to which we had access. For collaboration tools the most often observed are email and video. Our interviewees did not mention project management software, but our sample is not large enough to rule out other projects using such software. GitHub is the home for many of the projects. Developers use the platform to help manage their projects, especially bug related issues. Most of the projects appear to use change tracking and version control tools. Half of the packages mention document generation tools, like Sphinx and Doxygen.

Based on information provided by Jung et al. (2022), tool usage for LBM software has much in common with tool usage for ocean modelling software. Both use tools for editing, compiling, code management, testing, building, and project management. From the data available, ocean modelling differs from LBM in its use of Kanban boards for project management. LBM software differs from the ocean modelling through the use of continuous integration, and document generation.

6.1. Version Control Usage Compared to Research Software in General

The poor adoption of version control tools that Wilson lamented in 2006 (Wilson, 2006) has greatly improved in the intervening years. From Section 3.7, 67% of LBM packages use version control (GitHub, GitLab, or CVS). If we restrict our attention to alive packages, Table 4, shows 11/15 or 73% of alive packages use version control. The proliferation of version control tools for LBM matches the increase in the broader research software community. Nguyen-Hoan et al. (2010) estimated a little over 10 years ago that only 50% of research software projects use version control, but even at that time they noted an increase from previous usage levels. A survey in 2018 shows 81% of developers use a version control system (AlNoamany and Borghi, 2018). Smith (2018) has similar results, showing version control usage for alive projects in mesh generation, geographic information systems and statistical

software for psychiatry increasing from 75%, 89% and 17% (respectively) to 100%, 95% and 100% (respectively) over a 4-year period ending in 2018. (For completeness the same study showed a decrease in version control usage for seismology software over the same time period, from 41% down to 36%). Almost every software guide cited in Section 5 includes the advice to use version control. The high usage of version control tools in LBM software matches the trend in research software in general.

6.2. CI Usage Compared to Research Software in General

As mentioned in Section 3.2, LBM developers rarely uses CI (3 of 24 packages or 12.5%), although interviews with developers suggest that the actual rate might be higher. Low use of CI for LBM contrasts with how frequently it is recommended in research software development guidelines (Brett et al., 2021; Brown, 2015; Thiel, 2020; Zadka, 2018; van Gompel et al., 2016) and its popularity with non-research software. Hilton et al. (2016) show CI usage across a large sample of GitHub projects (34,544 projects) is at least 40%, with more popular projects (as measured by stars), having rates up to 70%. Moreover, Hilton et al. (2016) predict that CI adoption rates for open-source projects will increase even further in the future. We could not find published data on the CI usage rates for research software in general. However, we saw 17% utilization in the specific domain of imaging software (Dong, 2021), which is comparable to the LBM usage rate. Our impression is that medical imaging software and LBM software are not unique — most research software domains likely lag behind the best practice recommendations, and non-research software usage rates, for CI.

Continuous integration is the process of building and testing software on every push to the code repository, with pushes done very frequently (Humble and Farley, 2010, p. 13), (Shahin et al., 2017; Fowler, 2006). CI consists of the following elements:

- A version control system (Fowler, 2006). To be effective, all files should be under version control, not just code files. Anything that is needed to build, install and run the software should be under version control, including configuration files, build scripts, test harnesses, and operating system configurations (Humble and Farley, 2010, p. 19). Fortunately for the LBM domain, as discussed above, most alive projects (73% of those assessed) use version control.

- A fully automated build system (Fowler, 2006). As Humble and Farley (2010, p. 5) point out, deploying software manually is an anti-pattern. Fortunately, as discussed above, most LBM projects use build automation.

- An automated test system (Fowler, 2006). Building quality software involves creating automated tests at the unit, component, and acceptance test level, and executing these tests whenever someone makes a change to the code, its configuration, the environment, or the software stack that it runs on (Humble and Farley, 2010, p. 83). As Table 7 shows, test cases are in the uncommon category for LBM software artifacts, which means that some LBM projects will need to increase their testing automation if they wish to pursue CI. The usual advice for CI is to keep the build and test process short (Humble and Farley, 2010, p. 60). Given that LBM can be computationally expensive, the tests run with every check-in may need to focus on simple code interface tests, saving large tests for less frequent runs. A more sophisticated option to address the bottleneck for merges is CIVET (Continuous Integration, Verification, Enhancement, and Testing), which solves this problem by intelligently pinning, cancelling, and if necessary, restarting jobs as merges occur (Slaughter et al., 2021).

- An automated system for other tasks, such as code checking, documentation building and web-site updating. These other tasks are not essential to CI, but they can be incorporated to improve the quality of the code and the communication between developers and users. For instance, a static analysis (possibly via linters) of the code may find poor programming practice or lack of adherence to adopted coding standards. Slaughter et al. (2021) provides another example of an automated task — checking that a test specification includes the test's motivation, a test description, and a design description for all changes.

- An integration build system to pull everything together. Every time there is a check-in (for instance a pull request), the integration server automatically checks out the sources onto the integration machine, starts a build, runs tests, and informs the committer of the results.

Although CI can take some time and effort to set up and integrate into a team's workflow, the benefits can be significant, as follows:

- Elimination of headaches associated with a separate integration phase (Fowler, 2006), (Humble and Farley, 2010, p. 20). If integration of the work of different developers, or even separate chunks of work by the same developer, is postponed, integration problems are inevitable. By continuously integrating, problems are immediately obvious and the source of the problem can be isolated to the small increment that was just committed.
- Bugs can be quickly detected and removed (Fowler, 2006) via automated testing. To improve productivity, defects are best discovered and fixed at the point where they are introduced (Humble and Farley, 2010, p. 23). Code is not the only source of errors; they are also found in the files and scripts related to configuration management (Humble and Farley, 2010, p. 18).
- Developers are always working on a stable base, since the master branch will always be working. A stable base passes all tests and, if the CI system uses generators and linters, it will also have current documentation and coding standard compliant code. A stable base improves developer productivity, allowing them to focus on coding, testing, and documentation.

Although there are still challenges, setting up a CI system has never been easier than it is today. The option of installing a dedicated CI server (either physically or virtually) exists with tools such as Jenkins, Buildbot, Go, and Integrity. However, installation on your own server is unnecessary since there are many hosted CI tool solutions available, such as: Travis CI, GitHub Actions and CircleCI. Getting started with a hosted CI is straightforward. All that is required to begin is editing a few lines of a YAML configuration file in the project's root directory.

Challenges that exist for adopting CI include lack of awareness and transparency, coordination and collaboration challenges, lack of expertise and skills, more pressure and workload for team members, general resistance to change, scepticism and distrust on continuous practices (Shahin et al., 2017). The most common reason given for not adopting CI is that developers on my project are not familiar enough with CI (Hilton et al., 2016). These problems can be mitigated via planning and documentation, promoting a team mindset, adopting new rules and policies, improving testing activities, and decomposing development into smaller units (Shahin et al., 2017).

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

We answer research question RQ6 by comparing the principles, processes, and methodologies used for developing LBM software to those used for research software in general. Our measured data on LBM software artifacts does not explicitly indicate the development process. However, during our 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. Although eleven of the packages explicitly convey that they would accept outside contributors, teams are generally centralized, often working at the same institution. Working at the same institution means that an informal process can show success, since informal conversations are relatively easy to have.

Our interviews with LBM developers confirmed a similar project management process. In teams of only a couple of developers, the entire team discusses additions of new features or major changes. Projects with more than a couple developers have lead developer roles. These lead developers review potential additions to the software. One of the developers (ESPResSo) noted that they use an ad hoc peer review process to assess major changes and additions. Using peer review (also called technical review) matches with recommended practice for research software (Heroux et al., 2008; Givler, 2020; Orviz et al., 2017; USGS, 2019).

During our interviews, we discussed two types of software changes. One is feature additions, which arise from a scientific or functional need. These changes involve formal discussions within the development team, and lead developer participation is mandatory. The other change type is code refactoring, which only sometimes involves formal discussions with the development team. We learned that new developers play a larger role with these changes compared to the former changes. Developers address software bugs similarly to code refactoring, and they manage change via issue tracking.

Our observations of an informally defined process, with elements of agile methods, matches what has been observed for research software in general. Scientific developers naturally use an agile philosophy (Ackroyd et al., 2008; Carver et al., 2007; Easterbrook and Johns, 2009; Segal, 2005; Heaton and Carver, 2015), an amethododical process (Kelly, 2013), or a knowledge acquisition driven process (Kelly, 2015).

7.1. Design Principles and Priorities

Most of the software packages do not explicitly state the motivations or design principles that were considered when developing the software. One package, Sailfish, indicates in its artifacts the explicit goal of shortening development time, with the developers using Python and CUDA/OpenCL to achieve this without sacrificing performance. The Sailfish documentation explicitly lists the goals of performance, scalability, agility and extendability, maintenance, and ease of use. The project scored well in these categories during our assessment. The quality priorities for Sailfish roughly match the priorities observed for research software in general. Nguyen-Hoan et al. (2010) surveys developers to find the following list of qualities, in decreasing order of importance: reliability, functionality, maintainability, availability, performance, flexibility, testability, usability, reusability, traceability, and portability. The Sailfish list does not list reliability or functionality, but we can safely assume those are implicitly high priorities for any scientific project. In earlier studies Kelly and Sanders (2008) and Carver et al. (2007) highlight how important correctness is for research software.

7.2. Documentation as Part of the Development Process

During our interviews, developers noted that documentation plays a significant role in the development process, specifically with on-boarding new developers. A goal of documentation is to lower the entry barrier for these new contributors. The documentation provides information on how to get started, orients the user to artifacts and the source code, and explains how the system works, including the so-called simulation engine and interface. This emphasis on documentation, especially for new developers, is echoed in research software guidelines summarized in Table 8. For open source software in general (not just research software), Fogel (2005) recommends providing tutorial style examples, developer guidelines, demos, and screenshots.

