SAND2018-6922C

Talk to Me: A Case Study on Coordinating Expertise in Large-Scale Scientific Software Projects

Reed Milewicz and Elaine M. Raybourn
Sandia National Laboratories, 1611 Innovation Pkwy SE, Albuquerque, New Mexico 87123

***Abstract*—Large-scale collaborative scientific software projects require more knowledge than any one human can possess. This makes the coordination and communication of that expertise a key factor in creating and safeguarding software quality, without which we cannot have sustainable software. However, as researchers attempt to scale up the production of their software, they are confronted by problems of awareness and understanding. This presents an opportunity for us to develop better practices and tools that directly address these challenges. To that end, we conduct case study of developers of the Trilinos project, a multi-million line computational mathematics library developed at Sandia National Laboratories. We survey the software development challenges they face and show how they are connected with what they know and how they communicate. Based on this data, we provide a series of practicable recommendations, and outline a path forward for future research.**

I. Introduction

Large-scale scientific software projects are among the most knowledge-intensive undertakings in all of human history, consisting of extremely diverse communities of practice and inquiry. For example, a climate modeling application can consist of numerous codes for modeling the atmosphere and the ocean, each of which is written by a distinct research team. The effective realization of that application in an high-performance computing (HPC) environment relies heavily upon those with backgrounds in computational science and software engineering. The orchestration of that talent demands disciplined project managers and communication with stakeholders. Thousands of man-years of labor are poured into the software over the course of decades.

Given the long lifespan and criticality of these projects, **sustainability** has been a focal point for research in recent years. By sustainability, we mean the ability of the software to continue to function as intended in the future, which is necessary for the reliability and reproducibility of research[1]. Sustainability is a multi-faceted challenge that encompasses both social and technical aspects of software development. In this work, we focus on one such aspect: the creation, communication, and use of knowledge integral to the scientific software development process. Large scientific software projects require diverse forms of expertise, bringing together people of different backgrounds and perspectives; to have success, there must be close, effective interaction between those parties[2]. Unfortunately, as we attempt to scale up these projects, we are confronted by barriers – logistical, technical, and cultural – that make it hard for people to share and apply what they know. This increases both the cost and difficulty of software development and maintenance, and it ultimately threatens sustainability.

From a software engineering perspective, more work is needed to create better tools and methodologies to manage and maintain that software development knowledge. However, as Dennehy and Conboy 2016 observes, the culture and context of a software project are "critical determinants of software development success" and that "a method, practice, or tool cannot be studied in isolation"[3]. For these reasons, we offer a survey and study of knowledge management practices within the Trilinos project, a keystone scientific software library that is used by numerous applications within Sandia National Laboratories and elsewhere. We model how knowledge is created and shared and its relationship to common software development challenges in order to identify targets for intervention.

A. A Motivating Example

Robust public investment into next-generation supercomputers is vital to the scientific enterprise. At the same time, considering the enormous sums of money that must be spent to construct and maintain them, stakeholders involved must be held accountable to the taxpayers. For this reason, government agencies stipulate rigorous requirements that must be met both by the machine and the software that it runs; a supercomputer must provide sufficient capabilities and the software must be able to fully utilize them. In the acceptance testing phase, participating research organizations put forward representative codes to be run on a novel architecture, and their performance is compared against the capabilities advertised by the vendor.

In the past year, those safeguards were tested when an well-respected application powered by Trilinos struggled to scale beyond 2^{17} MPI processes during an acceptance phase, resulting in a nearly 30% drop in performance on the target architecture. Fortunately, all other applications passed the acceptance test and the contract was completed successfully, but the issue implied a potential "time bomb" for numerous

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

applications and had to be fixed[4]. A team of researchers was given several months to locate the bug to no avail, but the issue was finally resolved when one heroic Trilinos scientist-developer volunteered three weeks of his time to uncover it. The ultimate cause of the bug was two-fold. First and foremost, it was due to the misuse of an MPI function, *MPI_Reduce_scatter*, in a Trilinos meshing package used by the application due to a misunderstanding of its semantics. In addition to this, it was found that the vendor-supplied implementation of the function was inefficient, which contributed to the overall slowdown. This was fixed by splitting the call into separate calls to *MPI_Reduce* and *MPI_Scatter*.

A subsequent investigation into the incident revealed a deeper mystery: the *exact same bug* had been introduced, found, and fixed in Trilinos multiple times over the years. The offending code was first introduced in three packages between 1998-2000 and fixed in 2005, copied line-for-line into a fourth package in 2004 and fixed again in 2015, and finally introduced into the meshing package in 2014 and fixed in 2017. In each case, the discovery and solutions were socialized, comments were made in the code, and notes were left in an issue tracker, but that information did not flow to the right parties in each subsequent incident.

We stress that none of this reflects poorly upon the project; Trilinos is the work of a preeminent, world-class team of researchers and developers. Rather, this presents a chance for us to better understand the difficulties that large scientific software teams face. As Moe et al. 2014 describes, the hallmark of large-scale software is that no one can know everything[5]. Good, strategic communication and organization of expertise are necessary for a team to reach its full potential. For scientific software developers, that leads to several pertinent questions. How do researchers share (or fail to share) their knowledge? What practices or policies could be enacted to prevent or mitigate problems like the ones we have described? These we formulate as specific research questions:

- **RQ1**: Do scientific software developers face challenges in sharing their knowledge? If so, what are the challenges?
- **RQ2**: How does individual and organizational knowledge affect those problems?
- **RQ3**: How is that knowledge communicated?

II. Background

A. Terminology

There is no single, agreed-upon definition for what constitutes **knowledge** in a project. In this work, we follow the use the definition provided by Davenport and Prusak 1998, which states that knowledge is "a fluid mix of framed experience, values, contextual information, and expert insights that provides a framework for evaluating and incorporating new experiences and information. It originates in and is applied in the minds of knowers. In organizations, it often becomes embedded not only in documents or repositories but also in organizational routines, processes, practices, and norms"[6]. For example, Parise et al. 2006 shows how a senior research scientist at a company is valuable not only for their own expertise but also for their "critical relationships" with other knowledgeable people (e.g. in academia)[7]. In other words, a successful research project must exercise both individual knowledge and those individuals' connections to other sources of knowledge.

We adopt the refined model introduced by Kelly 2015, one built upon a decade of invaluable studies of scientific software development that better captures this dynamic[8]. It consists of five components or knowledge domains: **real world** (the phenomena being studied), **theory-based** (the models used to understand those phenomena), **software** (the development conventions and practices), **execution** (the tools and environment needed to create and run the software), and **operational** (the relationship between the use of the software solution and the real world problem). Each of these elements both inform the solution and many drive each other; for example, there is feedback between the theory and the real world, between writing the software and executing it on the hardware, and between the use of the software and its application to the real world problem.

B. Scientific Software Culture

Studies of complex R&D projects in industry have shown that a lack of a common language creates barriers to the communication and codification of knowledge surrounding project activities[9]. Because of the intimate relationship between software and science, the development process requires a diverse assortment of both domain experts and software engineers[10], and these teams are frequently distributed and multi-organizational[11]. As a multidisciplinary endeavour, each member can have highly specialized knowledge that isn't easily transferred from one person to another. This is also a frequent source of conflict, such as between scientists and software engineers[12][13] as well as between scientists from different disciplines[14][15].

Scientists typically see software as a tool for creating and expanding scientific understanding, and less emphasis is placed on activities which concern knowledge of the software itself, such as planning or documentation[12]. There is often an implicit assumption that a scientific model and its implementation in code are connected such that knowledge of one translates to the other. This leads to scientists use software that they don't truly understand and write software without creating artifacts needed to understand it[16]. Additionally, within the scientific community at large, there is a drive to produce publishable research, as publications are a pathway to funding, positions, and prestige[17]. Studies have shown that scientists, when placed under pressure to publish, tend to focus on activities that lead to publications while neglecting those which do not[18].