Although Table 8 shows that design documentation is frequently suggested, the recommendations are often short on details. For instance, the guidelines usually encourage modular design, but are generally silent on what criteria to use for the modularization. The guidelines give little information on how to document the design itself. Some non-mutually exclusive ideas for documenting the design, roughly in order of increasing effort, are as follows:

- Explicitly state the design goals and priorities, so that those using and developing your software know part of the rationale for the design decisions. Sailfish is an example where this has been done, as discussed in Section 7.1.
- Write down the likely changes that the design is intended to support. Since most designs are implicitly (or explicitly) motivated to facilitate future change, recording the likely changes can help readers understand the design.
- Explicitly record the philosophy or principle behind the decomposition. One potential principle, related to the previously mentioned list of likely changes, is the principle of information hiding (Parnas, 1972). This principle supports design for change through the "secrets" of each module. A text blurb can be given for each module or class to describe its secrets and the services it provides.
- Rigorously document the design following the template for a Module Guide (MG) (Parnas et al., 1984). An MG organizes the modules in a hierarchy by their secrets. The application of the Parnas approach to research software has been illustrated by applying it to the example of a mesh generator (Smith and Yu, 2009).
- Graphically represent the design using Unified Modelling Language class diagrams. This approach is suited to object-oriented design and designs that use patterns (Gamma et al., 1995).
- Explain the design using data flow diagrams to show typical use cases for input transformation.
- Using a textual description, list for each module (or class), the state variables (if any), exported constants and all exported access programs. This shows the interface that can used to access each module's services.

- Document the Module Interface Specification (MIS) (Hoffman and Strooper, 1995). An MIS is an abstract model that formally shows each module's access programs and the associated transitions and outputs based on their state, environment, and input variables (ElSheikh et al., 2004; Smith and Yu, 2009).

8. Developer Pain Points

Based on interviews with four developers, this section aims to answer the research questions: i) What are the pain points for developers working on research software projects (RQ7)?; and, ii) How do the pain points of developers from LBM compare to the pain points for research software in general (RQ8)? Below we go through each of the identified pain points and include citations that contrast the LBM experience with observations from researchers in other domains. Section 9 covers potential ways to address the pain points and Anonymous (xxxx)[1] provides the full interview questions.

Pinto et al. (2018) lists some pain points that did not come up in our conversations with LBM developers: Cross-platform compatibility, interruptions while coding, scope bloat, lack of user feedback, hard to collaborate on software projects, and aloneness. Wiese et al. (2019) repeat some previous pain points and add the following: dependency management, data handling concerns (like data quality, data management and data privacy), reproducibility, and software scope determination. Although LBM developers did not mention these pain points, we cannot conclude that they are not relevant for LBM software development, since we only interviewed four LBM developers for about an hour each.

P1: **Lack of Development Time:** A developer of pyLBM noted that their small development team has a lack of time to implement new features. Small development teams are common for LBM software packages (as shown in the measurement table excerpt in Figure 2). Other domains of research software also experience the lack of time pain point (Pinto et al., 2018, 2016; Wiese et al., 2019).

P2: **Lack of Software Development Experience:** The developer of TCLB noted a lack of software development experience, and others noted a need for improving software engineering education. Many of the team members are domain experts, not computer scientists or software engineers. Nguyen-Hoan et al. (2010) noted this same trend with research software developers, with only 23% of survey respondents having a computing-related background. Similarly, Nangia and Katz (2017) show that the majority (54%) of postdocs have not received training in software development. The LBM developer suggesting an increasing role for formal software education matches the trend observed by Pinto et al. (2018), where their replication of a previous study (Hannay et al., 2009), shows a growing interest in formal training (from 13% of respondents in 2009 to 22% in 2018). Pinto et al. (2018) found that some developers feel there is a mismatch between coding skills and subject-matter skills.

P3: **Lack of Incentive and Funding:** The TCLB developer noted a lack of incentives and funding in academia for developing widely used research software. This problem has also been noted by others (Gewaltig and Cannon, 2012; Goble, 2014; Katerbow and Feulner, 2018). Wiese et al. (2019) reported developer pains related to publicity, since publishing norms have historically made it difficult to get credit for creating software. As studied by Howison and Bullard (2016), research software (specifically biology software, but the trend likely applies to other research software domains) is infrequently cited. Pinto et al. (2018) also mentions the lack of formal reward system for research software.

P4: **Lack of External Support:** Interviewees raised a concern that there are no organizations helping with the development of good quality software. The literature on research software does not echo this concern because there are such organizations, including Better Scientific Software (BSSw), Software Sustainability Institute (Crouch et al., 2013), and Software Carpentry (Wilson and Lumsdaine, 2006; Wilson, 2016). Over time awareness of these groups will grow, so this pain point is likely to disappear in the future for LBM and other research software developers.

P5: **Technology Hurdles:** Technology pain points for LBM developers include setting up parallelization and CI. The pain point survey of Wiese et al. (2019) also highlighted technical-related problems like dependency management, cross-platform compatibility, CI, hardware issues and operating system issues.

P6: **Ensuring Correctness:** Several developers noted difficulties with ensuring correctness. They alluded to difficulty with testing the correctness of large numbers of features, and challenges with automated testing. The TCLB developer commented that the amount of testing data needed was sometimes problematic, since free testing services do not offer adequate facilities for large amounts of data, requiring in-house testing solutions. Other research software domains point to the following problems with testing: i) Pinto et al. (2018) mention the problem of insufficient testing; ii) Hannay et al. (2009) show that more developers think testing is important than the number that believe they have a sufficient understanding of testing concepts; and, iii) Hannay et al. (2009); Kanewala and Bieman (2013); Kelly et al. (2011); Wiese et al. (2019) point to the oracle problem, which occurs in research software when we do not have a means to judge the correctness of the calculated solutions. Based on the interviews, the LBM experience seems to overlap with i and ii; the developers did not allude to problem iii (the oracle problem). As discussed in Section 9.2, the oracle problem likely did not come up because the LBM developers have strategies for verifying their work.

P7: **Usability:** Several developers noted that users sometimes try to use incorrect LBM method combinations to solve their problems. Furthermore, some users think that the packages will work out of the box to solve their cases, while in reality CFD knowledge needs to be applied to correctly modify the packages for the new endeavour. Some respondents in the survey of Wiese et al. (2019) also mentioned that users do not always have the expertise required to install or use the software.

P8: **Technical Debt:** The developer of ESPResSo said that they originally wrote their source code with a specific application in mind, which later caused too much coupling between components in the source code. This resulted in technical debt (Kruchten et al., 2012), which has an impact on future maintainability and reusability. Concern with technical debt is likely why researchers in the survey of Nguyen-Hoan et al. (2010) rate maintainability as the third most important software quality. More recently the push for sustainable software (de Souza et al., 2019) is motivated by the pain that past developers have had with accumulating too much technical debt.

P9: **Quality of Documentation:** Interviewees stressed the importance of documentation for both users and developers throughout the interviews. They emphasize that a lack of time and funding (P1) has a negative effect on the documentation. Most of the developers are scientific researchers evaluated on the scientific papers that they produce. Writing and updating documentation is something that is done in their free time, if that time arises. Others also mention inadequate research software documentation (Pinto et al., 2018; Wiese et al., 2019). The problem also arises with non-research software (Lethbridge et al., 2003). Recommendations on the development of research software state that developers should critically evaluate their own development processes in terms of quality assurance and comply with international standards for software documentation (Katerbow and Feulner, 2018).

9. Lessons from LBM Developers

This section answers RQ9 by summarizing the best practices LBM developers take to address the pain points mentioned in Section 8. The main source of information is the qualitative data from developer interviews (Section 2.7). The practices summarized in this section can potentially be emulated by the LBM software packages that do not currently follow them. Moreover, these practices may also provide examples that can be followed by other research software domains.

9.1. Design For Change

To address technical debt (P8), the top LBM developers show how modularization can be used to design research software for future change. Although the advice to modularize research software to handle complexity is common (Wilson et al., 2014; Stewart et al., 2017; Storer, 2017), specific guidelines on how to divide the software into modules is less prevalent. Not every decomposition is a good design for supporting change, as shown by Parnas (1972). A design with low cohesion and high coupling (Ghezzi et al., 2003, p. 48) will make change difficult. Especially in research software, where change is inevitable, designers need to produce a modularization that supports change. Ocean modelling software is currently feeling the pain of not emphasizing modularization in legacy code (Jung et al., 2022).

Interviewed LBM developers highlighted cases where their modularizations anticipated future changes. The developer of pyLBM mentioned that the geometries and models of their system had been "decoupled", using abstraction and modularization of the source code, to make it "very easy to add [new] features". The pyLBM design allows for independent changes to the geometry and the model. We also learned that the package pyLBM redeveloped data structures to ease future changes. The developer of TCLB noted that their design allows for the addition of some LBM features, but changes to major aspects of the system would be difficult. For example, "implementing a new model will be an easy contribution", but changes to the "Cartesian mesh ... will be a nightmare". The design of TCLB highlights that not every conceivable change needs to be supported, only the likely changes.

As the LBM developers illustrate, they accomplish design for change by first identifying likely changes, either implicitly or explicitly, and second by hiding each likely change behind a well-defined module interface. Although it is unclear whether the developers are aware of Parnas's work, the approach mirrors his recommendations (Parnas, 1972). Section 7.2 lists ideas for how to document the design, including the likely changes, so that they are more visible to others.

9.2. Circumventing the Oracle Problem and Addressing Reproducibility

The top LBM projects have adopted techniques for software verification that get around the oracle problem (mentioned earlier in P6). In the absence of a general test oracle, the following techniques have been adopted by LBM developers:

- Compare the calculated results against manually calculated results in the cases where manual solutions can be determined. Although it is not always feasible to calculate results a priori, developers can find sub-sets of the general problem where answers can be determined. In special cases that allow this, the results can be compared against exact analytical solutions, as done by OpenLB.
- Related to peer review, verify the mathematical foundations of the models to build confidence in the software's theoretical underpinnings.
- Compare the calculated results against pseudo-oracle solutions for verification benchmarks. The developer of ESPResSo mentioned the value of "tests which compare implementations against each other" where the tests "rely on different principles, and they've been written by different people". Latt et al. (2021) shows several benchmark tests comparing Palabos solutions to other software solutions, such as solutions calculated using AnSys.
- Use peer review of the code for major changes and additions, as mentioned in Section 7 for ESPResSo.