In many respects, this state of affairs is not unique to the scientific software domain: the success of distributed and multidisciplinary teams almost never happens by accident. In the words of Ratcheva 2009, "simply putting people together in groups, representing many disciplines, does not necessarily guarantee the development of a shared understanding"[19].

TABLE I: Characterization of Trilinos compared with findings of Post and Cook 2000[20]

Property	Typical DOE CSE Project
Code Complexity	20-50 independent packages
Code Size	500,000 LOC
Project Age	10-35 years
Release Schedule	1-2 major releases, 20-100 minor releases per year.
Size of Teams	3-25 professionals
	Trilinos
Code Complexity	57 packages
Code Size	2,247,210 SLOC
Project Age	19 years
Release Schedule	2 major releases per year
Size of Teams	3-6 professionals

III. Case Study

To investigate this topic, we surveyed developers of the Trilinos mathematical library project at Sandia National Labs. Trilinos is a confederation of several dozen semi-independent packages. While packages may differ from one another in purpose, size, maturity, testedness, clients, and development teams, they are generally interoperable with one another, and share datatypes, standardized interfaces, and a common vision for the project's ecosystem. It is rare for a client to use the entire codebase, rather they select a subset of the packages that pertain to their application; this leads to a combinatorial explosion in the number of ways in which Trilinos packages can be configured, built, and arranged. To give some perspective on the scale and complexity of the software, we characterize Trilinos in the context of other, similar DOE projects on Table I. The project is available as open-source software, hosted on Github, and follows a master-development branch model in which contributions are promoted to master if and only if all tests pass.

We recruited participants for our survey through the internal developers' mailing list and print advertisements; the survey was distributed as a PDF file and as printed copies, and respondents provided code names to separate their identities from their responses. The mailing list consisted of 60 individuals and, among these, 29 developers were considered to be active, primary contributors. We received 36 responses, with 26 coming from the "primary" group, 5 from more peripheral developers on the mailing list, and an additional 6 responses from (mostly junior) members who were not subscribed to the list. This gives us a confidence level of 95% with an interval of $\pm 6\%$ for primary developers alone and $\pm 11\%$ for the entire population.

Our questionnaire consisted of multiple sections: demographic information, career priorities, methods of communication, areas of expertise, and problems encountered in software development. Additionally, we requested the Github handle of our respondents so that we could cross-reference the survey results with metrics on software contributions. We begin by presenting the demographic information, which is summarized on Figure 1. The following are the highlights from this section:

- 86% of respondents had completed their PhD, and 83% were members of staff (the remainder being interns, contractors, and postdocs, etc.). Collectively, respondents had between 233 to 345 years of combined experience, with the median respondent having between 11 to 15 years of experience.
- 77% of respondents reported that they worked on 4 or more projects, which typically means that they spend half their time working on a primary project, with the other half divided between three or more smaller, focused initatives. In total, 72% of respondents reported working with 6 or more people on a regular basis, with the median respondent working with between 6 and 10 people.
- The research interests among Trilinos scientist-developers are very diverse, and, while there is some overlap, none of the respondents listed shared all the same interests. As a rule, projects like Trilinos do not hire for redundancy; each project member contributes unique skills to the project.

In other words, our respondents tend to be highly educated, uniquely qualified, and, at best, only in regular contact with a small fraction of the overall organization.

IV. Analysis

In this section, we present the findings of our survey. Our presentation centers on commonly encountered software development issues and their relationship to individual and organizational knowledge. In Section V, we provide a summary and discussion of the findings. Where correlations are provided, we use Pearson's r and set $\alpha = 0.05$ as the threshold for p-values to reject the null hypothesis. Survey materials and anonymized responses can be found online[1]. On Table II, we provide a brief overview of the key findings in this section.