Section 10 provides additional strategies for getting around the oracle problem.

Some LBM developers improve reproducibility via automating the verification tests and making them part of a CI process (mentioned in Section 6). Providing unit tests improves reproducibility because future developers will require a means to verify that they can obtain the same computational results as today. Unit tests are a good way to embed self-diagnostics of reproducibility in the code (Benureau and Rougier, 2017).

9.3. Prioritize Documentation and Usability

Although documentation is difficult (as mentioned under pain point P9), some LBM developers have realized that prioritizing time on documentation provides a justifiable return on investment. For instance, the interviewed developer of ESPResSo said that all major additions to their package had accompanying changes to artifacts and documentation. They put considerable time into their documentation. They further commented that their goal was to lower the entry barrier for new developers, which meant providing considerable developer documentation. This documentation informs developers on how to get started; orients them to the artifacts, source code, system architecture, and build system; and, explains the coupling between the simulation engine and the interface. In addition, ESPResSo developers included API documentation, although they are the only one of the top five packages to do so.

Furthermore, LBM developers use documentation to address usability challenges (P7), as follows:

- Explicitly state appropriate fluid dynamics problems that the software uses in its models and explicitly state the software's limitations. This is done in the ESPResSo user guide.

- Provide documentation that details the background theory, or provide a reference to such information. This was done by several top ranked packages, including ESPResSo, Ludwig, LUMA, and OpenLB.
- Identify expected user characteristics. LIMBES, Ludwig, laboetie, and Palabos all did this. Explicit user characteristics are important so the application and documentation can be pitched toward the right user background (Smith et al., 2007).
- Keep all documentation in one location. This was done by all top ranked packages.
- Use documentation generators, like Doxygen, as done (for instance) by OpenLB. Sections 3.2 and 6 give more details on the packages that use generators.
- Provide an explanation of theoretical principles through the user interface.
- Add guards in the code to test user inputs for compatibility before proceeding with the calculations.
- Include an FAQ (Section 5).

10. Recommendations for Future Practices

In this section we provide recommendations to address the pain points from Section 8 to answer RQ10. Our recommendations are not lists of criticisms for what should have been done in the past, or what should be done now; they are suggestions for consideration in the future. We will not be repeating previously discussed ideas here, such as CI, documentation of APIs, etc. Our aim is to mention ideas that are at least somewhat beyond conventional best practices. The ideas listed here have the potential to become best practices in the medium to long-term. The ideas in the following subsections are roughly in the order of increasing implementation effort.

10.1. Employ Linters

Except for Thiel (2020), the research software guidelines that we consulted do not mention linters. However, we believe that linters have the potential to improve code quality at a relatively low cost. A linter is a tool that statically analyzes code to find programming errors, stylistic inconsistencies and suspicious constructs (Wikipedia, 2022). Linters can be used to spot check code files, or even better as part of a continuous integration system, as discussed in Section 6.2.

A sample of the potential benefits of linters include: finding memory leaks, finding potential bugs, standardizing code with respect to formatting, improving performance, removing silly errors before code reviews, and catching potential security issues (SourceLevel, 2022). Linters are available for most popular programming languages. For instance Python has the options of PyLint, flake8 and Black (Zadka, 2018).

Linters can address the pain points for LBM developers. For instance, a linter can decrease the amount of development time (P1) by decreasing the number of mundane mistakes programmers have to catch. Since the linter can include rules that capture the wisdom of senior programmers, it can help newer developers avoid common mistakes (P2). Although there are not many linters for checking parallel programming mistakes (P5), Parallel Lint is an option for OpenMP. Consistently using a linter can guard against some technical debt (P8). Although linters are tools for code analysis, similar ideas can be applied to "lint" documentation to ensure adherence to basic rules. The use of tools to check documentation is a partial explanation for the relatively higher quality of statistical tools that are part of the Comprehensive R Archive Network (CRAN) (Smith et al., 2018d).

10.2. Conduct Rigorous Peer Reviews

Peer review is a strategy already adopted by some LBM developers to improve confidence in their software (Section 9.2), but the benefits of peer review can be improved by making its application more rigorous. Jones (2008) shows that rigorous inspection finds 60-65% of latent defects on average, and often tops 85% in defect removal efficiency. By way of comparison, most forms of testing average between 30 and 35% for defect removal efficiency (Ebert and Jones, 2009; Jones, 2008). A formal code inspection involves asking the reviewer to either follow a review checklist (check consistency of variable names, look for terminating loops, etc.), or perform specific review tasks (like summarize the

purpose of the code, create a data dictionary for a given module, cross-reference the code to the technical manual, etc.) The task based inspection approach has been effectively used for research software, as described by Kelly and Shepard (2000). Task based inspection is an ideal fit with an issue tracking system, like GitHub. The review tasks can be issues, so that they can be easily assigned, monitored and recorded. Other potential issues for the tracker include assigning junior team members to test installation instructions and getting-started tutorials.

Rigorous peer review addresses the same pain points as linters (Section 10.1): P1, P2, P5, and P8. Peer review addresses these points by efficiently searching for defects and problems. Finding misunderstandings in of how the code implements the required theory improves the software's correctness (P6).

10.3. Write and Submit More Papers on Software

To address the pain point of a lack of funding (P3), LBM developers should watch for new opportunities to publish their research software source code. For instance, the Journal of Open Source Software (JOSS) (Smith et al., 2018c) reviews and publishes articles that credit the scholarship contained in the software itself. Smith et al. (2016a) presents a set of software citation principles that may encourage broad adoption of a consistent policy for software citation across disciplines and venues. Chue Hong et al. (2019) provides a software citation checklist for developers to make a release of their software citable. Katz et al. (2021) provides further guidance on software citation. Another option is to use GitHub for citation, since it is relatively easy to add metadata to repositories and to generate citations for those repositories (Smith, 2022).

10.4. Follow Advice From the Open-Source Community to Grow the Number of Contributors

In our interviews with developers we heard they would like to have additional software contributors. More developers would help with addressing the lack of development time pain point (P1). In investigating advice on increasing the number of contributors, we found that the advice usually starts with following best practices, including providing artifacts like those discussed in Section 5, such as a contributor guidelines, code style guidelines, a clear code of conduct and high quality code and documentation. This advice is logical, but in our assessment of LBM software we found examples of projects that already follow many best practices, but still have small development teams. Attracting developers apparently requires more than just building a good product.

Some potential additional ideas for growing the number of contributors are as follows:

- Clearly identify issues that are an appropriate starting points for new developers (Garcia, 2016; Jalan, 2016; Proffitt, 2017).
- Given that in most projects developers were once users (McQuaid, 2018), recruit future developers from the current set of users.
- Create templates for pull requests and issues (Jalan, 2016).
- Welcome all kinds of contributions, not just code. Non-code contributions include documentation, fixing typos, issue reporting and test cases (Jalan, 2016; Proffitt, 2017).
- Reward and recognize new contributors, via small rewards like stickers or shirts, or even just a simple shoutout or mention in a blog post or on social media (Jalan, 2016; Proffitt, 2017).
- Recognize that open source is more about people than it will ever be about code (Jalan, 2016).
- Look beyond just recruiting online and seek new developers at conferences or other user meet ups (Garcia, 2016).
- Follow the advice of Kuchner (2012) to adapt ideas from marketing to promote science. Specific ideas that could be applied to open source projects include the following:
  - Storytelling to motivate interest in the project, where the story is a sequence of events and pauses for reflection (Kuchner, 2012, p. 21–22).
  - Invest effort in building relationships with users and with potential future developers.

  - Brand your project with what makes it unique and special.

- Consider combining existing projects to pool collective resources, and reduce competition between different open source projects.

10.5. Further Address the Oracle Problem

To address the pain point of ensuring correctness (P6), Section 9.2 outlined techniques that LBM developers use to circumvent the oracle problem for testing. Additional techniques to build confidence in correctness are as follows (Smith, 2016):

- Create test cases by assuming a solution and using this solution to calculate the necessary inputs. For instance, for a linear solver, if A and x are assumed, b can be calculated as $b = Ax$. Following this, $Ax^* = b$ can be solved and then x and x^*, which should theoretically be equal, can be compared. In the case of solving Partial Differential Equations (PDEs), the name for this approach is the Method of Manufactured Solutions (Roache, 1998).

- Use interval arithmetic to build test cases. For testing purposes, the slow, but guaranteed correct, interval arithmetic (Hickey et al., 2001) can be employed. Analysts can then verify the faster floating point algorithm by ensuring that the calculated answers lie within the guaranteed bounds.

- Include convergence studies. The discretization used in the numerical algorithm should be decreased (usually halved) and the change in the solution assessed. This is very likely already done by LBM developers, but it did not come up during our interviews.

- Use *metamorphic* testing, which checks a program to see if it satisfies a set of metamorphic relations, where a metamorphic relation relates multiple inputs and output pairs (Kanewala and Lundgren, 2016). An example could be the relationship that the output flow rate increases as the input flow rate increases.

10.6. Augment Theory Manuals to Include Requirements Information

As discussed in Section 5, developers rarely write requirements specifications for research software. Although a full requirements specification has advantages (Smith et al., 2007; Smith and Lai, 2005), given the lack of development time (P1), we recommend the intermediate step of augmenting the existing theory manuals with some requirements information. Table 7 shows that theory documentation is common. We observed 17 of 24 packages had at least some theory documentation. Templates for scientific requirements, like Smith et al. (2007); Smith and Lai (2005), show theory is a significant part of the requirements documentation. With the addition of some extra information, the theory documents can be transformed into requirements specification. The key extra information includes explicitly stating user characteristics, explicitly stating how the user interacts with the software in terms of input data requirements, and listing likely and unlikely changes. Explicit statements about likely changes are invaluable in the design stage, since they provide developers guidance on how general the software needs to be. For those wishing to maximize the value of a requirement specification, information could be added like prioritizing the nonfunctional requirements (to show the relative importance between qualities like portability, reliability, and performance) and traceability information (to show the consequences of changes to the assumptions). Spending time to incorporate additional information in the theory manuals should improve the software's usability (P7) and help address the quality of existing documentation (P9).

10.7. Improve Re-runnability, Repeatability, Reproducibility, and Replicability

Developers identified ensuring correctness as a pain point (P6). Inspired by Benureau and Rougier (2017), we make recommendations to ensure that correctness can be maintained into the future by prioritizing four increasingly sophisticated levels of replication.

Re-runnability means that today's code can be run again in the future when needed. Re-runnability becomes more difficult as code ages and the software and hardware infrastructure around it changes, like changes to compilers, GPUs, libraries etc. To be re-runnable on other researchers' computers, a code should provide a detailed description of the execution environment in which it is executable. Besides recording details, virtual machines, docker and environment managers, like Conda, can be used to improve re-runnability.

Repeatability means producing the same output over successive runs of a program. As pointed out during our developer interviews, this can be difficult for LBM software because of dependence on probability distributions. However, deterministic output is important for testing purposes and for debugging problematic code. For LBM software repeatability will likely require running a serial version of the code with full control of the initialization of the pseudo-random number generators.

Reproducibility Once we achieve re-runnability and repeatability, the next step is reproducibility. Section 9.2 discusses current work by LBM developers on reproducibility. The proposed goal for the future is for all results to be reproducible, which means documenting dependencies, platforms, parameter values and input files. The data and scripts behind the graphs should be published (Benureau and Rougier, 2017). Piccolo and Frampton (2016) summarizes some tools and techniques for computational reproducibility.

Replicability means that the documentation provided for the software, not the software itself, is sufficient for an independent third party to reproduce the computational results. Reproducibility does not help us if our trust in the original code is in doubt. What if we need to reproduce the results not starting from the code, but starting from the original theory? Unfortunately, multiple examples exist (Crick et al., 2014; Ionescu and Jansson, 2012) where results from research software could not be independently reproduced, due to a need for local knowledge missing from published documents. Examples of missing knowledge includes missing assumptions, derivations, and undocumented modifications to the code. In the future, we wish for it to be easier to independently replicate the work of others, starting from their theory documentation, augmented with requirements information as discussed above, without needing to rely on their code.

10.8. Generate All Things

We propose automatically generating LBM code and its documentation, using a Domain Specific Modelling (DSM) approach. DSM means creating a knowledge base of models for physics, computing, mathematics, documentation, and certification and then writing explicit "recipes" that weave together this knowledge to generate the desired code, documentation, test cases, inspection reports and build scripts. This definition of DSM is more general than usual; in this recommendation DSM implies generation of all software artifacts, not just the code. DSM moves development to a higher level of abstraction; domain experts can work without concern for low-level implementation details. Using DSM, we optimally generate code and documentation, eliminate redundancy, reduce the likelihood of errors, and automate maintenance. Moreover, a generative approach facilitates the inevitable exploration of LBM modelling assumptions.

DSM provides a transformative technology for documentation, design, and verification (Johanson and Hasselbring, 2018; Smith, 2018). DSM allows scientists to focus on their science, not software. A generative approach removes the maintenance nightmare of documentation duplicates and near duplicates (Luciv et al., 2018), since developers capture knowledge once and transform it as needed. Code generation has previously been applied to improve research software. For instance, ATLAS (Automatically Tuned Linear Algebra Software) (Whaley et al., 2001) and Blitz++ (Veldhuizen, 1998) produces efficient and portable linear algebra software. Spiral (Püschel et al., 2001) uses software/hardware generation for digital signal processing. Carette and Kiselyov (2011) shows how to generate a family of efficient, type-safe Gaussian elimination algorithms. FEniCS (Finite Element and Computational Software) uses code generation when solving differential equations (Logg et al., 2012). Ober et al. (2018) and Matkerim et al. (2013) apply DSM to High Performance Computing (HPC), using UML (Unified Modelling Language) for their domain models. Unlike previous DSM work, the current recommendation focuses on generating all software artifacts (requirements, design, etc.), not just code. Szymczak et al. (2016) presents initial work on this "generate all things" approach, Smith and Carette (2021) presents a motivating example, and (Carette et al., 2021) provides a prototype.

A DSM approach addresses multiple pain points. For instance, once the infrastructure is in place, a DSM can decrease the amount of development time (P1) by automation. Since a DSM approach allows scientists to focus on their science, rather than software, the lack of software development experience (P2) will be less of an issue. The DSM approach can capture computing knowledge to mitigate the technology related pain points (P5). For ensuring correctness (P6) a generative approach can aim to be correct by construction. If there are mistakes, DSM has the advantage that they are propagated throughout the generated artifacts, which greatly increases the chance that someone will notice the mistake. Technical debt (P8) is not a concern with a DSM approach since developers write the recipes

used for generation at a high level making them relatively easy to change. With respect to pain points on usability (P7) and documentation (P9), the generative approach we are proposing addresses these directly, since documentation is a primary concern, rather than an afterthought.

11. Threats To Validity

Our categories for threats to validity come from an analysis of software engineering secondary studies by Ampatzoglou et al. (2019), where a secondary study analyzes the data from a set of primary studies. We use the classification of Ampatzoglou et al. (2019) because a common example of a secondary study is a systematic literature review. Our methodology is essentially a systematic software review — the primary studies are the software packages, and our work collects and analyzes these primary studies.

11.1. Reliability

A study is reliable if its data and analysis are independent of the specific researcher(s) doing the study. That is, repetition of the study by a different researcher should achieve the same results, as long as they use the same methodology (Runeson and Höst, 2009). For our study the identified reliability related threats include the following:

- In our methodology, one individual does the manual measures for all packages. The grader's abilities, experiences, and biases may influence the results.
- Completing the manual measures of all packages took place over several months. During this time the contents of the software repositories may change and the reviewer's judgment may drift.

We investigated the risk of reviewer dependence in previous work (Anonymous, xxxx)[1], where we demonstrated that the measurement process is reasonably reproducible by having five products graded by a second reviewer. The ranking via this independent review was almost identical to the original ranking. As long as each grader uses consistent definitions, the relative comparisons in the AHP results will be consistent between graders.

11.2. Construct Validity

Construct validity means that adopted metrics represent what they are intended to measure (Runeson and Höst, 2009). The construct validity related threats for the LBM study include the following:

- We measure our software qualities indirectly, rather than directly. Unfortunately meaningful direct measures for qualities like maintainability, reusability and verifiability, are not available. Therefore, we follow the usual approach of assuming that following procedures and adhering to standards yields a higher-quality product (van Vliet, 2000, p. 112).
- The measure of surface robustness (Section 3.4) is only based on two pieces of data per project. The measurement of robustness could be expanded, as it currently only measures the impact of unexpected input. Other intentional faults could be introduced, but this would require a larger investment of measurement time.
- We may have inaccurately estimated maintainability by assuming a higher ratio of comments to source code improves maintainability (Section 3.7).
- Reusability (Section 3.8) is assessed via the number of code files and LOC per file. While this measure is indicative of modularity, it is possible that some packages have many files, with few LOC, but the files do not contain source code that is easily reusable. The formatting may be poor, or the source code may be vague and have ambiguous identifiers.
- The measurement of understandability (Section 3.9) relies on 10 random source code files. By random chance, the 10 files that were chosen may not be representative.
- Our overall AHP ranking makes the unrealistic assumption of equal weighting between qualities (Section 3.11).
- Our approximation of popularity by stars and watches (Section 4) may not be valid.

11.3. Internal Validity

A study is internally valid if the discovered causal relations are trustworthy and cannot be explained by other factors (Runeson and Höst, 2009). The internal validity threats include the following:

- We may have missed a relevant software package in our search for LBM software packages (Section 2.3).
- A temporal gap exists between the manual measurement of Musubi and the measurement of all other packages (Section 2.4). This gap caused a mismatch between the time of manual and automated data collection, since we updated the automated measures (like the number of files, LOC, etc.) when Musubi was added.
- Our methodology assumes that artifacts in the repositories will show us the approach used for software development. However, not all activities will necessarily leave a trace in the repositories. For instance, only three packages (ESPResSo, Ludwig, Musubi) show evidence in their artifacts of unit testing and CI (Section 3.2), even though interviews suggest a more frequent use of both unit testing and CI. For example, OpenLB, pyLBM, and TCLB use such methods during development despite this not being obvious from an analysis of the online material.
- In our manual measures of the software repositories, we may have missed some data due to technology issues like broken links. This issue arose with the measurement of Palabos, which had a broken link to its user manual, as noted in Section 5.
- Our interviews were with a relatively small sample of four LBM developers. The pain points (Section 8) they expressed may not be representative of the rest of their community.

Although we found that unit testing and CI usage is likely higher than the repository artifacts suggest, we did not change the manual measurement results. Given that we could not interview a representative for every project, we left the data as collected to keep our measures consistent. Moreover, CI and unit testing done in private should not be counted as equivalent to the public versions we detected. When CI and unit testing are private, they are not as effective since they are not available for others to inspect and reproduce.

11.4. External Validity

External validity concerns whether the results of the study can be generalized (applied) to other situations/cases (Runeson and Höst, 2009). The scope of our work is LBM solvers. We are confident that our search was exhaustive and that we did not miss any highly popular examples. Therefore, the bulk of our validity concerns about LBM software observations are internal (Section 11.3), not external. However, our hope is that the trends observed, and the lessons learned for LBM software can be applied to other research software. With that in mind we identified the following threat to external validity:

- Generalization of our results will not be possible if the development of LBM software is fundamentally different from other research software.

12. Conclusion

This paper will help scientists select a suitable LBM package for their needs. However, its more fundamental purpose is to deepen our understanding of the state of the practice for developing LBM software. Understanding what artifacts developers produce, what tools they use, what processes they employ, and what pain points they experience is the starting point for devising future methods and tools for reducing development time and improving software quality.