A. Findings

For **RQ1**, respondents were presented with a list of commonly encountered issues in software development based on those in LaToza et al. 2006[21], a study of the work habits and mental models of software developers. For each, participants reported whether the issue was not a problem, a moderately difficult problem, or a difficult problem. The overall results can be seen in Table III. The median respondent reported having eleven of the nineteen problems, two of which were considered especially difficult. Our survey results suggest widespread agreement with the problems we listed, with majority support for 13 of the 19 challenges. We can now refine **RQ1** into several subquestions:

- **RQ1.1**: Are these problems widely correlated with each other or are they independent?
- **RQ1.2**: To what extent can they be explained by general measures of experience or position in the organization?

All of the problems we investigated are, in some sense, problems of coordination, understanding, and awareness; they

[1] https://github.com/rmmilewi/KnowledgeManagementSurvey

Fig. 1: A visual summary of the demographic information collected by our survey.

TABLE II: A summary of the findings of our analysis.

Research Questions	Findings
RQ1.1. Are the problems correlated with one another?	(F1) No, most problems appear to be independent of each other, suggesting multiple, latent causes.
RQ1.2. Do problems reflect experience, education, organizational position, etc.?	(F2, F3) Yes, six problems can be related to demographic factors. Moreover, there is a general relationship between the number of problems someone faces and their position in the organizational network. Well-connected people have fewer problems.
RQ2.1. Does expertise in specific areas help?	(F4, F5) Yes, operational and execution domain knowledge are helpful for four problems. Surprisingly, we find no evidence that real-world, theoretical, or software domain knowledge are helpful.
RQ2.2. Does knowing people who have expertise in specific areas help?	(F6) Yes, ten problems are related to having access to other people in specific areas of knowledge.
RQ3. Do communication strategies affect the occurrence of problems?	(F7,F8,F9) Yes, communication strategies appear to affect eight problems. Additionally, the frequency and variety of communication correlates positively with self-perception of knowledge.

concern what people know, who they know, and how they use that information. These challenges are certainly not unique to scientific software, but they take on added weight and meaning given the demanding and knowledge-intensive nature of the work. We must understand these obstacles if we intend on helping scientists create better software.

For **RQ.1.1**, we want to know whether the high number and frequency of problems is because they are all similar. The three most commonly reported problems were dividing attention between projects ($p_{dividedattention}$), understanding other's code ($p_{otherscode}$), and finding bugs related to code ($p_{bugrelatedcode}$); in terms of severe problems, the top three were $p_{dividedattention}$ (again), switching tasks because of requests ($p_{taskrequest}$), and convincing managers about code improvement tasks ($p_{convincingmanagers}$). We found that some categories of problems were highly intercorrelated, but many are composed of independent items. This we can infer using Cronbach's alpha (which we will refer to as C_α) as a measure of interrelatedness or reliability[22]. The score ranges from 0 to 1, and the rule of thumb is that $C_\alpha >= 0.70$ suggests a set of items has good internal consistency; strong internal consistency in survey measures suggests that they all measure some common, latent variable. In Table IV, we see that this holds true for only two of the five categories. From this, we can conclude that while many problems are common, they are often independent of each other and will need to be addressed separately (e.g. finding a reviewer for your code and finding someone to talk about a bug require different information).

Finding (F1): There are many common problems, but likely no common explanations. The weak relationship between survey items within each category suggests that there are many independent, latent causes for the problems.

Next, for **RQ.1.2** we want to apply Occam's Razor: can these problems be explained by simple demographic or organizational measures (e.g. do more experienced people have fewer problems), without our nuanced survey data?

First, we compare these responses against our demographic data, to see which problems can be most easily explained

TABLE III: Respondent ratings of proposed problems. In the survey, problems were presented without headings and in a different order.