To assess the state of the practice, we follow five steps: i) Identify a list of software packages; ii) Measure the packages using a measurement template (the template consists of 108 questions to assess 9 qualities (including the qualities of installability, usability and visibility)); iii) Rank the software using the Analytic Hierarchy Process (AHP); iv) Interview developers (each interview consists of 20 questions and takes about an hour); and, v) Conduct a domain analysis. We analyze the collected data by: i) Comparing our 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.

Current LBM software mostly follows best practices, as illustrated by Table 10, which summarizes the top five performers for each quality and for the overall quality. Taking the union of all entries in the table, we have 16 top performers: ESPREsSo, MechSys, OpenLB, DL_MESO, Sailfish, Ludwig, Musubi, pyLBM, waLBerla, laboetie, lettuce, ESPResSo++, LUMA, MP-LABS, Palabos, and HemeLB. Therefore, two thirds (67%) of the 24 packages appear in the top five for at least one quality. Almost all entries are alive, except for three packages: laboetie, MP-LABS, and HemeLB. Alive packages are in a healthy state with respect to adopting best practices.

Quality	Ranked 1st	Ranked 2nd	Ranked 3rd	Ranked 4th	Ranked 5th
Installability	ESPResSo	MechSys	OpenLB	DL_MESO (LBE)	Sailfish
Surface Correctness and Verifiability	Ludwig	Musubi	pyLBM	OpenLB	waLBerla
Surface Reliability	DL_MESO (LBE)	laboetie*	lettuce	MechSys	Musubi
Surface Robustness	DL_MESO (LBE)	ESPResSo++	lettuce	LUMA	OpenLB
Surface Usability	laboetie*	ESPResSo	OpenLB	Ludwig	Sailfish
Maintainability	Ludwig	MP-LABS*	ESPResSo++	ESPResSo	Palabos
Reusability	ESPResSo	waLBerla	ESPResSo++	Sailfish	HemeLB*
Surface Understandability	LUMA	Palabos	pyLBM	DL_MESO (LBE)	MechSys
Visibility and Transparency	Ludwig	Palabos	LUMA	ESPResSo	MP-LABS*
Overall Quality	ESPResSo	Ludwig	Palabos	OpenLB	LUMA

Table 10: Top performers for each quality (all packages were alive at the time of measurement, except for those marked by an asterisk (*)) (Sorted by Order of Quality Measurement)

LBM software is in a healthy state, as shown by comparing LBM artifacts, tools and processes, to current software guidelines and to other research software. The majority of LBM generated artifacts correspond to general recommendations from research software developers (Section 5). LBM software even seems to be ahead of best practices with an example of a video user guide for one of the packages. LBM is mostly keeping pace with other research software with respect to tool usage (Section 6), with 67% of the surveyed tools using version control. The state of the practice for the development of LBM software matches the quasi-agile process generally adopted for research software (Section 7). Some LBM developers employ the best practice process of peer review.

Although the state of the practice for LBM software is healthy, we note areas where the state does not match existing research software development guidelines, as follows:

- Add a roadmap document showing what is planned for the project's future to provide continuity of purpose, facilitate stakeholder collaboration and assist with prioritization. (Section 5)

- Add a code of conduct to improve inclusivity and attract a wider pool of collaborators. (Section 5)
- Increase the use and enforcement of programming style guides to improve understandability and productivity. (Section 5)
- Use checklists to ensure developers follow best practices. (Section 5)
- Include uninstall instructions as a courtesy to users that wish to save space and avoid polluting their build system. (Section 5)
- Increase the amount of API documentation to improve API understandability. (Section 5)
- Increase the use of CI to remove headaches associated with a separate integration phase, quickly detect and remove bugs and improve productivity, since developers will always be working from a stable base. (Section 6)
- Explicitly document the software's motivation and design principles to assist with on-boarding new developers. (Section 7)

For insight into devising future methods and tools, we interviewed four developers to learn about their pain points (Section 8). The identified pain points include:

P1 Lack of development time,

P2 Lack of software development experience,

P3 Lack of incentive and funding,

P4 Lack of external support,

P5 Technology hurdles,

P6 Ensuring correctness,

P7 Usability,

P8 Technical debt, and

P9 Documentation quality.

We found that developers are currently creating practices that address these pain points (Section 9). To address technical debt (P8) some developers have committed to designing for change (Section 9.1). To improve correctness (P6) developers are finding ways to circumvent the oracle testing problem, such as using peer review and pseudo-oracles (Section 9.2). Developers have prioritized documentation and usability (Section 9.3) to address the pain points of documentation and usability challenges (P7 and P9). For the pain point related to external support (P4), we found that this is currently addressed by groups like Better Scientific Software (BSSw), Software Sustainability Institute (Crouch et al., 2013), and Software Carpentry (Wilson and Lumsdaine, 2006; Wilson, 2016). Other pain points do not seem to be addressed by current practices. This motivates our eight recommendations for potential future directions (Section 10), in order of increasing development effort:

1. Employ linters to address P1, P2, P5 and P8. (Section 10.1)
2. Conduct rigorous peer reviews to address P1, P2, P5, P6, and P8. (Section 10.2)
3. Write and submit papers on software, to journals like JOSS, to address P3. (Section 10.3)
4. Apply advice from the open-source community on how to grow the number of contributors to address P1. (Section 10.4)

5. Further address the oracle problem for testing, with techniques such as the method of manufactured solutions and metamorphic testing, to address P6. (Section 10.5)

6. Augment theory manuals to include requirements information to address P1, P7 and P9. (Section 10.6)

7. Improve re-runnability, repeatability, reproducibility, and replicability to improve P6. (Section 10.7)

8. Use Domain Specific Modelling to generate all LBM code, build scripts, documentation, test cases from a stable knowledge base to address P1, P2, P5, P6, P7, P8 and P9. (Section 10.8)

Future work can be done to address the threats to validity (Section 11). For instance, our measures could be broadened to better capture shallow qualities. For example, usability experiments and performance benchmarks could be incorporated into the assessment. The accuracy of our estimate of participation level for CI and unit testing, which is not always visible in public artifacts, can be improved by adding interview questions. This additional data could be incorporated into the AHP ranking, ensuring quality designations that are more representative of the truth. In the future we could also expand the interview questions to go beyond reproducibility to also discuss replicability (Section 10.7). Furthermore, future interview guidelines should clarify the differences between usability and understandability (Section 2.1), since their answers suggest that not all participants understood the distinction. Analysis of future interview data would likely benefit from a more rigorous process, like a thematic domain analysis (Jung et al., 2022).

Our domain analysis (Section 2.8) was relatively shallow. In the future we would like to dig more deeply. We would like to produce tables of commonalities, variabilities, and parameters of variation for the family of LBM solvers. We could follow the template from Smith et al. (2008), or adopt the analysis techniques from Weiss (1998). Smith and Chen (2004) and Smith et al. (2017) provide examples that could be followed for a commonality analysis of a family of mesh generating software and a family of material models, respectively.

Acknowledgements

We would like to thank Ao Dong, Oluwaseun Owojaiye, Mohammad Ali Daeian, and Reza Sadeghi for fruitful discussions on topics relevant to this paper.

Conflict of Interest

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

References

Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. 2008. Scientific Software Development at a Research Facility. *IEEE Software* 25, 4 (July/August 2008), 44–51.

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

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

Shadab Anwar and Michael C. Sukop. 2009. Regional scale transient groundwater flow modeling using Lattice Boltzmann methods. *Computers & Mathematics with Applications* 58, 5 (2009), 1015–1023. https://doi.org/10.1016/j.camwa.2009.02.025

Yuanxun Bill Bao and Justin Meskas. 2011. Lattice Boltzmann method for fluid simulations. *Department of Mathematics, Courant Institute of Mathematical Sciences, New York University* (2011), 44.

Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnes, Harald Köstler, et al. 2021a. waLBerla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications* 81 (2021), 478–501.

Martin Bauer, Harald Köstler, and Ulrich Rüde. 2021b. lbmpy: Automatic code generation for efficient parallel lattice Boltzmann methods. *Journal of Computational Science* 49 (2021), 101269.

Mario Christopher Bedrunka, Dominik Wilde, Martin Kliemank, Dirk Reith, Holger Foysi, and Andreas Krämer. 2021. Lettuce: PyTorch-based Lattice Boltzmann Framework. In *International Conference on High Performance Computing*. Springer, 40–55.

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

Barry W Boehm. 2007. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*. Vol. 69. John Wiley & Sons.
Ben Boyter. 2021. Sloc Cloc and Code. https://github.com/boyter/scc. [Online; accessed 27-May-2021].
Alys Brett, James Cook, Peter Fox, Ian Hinder, John Nonweiler, Richard Reeve, and Robert Turner. 2021. Scottish Covid-19 Response Consortium. https://github.com/ScottishCovidResponse/modelling-software-checklist/blob/main/software-checklist.md.
Titus Brown. 2015. Notes from"How to grow a sustainable software development process (for scientific software)". http://ivory.idyll.org/blog/2015-growing-sustainable-software-development-process.html.
Jacques Carette and Oleg Kiselyov. 2011. Multi-stage programming with Functors and Monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76, 5 (2011), 349–375.
Jacques Carette, Spencer Smith, Jason Balaci, Anthony Hunt, Ting-Yu Wu, Samuel Crawford, Dong Chen, Dan Szymczak, Brooks MacLachlan, Dan Scime, and Maryyam Niazi. 2021. *Drasil*. https://github.com/JacquesCarette/Drasil/tree/v0.1-alpha
David Carty. 2020. Follow Google's lead with programming style guides. https://www.techtarget.com/searchsoftwarequality/feature/Follow-Googles-lead-with-programming-style-guides.
Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 2007. 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*. IEEE Computer Society, Washington, DC, USA, 550–559. https://doi.org/10.1109/ICSE.2007.77
Shiyi Chen and Gary D Doolen. 1998. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics* 30, 1 (1998), 329–364.
Neil P. Chue Hong, Alice Allen, Gonzalez-Beltran, Anita de Waard, Arfon M. Smith, Carly Robinson, Catherine Jones, Daina Bouquin, Daniel S. Katz, David Kennedy, Gerry Ryder, Jessica Hausman, Lorraine Hwang, Matthew B. Jones, Melissa Harrison, Mercè Crosas, Mingfang Wu, Peter Löwe, Robert Haines, Scott Edmunds, Shelley Stall, Sowmya Swaminathan, Stephan Druskat, Tom Crick, Tom Morrell, and Tom Pollard. 2019. Software Citation Checklist for Developers. https://doi.org/10.5281/zenodo.3482769
Tom Crick, Benjamin A. Hall, and Samin Ishtiaq. 2014. "Can I Implement Your Algorithm?": A Model for Reproducible Research Software. *CoRR* abs/1407.5981 (2014). http://arxiv.org/abs/1407.5981
S. Crouch, N. C. Hong, S. Hettrick, M. Jackson, A. Pawlik, S. Sufi, L. Carr, D. De Roure, C. Goble, and M. Parsons. 2013. The Software Sustainability Institute: Changing Research Software Attitudes and Practices. *Computing in Science Engineering* 15, 6 (Nov 2013), 74–80. https://doi.org/10.1109/MCSE.2013.133
Mario Rosado de Souza, Robert Haines, Markel Vigo, and Caroline Jay. 2019. What Makes Research Software Sustainable? An Interview Study With Research Software Engineers. *CoRR* abs/1903.06039 (2019). arXiv:1903.06039 http://arxiv.org/abs/1903.06039
Jean-Christophe Desplat, Ignacio Pagonabarraga, and Peter Bladon. 2001. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. *Computer Physics Communications* 134, 3 (2001), 273–290.
Ao Dong. 2021. *Assessing the State of the Practice for Medical Imaging Software*. Master's thesis. McMaster University, Hamilton, ON, Canada.
Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the Software for Understanding Climate Change. *Comuting in Science & Engineering* 11, 6 (November/December 2009), 65–74. https://doi.org/10.1109/MCSE.2009.193
C. Ebert and C. Jones. 2009. Embedded Software: Facts, Figures, and Future. *Computer* 42, 4 (April 2009), 42–52. https://doi.org/10.1109/MC.2009.118
Ahmed H. ElSheikh, W. Spencer Smith, and Samir E. Chidiac. 2004. Semi-formal design of reliable mesh generation systems. *Advances in Engineering Software* 35, 12 (2004), 827–841.
ESA. February 1991. *ESA Software Engineering Standards, PSS-05-0 Issue 2*. Technical Report. European Space Agency.
S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta. 2009. Scientific Computing's Productivity Gridlock: How Software Engineering Can Help. *Computing in Science Engineering* 11, 6 (Nov 2009), 30–39. https://doi.org/10.1109/MCSE.2009.205
Karl Fogel. 2005. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc.
Martin Fowler. 2006. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html.
SA Galindo-Torres. 2013. A coupled Discrete Element Lattice Boltzmann Method for the simulation of fluid–solid interaction with particles of general shapes. *Computer Methods in Applied Mechanics and Engineering* 265 (2013), 107–119.
E. Gamma, R. Helm, J. Vlissides, and I R Johnson. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
Davood Domairry Ganji and Sayyid Habibollah Hashemi Kachapi. 2015. *Application of nonlinear systems in nanomechanics and nanofluids: analytical methods and applications*. William Andrew.
Jeremy Garcia. 2016. How do you get programmers to join your project? https://opensource.com/business/16/9/how-to-get-programmers.
Lauraine Genota. 2018. Why Generation Z Learners Prefer YouTube Lessons Over Printed Books. Education Week, https://www.edweek.org/teaching-learning/why-generation-z-learners-prefer-youtube-lessons-over-printed-books/2018/09.
Marc-Oliver Gewaltig and Robert Cannon. 2012. Quality and sustainability of software tools in neuroscience. *Cornell University Library* (2012), 1–20.
Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2003. *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall, Upper Saddle River, NJ, USA.
Tomasz Gieniusz. 2019. GitStats. https://github.com/tomgi/git_stats. [Online; accessed 27-May-2021].
Ray Givler. 2020. A Checklist of Basic Software Engineering Practices for Data Analysts and Data Scientists. https://www.linkedin.com/pulse/checklist-basic-software-engineering-practices-data-analysts-givler/?articleId=6681691007303630849.
Carole Goble. 2014. Better Software, Better Research. *IEEE Internet Computing* 18, 5 (2014), 4–8. https://doi.org/10.1109/MIC.2014.88
Benjamin Graille and Loïc Gouarin. 2017. pylbm Documentation. (2017).
Alan Gray and Kevin Stratford. 2013. Ludwig: multiple GPUs for a complex fluid lattice Boltzmann application. *Designing Scientific Applications on GPUs. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, Taylor & Francis* (2013).
Jonathan D Halverson, Thomas Brandes, Olaf Lenz, Axel Arnold, Staš Bevc, Vitaliy Starchenko, Kurt Kremer, Torsten Stuehn, and Dirk Reith. 2013. ESPResSo++: A modern multiscale simulation package for soft matter systems. *Computer Physics Communications* 184, 4 (2013),

1129–1149.
Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How Do Scientists Develop and Use Scientific Software?. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE '09)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/SECSE.2009.5069155
Adrian RG Harwood, Joseph O'Connor, Jonathan Sanchez Muñoz, Marta Camps Santasmasas, and Alistair J Revell. 2018. LUMA: A many-core, fluid–structure interaction solver based on the lattice-Boltzmann method. *SoftwareX* 7 (2018), 88–94.
Manuel Hasert, Kannan Masilamani, Simon Zimny, Harald Klimach, Jiaxing Qi, Jörg Bernsdorf, and Sabine Roller. 2014. Complex fluid simulations with the parallel tree-based lattice Boltzmann solver Musubi. *Journal of Computational Science* 5, 5 (2014), 784–794.
Dustin Heaton and Jeffrey C. Carver. 2015. Claims About the Use of Software Engineering Practices in Science. *Inf. Softw. Technol.* 67, C (Nov. 2015), 207–219. https://doi.org/10.1016/j.infsof.2015.07.011
Michael A. Heroux and David E. Bernholdt. 2018. Better (Small) Scientific Software Teams, tutorial in Argonne Training Program on Extreme-Scale Computing (ATPESC). https://doi.org/articles/journal_contribution/ATPESC_Software_Productivity_03_Better_Small_Scientific_Software_Teams/6941438
Michael A. Heroux, James M. Bieman, and Robert T. Heaphy. 2008. Trilinos Developers Guide Part II: ASC Softwar Quality Engineering Practices Version 2.0. https://faculty.csbsju.edu/mheroux/fall2012_csci330/TrilinosDevGuide2.pdf.
Vincent Heuveline and Mathias J Krause. 2010. OpenLB: towards an efficient parallel open source library for lattice Boltzmann fluid flow simulations. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing. PARA*, Vol. 9. 570.
Vincent Heuveline, Mathias J Krause, and Jonas Latt. 2009. Towards a hybrid parallelization of lattice Boltzmann methods. *Computers & Mathematics with Applications* 58, 5 (2009), 1071–1080.
Timothy Hickey, Qun Ju, and Maarten H. Van Emden. 2001. Interval Arithmetic: From Principles to Implementation. *J. ACM* 48, 5 (Sept. 2001), 1038–1068. https://doi.org/10.1145/502102.502106
Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 426–437.
Daniel M. Hoffman and Paul A. Strooper. 1995. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA.
James Howison and Julia Bullard. 2016. Software in the Scientific Literature: Problems with Seeing, Finding, and Using Software Mentioned in the Biology Literature. *J. Assoc. Inf. Sci. Technol.* 67, 9 (Sept 2016), 2137–2155. https://doi.org/10.1002/asi.23538
Jez Humble and David G. Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, Upper Saddle River, NJ. http://my.safaribooksonline.com/9780321601919
IEEE. 1991. *IEEE Standard Glossary of Software Engineering Terminology*. Standard. IEEE.
IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998* (Oct. 1998), 1–40. https://doi.org/10.1109/IEEESTD.1998.88286
Joint Task Force on Software Engineering Ethics IEEE-CS/ACM and Professional Practices. 1999. Code of Ethics, IEEE Computer Society. https://www.computer.org/education/code-of-ethics.
Software Sustainability Institute. 2022. Online sustainability evaluation. https://www.software.ac.uk/resources/online-sustainability-evaluation.
Cezar Ionescu and Patrik Jansson. 2012. Dependently-Typed Programming in Scientific Computing — Examples from Economic Modelling. In *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages (Lecture Notes in Computer Science, Vol. 8241)*. Springer International Publishing, 140–156. https://doi.org/10.1007/978-3-642-41582-1_9
ISO/IEC. 2001. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC.
ISO/IEC. 2011. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Standard. International Organization for Standardization.
Shubhkesha Jalan. 2016. How to attract new contributors to your open source project. https://www.freecodecamp.org/news/how-to-attract-new-contributors-to-your-open-source-project-46f8b791d787/.
Michal Januszewski and Marcin Kostur. 2014. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. *Computer Physics Communications* 185, 9 (2014), 2350–2368.
Arne N. Johanson and Wilhelm Hasselbring. 2018. Software Engineering for Computational Science: Past, Present, Future. *Computing in Science & Engineering* Accepted (2018), 1–31.
Capers Jones. 2008. Measuring Defect Potentials and Defect Removal Efficiency. *Crosstalk, The Journal of Defense Software Engineering* 21, 6 (June 2008), 11–13.
Reiner Jung, Sven Gundlach, and Wilhelm Hasselbring. 2022. Thematic Domain Analysis for Ocean Modeling. *Environmental Modelling & Software* (Jan 2022), 105323. https://doi.org/10.1016/j.envsoft.2022.105323
Panagiotis Kalagiakos. 2003. The Non-Technical Factors of Reusability. In *Proceedings of the 29th Conference on EUROMICRO*. IEEE Computer Society, 124.
U. Kanewala and J. M. Bieman. 2013. Techniques for testing scientific programs without an oracle. In *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*. 48–57. https://doi.org/10.1109/SECSE.2013.6615099
Upulee Kanewala and Anders Lundgren. 2016. Automated Metamorphic Testing of Scientific Software. In *Software Engineering for Science*, Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal (Eds.). Taylor & Francis, Chapter Examples of the Application of Traditional Software Engineering Practices to Science, 151–174.
Matthias Katerbow and Georg Feulner. 2018. Recommendations on the development, use and provision of Research Software. https://doi.org/10.5281/zenodo.1172988
DS Katz, NP Chue Hong, T Clark, A Muench, S Stall, D Bouquin, M Cannon, S Edmunds, T Faez, P Feeney, M Fenner, M Friedman, G Grenier, M Harrison, J Heber, A Leary, C MacCallum, H Murray, E Pastrana, K Perry, D Schuster, M Stockhause, and J Yeston. 2021. Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved]. *F1000Research* 9, 1257 (2021). https://doi.org/10.12688/f1000research.26932.2