Problem	This is a problem (% agree) / a difficult problem (% agree)
Code Understanding	
Understanding the rationale behind a piece of code ($p_{coderationale}$)	63.9% (13.9%)
Understanding code that someone else wrote ($p_{otherscode}$)	83.3% (22.2%)
Understanding the history of a piece of code ($p_{codehistory}$)	58.3% (8.3%)
Understanding code that I wrote a while ago ($p_{youroldcode}$)	22.2% (0.0%)
Task Switching	
Having to switch tasks often because of requests from my teammates or manager ($p_{taskrequest}$)	75.0% (38.9%)
Having to switch tasks because my current task gets blocked ($p_{taskblocked}$)	55.6% (11.1%)
Having to divide my attention between many different projects ($p_{dividedattention}$)	94.4% (58.3%)
Modularity	
Being aware of changes to code elsewhere that impact my code ($p_{changeothers}$)	58.3% (11.1%)
Understanding the impact of changes I make on code elsewhere ($p_{changeself}$)	61.1% (2.8%)
Links Between Artifacts	
Finding all the places code has been duplicated ($p_{duplication}$)	58.3% (2.8%)
Understanding who "owns" a piece of code ($p_{ownership}$)	38.9% (0.0%)
Finding the bugs related to a piece of code ($p_{bugsincode}$)	75.0% (8.3%)
Finding code related to a bug ($p_{bugrelatedcode}$)	83.3% (11.1%)
Finding out who is currently modifying a piece of code ($p_{modifiers}$)	33.3% (0.0%)
Team	
Convincing managers that I should spend time rearchitecting, refactoring, or rewriting code ($p_{convincingmanagers}$)	41.7% (25.0%)
Convincing developers to make changes to code I depend on ($p_{convincingdevelopers}$)	61.1% (16.7%)
Expertise Finding	
Finding the right person to talk to about a piece of code ($p_{rightpersoncode}$)	50.0% (8.3%)
Finding the right person to talk to about a bug ($p_{rightpersonbug}$)	38.8% (5.6%)
Finding the right person to review a change before a check-in ($p_{rightpersonreview}$)	25.0% (5.6%)

TABLE IV: Cronbach's alpha scores for problem categories.

Category	C_α
Code Understanding	0.770
Task Switching	0.715
Modularity	0.474
Artifacts	0.595
Team	0.594
Expertise Finding	0.579

by factors such as the level of education or the number of projects. We find that only three problems have any correlations significant at the $\alpha = 0.05$ level: work experience and $p_{bugrelatedcode}$ ($r = -0.350$,$p = 0.036$), and the number of projects and both $p_{codehistory}$ ($r = 0.361$,$p = 0.030$) and $p_{changeothers}$ ($r = -0.342$,$p = 0.041$). Meanwhile, none of the demographic variables can directly account for the number of problems that people face (difficult or otherwise). In other words, even though many of the problems we listed relate to understanding and awareness, raw quantities of experience and contact with others are only narrowly useful predictors.

Finding (F2): Experience and workload influence three of the nineteen problems (two positively, one negatively). However, there are no significant correlations between demographic measures and the number of problems that respondents face in general.

Second, we consider the relationship between problems and organizational network structure by looking at team composition. The Team API of Github allows projects to group developers into teams, and the Trilinos project uses this feature to match developers with particular packages or cross-cutting concerns (e.g. the framework team); we found 57 teams, one of which was a global team of all developers that we excluded from our analysis. From this data we produced the team member graph seen in Figure 2, which provides a rough estimation of the lines of communication between developers. For each node in the graph, we computed its triangle count, which is the number of triangles (cyclic paths of length 3) formed between it and its neighbors; triangle counting is a common measure in social network analysis and it is the underpinning for measures such as the clustering coefficient (see [23]). As we illustrate in Figure 2, there is a moderate, statistically significant negative correlation between triangles and problems: the more embedded a person is in the team network, the less likely they are to have problems. However, the triangle count only correlates with three specific problems at the $\alpha = 0.05$ level: $p_{youroldcode}$ ($r = -0.335, p = 0.045$), $p_{bugsincode}$ ($r = -0.465, p = 0.004$), and $p_{rightpersonbug}$ ($r = -0.353, p = 0.004$). From this, we can conclude for **RQ1.2** that we need to dig deeper into what and who people know, how they know them, and why.

Finding (F3): There is a moderate, significant relationship between embeddedness in the organizational network (as measured by triangle count) and the number of problems that people face, and three of the nineteen problems can be directly tied to this näive measure.