Diane Kelly. 2013. 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* (Ontario, Canada) *(CASCON '13)*. IBM Corp., Riverton, NJ, USA, 299–310. http://dl.acm.org/citation.cfm?id=2555523.2555555

Diane Kelly. 2015. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software* 109 (2015), 50–61. https://doi.org/10.1016/j.jss.2015.07.027

Diane Kelly and Terry Shepard. 2000. Task-directed software inspection technique: an experiment and case study. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research* (Mississauga, Ontario, Canada). IBM Press, 6. http://portal.acm.org/citation.cfm?id=782040#

Diane F. Kelly. 2007. A Software Chasm: Software Engineering and Scientific Computing. *IEEE Software* 24, 6 (2007), 120–119. https://doi.org/10.1109/MS.2007.155

Diane F. Kelly and Rebecca Sanders. 2008. Assessing the Quality of Scientific Software. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE 2008)*. In conjunction with the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany. http://www.cse.msstate.edu/~SECSE08/schedule.htm

Diane F. Kelly, W. Spencer Smith, and Nicholas Meng. 2011. Software Engineering for Scientists. *Computing in Science & Engineering* 13, 5 (Oct. 2011), 7–11.

Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 29, 6 (2012), 18–21.

Marc J. Kuchner. 2012. *Marketing for Scientists: How to Shine in Tough Times*. Island Press, Washington, D.C.

Jonas Latt, Orestis Malaspinas, Dimitrios Kontaxakis, Andrea Parmigiani, Daniel Lagrava, Federico Brogi, Mohamed Ben Belgacem, Yann Thorimbert, Sébastien Leclaire, Sha Li, et al. 2021. Palabos: parallel lattice Boltzmann solver. *Computers & Mathematics with Applications* 81 (2021), 334–350.

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

T.C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. *IEEE Software* 20, 6 (2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

Maximilien Levesque, Magali Duvail, Ignacio Pagonabarraga, Daan Frenkel, and Benjamin Rotenberg. 2013. Accounting for adsorption and desorption in lattice Boltzmann simulations. *Physical Review E* 88, 1 (2013), 013308.

A. Logg, K.-A. Mardal, and G. N. Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer. https://doi.org/10.1007/978-3-642-23099-8

D. V. Luciv, D. V. Koznov, G. A. Chernishev, A. N. Terekhov, K. Yu. Romanovsky, and D. A. Grigoriev. 2018. Detecting Near Duplicates in Software Documentation. *Programming and Computer Software* 44, 5 (01 Sep 2018), 335–343. https://doi.org/10.1134/S0361768818050079

Bazargul Matkerim, Darhan Akhmed-Zaki, and Manuel Barata. 2013. Development High Performance Scientific Computing Application Using Model-Driven Architecture. *Applied Mathematical Sciences* 7, 100 (2013), 4961–4974.

Marco D Mazzeo and Peter V Coveney. 2008. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications* 178, 12 (2008), 894–914.

Mike McQuaid. 2018. The Open Source Contributor Funnel (or: Why People Don't Contribute To Your Open Source Project). https://mikemcquaid.com/2018/08/14/the-open-source-contributor-funnel-why-people-dont-contribute-to-your-open-source-project/.

Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2018. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication* 48, 3 (2018), 295–330. https://doi.org/10.1177/0047281617721853 arXiv:https://doi.org/10.1177/0047281617721853

Peter Michalski. 2021. *State of The Practice for Lattice Boltzmann Method Software*. Master's thesis. McMaster University, Hamilton, Ontario, Canada.

Jürgen Münch, Stefan Trieflinger, and Dominic Lang. 2019. Product Roadmap – From Vision to Reality: A Systematic Literature Review. In *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. 1–8. https://doi.org/10.1109/ICE.2019.8792654

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

Udit Nangia and Daniel S. Katz. 2017. Track 1 Paper: Surveying the U.S. National Postdoctoral Association Regarding Software Use and Training in Research. Zenodo, 1–6. https://doi.org/10.5281/zenodo.814220 This paper was submitted to WSSSPE5.1 - http://wssspe.researchcomputing.org.uk/wssspe5-1/ The final accepted version is https://doi.org/10.6084/m9.figshare.5328442.

Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. A Survey of Scientific Software Development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy) *(ESEM '10)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1852786.1852802

Jakob Nielsen. 2012. Usability 101: Introduction to Usability. https://www.nngroup.com/articles/usability-101-introduction-to-usability/

Ileana Ober, Marc Palyart, Jean-Michel Bruel, and David Lugato. 2018. On the use of models for high-performance scientific computing applications: an experience report. *Software & Systems Modeling* 17, 1 (01 Feb 2018), 319–342. https://doi.org/10.1007/s10270-016-0518-0

Pablo Orviz, Álvaro López García, Doina Cristina Duma, Giacinto Donvito, Mario David, and Jorge Gomes. 2017. A set of common software quality assurance baseline criteria for research projects. https://doi.org/10.20350/digitalCSIC/12543

Oluwaseun Owojaiye, W. Spencer Smith, Jacques Carette, Peter Michalski, and Ao Dong. 2021. State of Sustainability for Research Software (poster). In *SIAM-CSE 2021 Conference on Computational Science and Engineering, Minisymposterium: Software Productivity and Sustainability for CSE*. https://doi.org/10.6084/m9.figshare.14039888.v2

D.L. Parnas, P.C. Clement, and D. M. Weiss. 1984. The modular structure of complex systems. In *International Conference on Software Engineer-*

ing. 408–419.
David L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM* 15, 2 (Dec. 1972), 1053–1058.
David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* 1 (1976), 1–9.
R. Phaal, C.J.P. Farrukh, and D.R. Probert. 2005. Developing a technology roadmapping system. In *A Unifying Discipline for Melting the Boundaries Technology Management:*. 99–111. https://doi.org/10.1109/PICMET.2005.1509680
Stephen R Piccolo and Michael B Frampton. 2016. Tools and techniques for computational reproducibility. *GigaScience* 5, 1 (07 2016). https://doi.org/10.1186/s13742-016-0135-4 arXiv:https://academic.oup.com/gigascience/article-pdf/5/1/s13742-016-0135-4/25513324/13742_2016_article_135.pdf s13742-016-0135-4.
Roman Pichler. 2012. Working with an Agile Product Roadmap. https://www.romanpichler.com/blog/agile-product-roadmap/.
Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. 2016. More Common Than You Think: An In-depth Study of Casual Contributors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 112–123. https://doi.org/10.1109/SANER.2016.68
Gustavo Pinto, Igor Wiese, and Luis Felipe Dias. 2018. How Do Scientists Develop and Use Scientific Software? An External Replication. In *Proceedings of 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 582–591. https://doi.org/10.1109/SANER.2018.8330263
Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2018. Categorizing the Content of GitHub README Files. arXiv:1802.06997 [cs.SE]
Professional Engineers Act. 2021. Professional Engineers Act, RSO 1990, c P. 28. https://canlii.ca/t/5568z.
Brian Proffitt. 2017. How to Attract New Contributors. https://www.redhat.com/en/blog/how-attract-new-contributors.
Markus Püschel, Bryan Singer, Manuela Veloso, and José M. F. Moura. 2001. Fast Automatic Generation of DSP Algorithms. In *International Conference on Computational Science (ICCS) (Lecture Notes In Computer Science, Vol. 2073)*. Springer, 97–106.
Patrick J. Roache. 1998. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico.
Suzanne Robertson and James Robertson. 1999. *Mastering the Requirements Process*. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, Chapter Volere Requirements Specification Template, 353–391.
J. Rokicki and L. Laniewski-Wollk. 2016. Adjoint lattice Boltzmann for topology optimization on multi-GPU architecture. *Computers & Mathematics with Applications* 71, 3 (2016), 833–848.
Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (19 Dec 2009), 131–164. https://doi.org/10.1007/s10664-008-9102-8
Mariusz Rutkowski, Wojciech Gryglas, Jacek Szumbarski, Christopher Leonardi, and Łukasz Łaniewski-Wołłk. 2020. Open-loop optimal control of a flapping wing using an adjoint Lattice Boltzmann method. *Computers & Mathematics with Applications* 79, 12 (2020), 3547–3569.
Reza Sadeghi, Nadav Gasner, Seyedvahid Khodaei, Julio Garcia, and Zahra Keshavarz-Motamed. 2022a. Impact of mixed valvular disease on coarctation hemodynamics using patient-specific lumped parameter and Lattice Boltzmann modeling. *International Journal of Mechanical Sciences* 217 (2022). https://doi.org/10.1016/j.ijmecsci.2021.107038
Reza Sadeghi, Seyedvahid Khodaei, Javier Ganame, and Zahra Keshavarz-Motamed. 2020. Towards non-invasive computational-mechanics and imaging-based diagnostic framework for personalized cardiology for coarctation. *Scientific Reports* 10, 1 (2020), 9048. https://doi.org/10.1038/s41598-020-65576-y
R. Sadeghi, B. Tomka, S. Khodaei, J. Garcia, J. Ganame, and Z. Keshavarz-Motamed. 2022b. Reducing Morbidity and Mortality in Patients With Coarctation Requires Systematic Differentiation of Impacts of Mixed Valvular Disease on Coarctation Hemodynamics. *Journal of the American Heart Association* 11, 2 (January 2022), 26.
Rebecca Sanders and Diane Kelly. 2008. Dealing with Risk in Scientific Software Development. *IEEE Software* 4 (July/August 2008), 21–28.
Tobias Schlauch, Michael Meinel, and Carina Haupt. 2018. DLR Software Engineering Guidelines. https://doi.org/10.5281/zenodo.1344612
Sebastian Schmieschek, Lev Shamardin, Stefan Frijters, Timm Krüger, Ulf D Schiller, Jens Harting, and Peter V Coveney. 2017. LB3D: A parallel implementation of the Lattice-Boltzmann method for simulation of interacting amphiphilic fluids. *Computer Physics Communications* 217 (2017), 149–161.
Michael A Seaton, Richard L Anderson, Sebastian Metz, and William Smith. 2013. DL_MESO: highly scalable mesoscale simulations. *Molecular Simulation* 39, 10 (2013), 796–821.
Judith Segal. 2005. When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering* 10, 4 (Oct. 2005), 517–536. https://doi.org/10.1007/s10664-005-3865-y
Judith Segal and Chris Morris. 2008. Developing Scientific Software. *IEEE Software* 25, 4 (July/August 2008), 18–20.
Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629
Vandana Singh, Brice Bongiovanni, and William Brandon. 2021. Codes of conduct in Open Source Software—for warm and fuzzy feelings or equality in community? *Software Quality Journal* (2021). https://doi.org/10.1007/s11219-020-09543-w
Andrew Slaughter, Cody Permann, Jason Miller, Brian Alger, and Stephen Novascone. 2021. Continuous Integration, In-Code Documentation, and Automation for Nuclear Quality Assurance Conformance. *Nuclear Technology* 207 (01 2021), 1–8. https://doi.org/10.1080/00295450.2020.1826804
Arfon Smith. 2022. Enhanced support for citations on GitHub. https://github.blog/2021-08-19-enhanced-support-citations-github/.