For **RQ2**, we want to characterize the range of expertise of each participant and their access to others with expertise. For this, we refine **RQ2** into two subquestions:

- **RQ2.1**: What do they know? Does that matter?
- **RQ2.2**: Who do they know? Does that matter?

We selected eight topics corresponding to the five knowledge areas described in Kelly 2015[8], and these are described

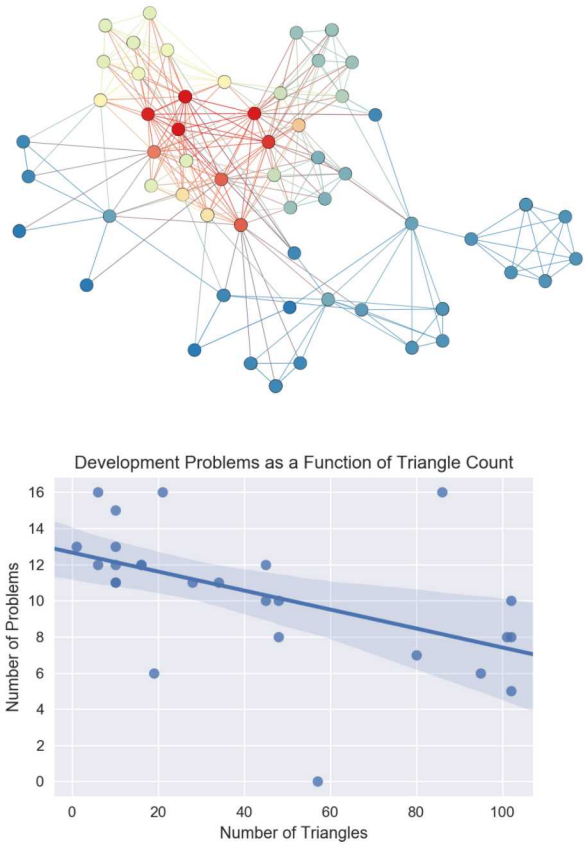

Fig. 2: Above, a graph of Trilinos developers on Github assigned to teams, where each edge indicates that two developers belong to the same development sub-team. The graph is color-coded to reflect the number of triangles formed between a node and its neighbors, red being more interconnected and blue being less interconnected. Below, we plot a linear regression on the number of problems reported by developers as a function of their triangle count ($r = -0.434$,$p = 0.008$). Even without considering our survey data, we see how problems of understanding and awareness are linked to the way in which developers are situated in the organization.

in Table V. For each topic, we asked participants to provide a self-assessment of their own familiarity with the topic on a 5-point Likert scale ranging from "not very knowledgeable" to "very knowledgeable". Additionally, we asked participants whether they worked with someone else that they "could turn to for help on that topic". Our survey results can be seen in Table V.

As a litmus test for our topic choices, we compare our survey findings against the five factor model by aggregating measures according to category, as can be seen on Figure 3. Our findings lend strong support to the model with agreement on eight out of ten possible edges. We found a strong relationship between operational and software knowledge ($r = 0.639$,$p = 0.00004$) which is not predicted by the five factor model; this is likely an artifact of the Trilinos team being

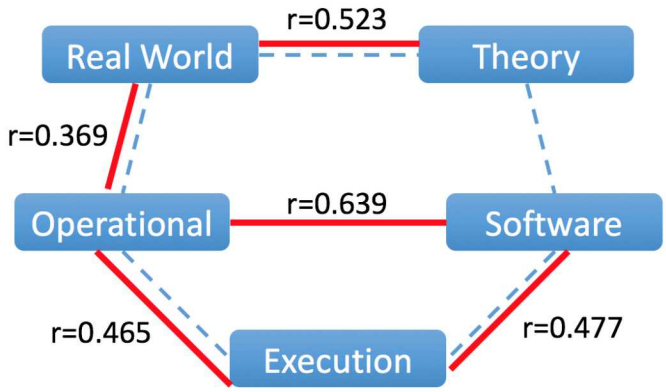

Fig. 3: A map of correlations between knowledge categories significant at the $\alpha = 0.05$ level. Dashed blue lines indicate relationships predicted by the model, and solid red lines indicate relationships suggested by our data.