Arfon M Smith, Daniel S Katz, Kyle E Niemeyer, and FORCE11 Software Citation Working Group. 2016a. Software Citation Principles. *PeerJ Preprints* 4 (Aug. 2016), e2169v4. https://doi.org/10.7287/peerj.preprints.2169v4
Arfon M. Smith, Kyle E. Niemeyer, Daniel S. Katz, Lorena A. Barba, George Githinji, Melissa Gymrek, Kathryn D. Huff, Christopher R. Madan, Abigail Cabunoc Mayes, Kevin M. Moerman, Pjotr Prins, Karthik Ram, Ariel Rokem, Tracy K. Teal, Roman Valls Guimera, and Jacob T. Vanderplas. 2018c. Journal of Open Source Software (JOSS): design and first-year review. *PeerJ Computer Science* 4 (12 Feb. 2018), e147+. https://doi.org/10.7717/peerj-cs.147

Barry Smith, Roscoe Bartlett, and xSDK Developers. 2018a. xSDK Community Package Policies. https://doi.org/10.6084/m9.figshare.4495136.v6

Spencer Smith, Yue Sun, and Jacques Carette. 2015. State of the practice for developing oceanographic software. *McMaster University, Department of Computing and Software* (2015).

W. Spencer Smith. 2016. A Rational Document Driven Design Process for Scientific Computing Software. In *Software Engineering for Science*, Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal (Eds.). Chapman and Hall/CRC, Boca Raton, FL, Chapter Examples of the Application of Traditional Software Engineering Practices to Science, 33–63.

W. Spencer Smith. 2018. Beyond Software Carpentry. In *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*. 1–8.

W. Spencer Smith and Jacques Carette. 2021. Sustainable Software via Generation. In *Proceedings of the 1st Annual Booth Resource and Innovation Cluster (BRIC) Symposium*. 21.

W Spencer Smith, Jacques Carette, and John McCutchan. 2008. Commonality analysis of families of physical models for use in scientific computing. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE08)*.

W. Spencer Smith and Chien-Hsien Chen. 2004. Commonality and Requirements Analysis for Mesh Generating Software. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, F. Maurer and G. Ruhe (Eds.). Banff, Alberta, 384–387.

W. Spencer Smith and Nirmitha Koothoor. 2016. A Document-Driven Method for Certifying Scientific Computing Software for Use in Nuclear Safety Analysis. *Nuclear Engineering and Technology* 48, 2 (April 2016), 404–418. https://doi.org/10.1016/j.net.2015.11.008

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

W. Spencer Smith, Lei Lai, and Ridha Khedri. 2007. Requirements Analysis for Engineering Computation: A Systematic Approach for Improving Software Reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation* 13, 1 (Feb. 2007), 83–107. https://doi.org/10.1007/s11155-006-9020-7

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. 2016b. State of Practice for Mesh Generation Software. *Advances in Engineering Software* 100 (Oct. 2016), 53–71.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. 2018b. State of the Practice for GIS Software. https://arxiv.org/abs/1802.03422.

W. Spencer Smith, John McCutchan, and Jacques Carette. 2017. *Commonality Analysis for a Family of Material Models*. Technical Report CAS-17-01-SS. McMaster University, Department of Computing and Software.

W. Spencer Smith, Yue Sun, and Jacques Carette. 2018d. Statistical Software for Psychology: Comparing Development Practices Between CRAN and Other Communities. https://arxiv.org/abs/1802.07362. 33 pp.

W. Spencer Smith and Wen Yu. 2009. A Document Driven Methodology for Improving the Quality of a Parallel Mesh Generation Toolbox. *Advances in Engineering Software* 40, 11 (Nov. 2009), 1155–1167. https://doi.org/10.1016/j.advengsoft.2009.05.003

W. Spencer Smith, Zheng Zeng, and Jacques Carette. 2018e. Seismology Software: State of the Practice. *Journal of Seismology* 22, 3 (May 2018), 755–788.

SourceLevel. 2022. What is a linter and why your team should use it? https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it.

Graeme Stewart et al. 2017. A Roadmap for HEP Software and Computing R&D for the 2020s. *arXiv* (2017). arXiv:1712.06982 [physics.comp-ph]

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

Keenan Szulik. 2017. Don't judge a project by its GitHub stars alone. https://blog.tidelift.com/dont-judge-a-project-by-its-github-stars-alone.

Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE)*. In conjunction with ICSE 2016, Austin, Texas, United States. 4 pp.

Carsten Thiel. 2020. EURISE Network Technical Reference. https://technical-reference.readthedocs.io/en/latest/.

Parastou Tourani, Bram Adams, and Alexander Serebrenik. 2017. Code of conduct in open source projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 24–33. https://doi.org/10.1109/SANER.2017.7884606

USGS. 2019. USGS (United States Geological Survey) Software Plannning Checklist. https://www.usgs.gov/media/files/usgs-software-planning-checklist.

Jarno Vähäniitty, Casper Lassenius, and Kristian Rautiainen. 2002. An Approach to Product Roadmapping in Small Software Product Businesses. University of Technologie, Software Business and Engineering Institute, Helsinki, Finland.

Maarten van Gompel, Jauco Noordzij, Reinier de Valk, and Andrea Scharnhorst. 2016. Guidelines for Software Quality, CLARIAH Task Force 54.100. https://github.com/CLARIAH/software-quality-guidelines/blob/master/softwareguidelines.pdf.

Hans van Vliet. 2000. *Software Engineering (2nd ed.): Principles and Practice*. John Wiley & Sons, Inc., New York, NY, USA.

Todd. L. Veldhuizen. 1998. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Lecture Notes in Computer Science*. Springer-Verlag.

Florian Weik, Rudolf Weeber, Kai Szuttor, Konrad Breitsprecher, Joost de Graaf, Michael Kuron, Jonas Landsgesell, Henri Menke, David Sean, and Christian Holm. 2019. ESPResSo 4.0–an extensible software package for simulating soft matter systems. *The European Physical Journal Special Topics* 227, 14 (2019), 1789–1816.

David M Weiss. 1997. Defining families: The commonality analysis. *submitted to IEEE Transactions on Software Engineering* (1997). http://www.research.avayalabs.com/user/weiss/Publications.html

David M Weiss. 1998. Commonality analysis: A systematic process for defining families. In *International Workshop on Architectural Reasoning for Embedded Systems*. Springer, 214–222. citeseer.ist.psu.edu/13585.html

R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* 27, 1–2 (2001), 3–35.

Wiegers. 2003. *Software Requirements, 2e*. Microsoft Press.

I. S. Wiese, I. Polato, and G. Pinto. 2019. Naming the Pain in Developing Scientific Software. *IEEE Software* (2019), 1–1. https://doi.org/10.1109/MS.2019.2899838

Wikipedia. 2022. Lint (software). https://en.wikipedia.org/wiki/Lint_(software).

Greg Wilson. 2016. Software Carpentry: lessons learned [version 2; referees: 3 approved]. *F1000Research* 3, 62 (2016), 1–12.

Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 2014. Best Practices for Scientific Computing. *PLoS Biol* 12, 1 (Jan. 2014), e1001745. https://doi.org/10.1371/journal.pbio.1001745

Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2016. Good Enough Practices in Scientific Computing. *CoRR* abs/1609.00037 (2016). http://arxiv.org/abs/1609.00037

Greg Wilson and Andrew Lumsdaine. 2006. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science Engineering* 8, 6 (Nov. 2006), 66–69. https://doi.org/10.1109/MCSE.2006.122

Gregory V. Wilson. 2006. Where's the Real Bottleneck in Scientific Computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist* 94, 1 (2006). http://www.americanscientist.org/issues/pub/wheres-the-real-bottleneck-in-scientific-computing

Moshe Zadka. 2018. How to open source your Python library. https://opensource.com/article/18/12/tips-open-sourcing-python-libraries.

Duo Zhang, Qiong Cai, and Sai Gu. 2018. Three-dimensional lattice-Boltzmann model for liquid water transport and oxygen diffusion in cathode of polymer electrolyte membrane fuel cell with electrochemical reaction. *Electrochimica Acta* 262 (2018), 282–296. https://doi.org/10.1016/j.electacta.2017.12.189