library developers (i.e. writing code for other people's code). More mysterious is the missing edge between theory and software knowledge: many Trilinos developers translate theory into code, but there's no evidence that one domain is used to increase knowledge in the other. If we dig into the data, we find that scores increase with years of experience for every topic *except* for mathematics ($r = 0.001$,$p = 0.994$). This suggests that self-perception of knowledgeability in this area is relatively fixed, so our survey instrument is not picking up on any cross-pollination that may happen between theory domain and software domain topics.

For question **RQ2.1**, we want to know whether self-reported expertise has any measurable impact on the occurrence of problems (i.e. whether more knowledgeable people are more or less likely to report having problems).

We found knowledge had a significant, positive influence on four of the nineteen problems. First, finding the right person to talk about a piece of code ($p_{rightpersoncode}$) was less likely a problem among people with knowledge of version control ($r = -0.337$,$p = 0.044$) and hardware ($r = -0.348$,$p = 0.037$). Second, finding duplicated code ($p_{duplication}$) was seen as easier by people with high knowledge of compilers ($r = -0.362$,$p = 0.030$) and hardware ($r = -0.336$,$p = 0.045$). Third, knowledge of client codes and of version control were related to the problem of understanding code written by others ($p_{otherscode}$, $r = -0.344$,$p = 0.040$ and $r = -0.365$,$p = 0.040$). Finally, compiler expertise strongly predicts the problem of determining code ownership ($p_{ownership}$, $r = -0.503$,$p = 0.002$). Taken all together, we argue that the common driver in these findings is a deep awareness of the work context, such as the needs of clients or the execution of the code on target architectures.

TABLE V: Results for knowledge self-assessment questions

Topic	Knowledge Area	Histogram	Median (out of 5)	% know someone else
Knowledge of the real-world phenomena that the software is used to study.	Real-World		3	63.8%
The selection of mathematical techniques to attack a problem.	Theory		4	63.8%
Software design	Software		4	50.0%
Software construction	Software		5	50.0%
Compilers and compiler optimizations	Execution		4	55.0%
The effects of hardware architecture on algorithm performance	Execution		3	58.3%
Using a version control system	Execution		5	55.0%
How the software is integrated with client codes	Operational		4	47.2%

Finding (F4): Operational and execution domain knowledge have a positive effect on four of the nineteen problems. The common denominator is a deep awareness of how the software is assembled and used.

We also observed a few negative impacts: people with a high knowledge of design are more likely to report problems with getting requests to switch tasks ($r = 0.353$,$p = 0.035$) and having to divide their attention between projects ($r = 0.337$,$p = 0.043$), and people with a high knowledge of math were more likely to have problems with tracking who is modifying different pieces of code ($r = 0.358$,$p = 0.037$). Equally interesting are the missing connections. For example, despite the fact that all of our problems pertain to software development, there's no evidence that knowledge of software design or construction is helpful.

Finding (F5): Real-world, theory, and software domain knowledge provided **no** measurable benefit. These are software development problems but not problems solved by writing better software or doing better research.

Next, for question **RQ2.2** we want to know what benefits there are to being connected with other people who have knowledge. We carefully worded the prompt to check for contacts that a respondent "could turn to for help", with the expectation that people who have problems are more likely to seek out people who can help them. The results matched expectations: the average respondent reported having contacts that covered 4.4 of the 8 topics, and the amount of coverage was found to be a strong predictor of problems ($r = 0.455$,$p = 0.005$). Additionally, the only reliable indicator that someone had a contact for one topic was that they had a contact for another topic (average $r = 0.808$, average $p = 0.0000002$). This is to say that the ability or tendency to engage in this kind of networking was independent of an individual's background or position in the organization.

Ten of the nineteen problems are correlated with having people to reach out to (average $r = 0.389$, average $p = 0.0245$) spread across the following categories: code understanding ($p_{otherscode}$,$p_{codehistory}$, and $p_{youroldcode}$), task switching ($p_{taskrequest}$,$p_{taskblocked}$, and $p_{dividedattention}$), modularity ($p_{changeothers}$), links between artifacts ($p_{ownership}$ and $p_{bugsincode}$), and team ($p_{convincingdevelopers}$). This suggests that people are motivated to seek out knowledge-related contacts in order to maintain awareness of code and to negotiate and coordinate with others. Conspicuously absent from this list

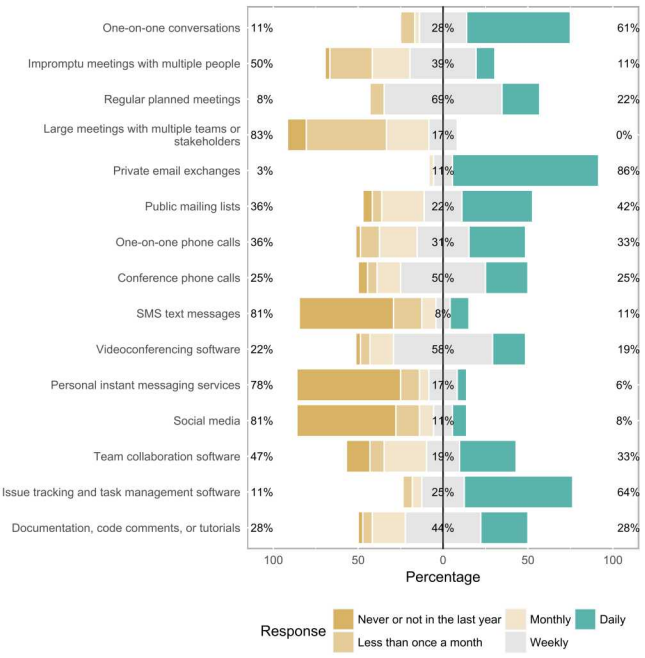

Fig. 4: The results of the communications portion of the survey, which examined how and how frequently developers communicated with each other.

are problems in the expertise finding category, all of which are highly independent of this measure (average $p = 0.53$), and we will return to this momentarily.

Finding (F6): Ten of the nineteen problems are influenced by seeking help from others. A theme that emerges is that these relationships are instrumental both for maintaining awareness of code as well for negotiating and coordinating with others.

Finally, for **RQ3**, we want to know how expertise is communicated among Trilinos developers, and what impact that has. We provided respondents with a list of different communication media, and for each we asked them to describe how frequently they used them on a 5-point scale from "never or not in the last year" to "daily". The results can be seen on Figure 4. We found that knowledge scores were strongly tied to communication scores in that those who communicated more frequently considered themselves more knowledgeable and vice versa (for average scores, $r = 0.491, p = 0.002$).

Meanwhile, we were able to link eight of the nineteen problems to differences in communication strategies.

Finding (F7): Knowledge scores mirror communication scores; those who communicate more, know more.

Face-to-face communication was a significant factor in all problems in the expertise finding category. Finding a reviewer for code ($p_{rightpersonreview}$) was easier for people who engaged in frequent one-on-one meetings ($r = -0.360$,$p = 0.031$); a similar relationship was found for one-on-one phone calls and finding someone to talk about a bug ($p_{rightpersonbug}$,$r = -0.403$,$p = 0.016$). Unstructured meetings with multiple people, meanwhile, was implicated in finding the right person to talk about a bug ($r = -0.360$,$p = 0.043$). Finally, large meetings with multiple stakeholders were correlated with both finding people to talk about code ($p_{rightpersoncode}$,$r = -0.383$,$p = 0.021$) as well as bugs ($r = -0.339$,$p = 0.031$). The takeaway is that these problems are a function of close and sustained communication, moreso than mere awareness of others (cf. **RQ2.2**).

Finding (F8): Face-to-face communications enable expertise-finding activities, affecting three of the nineteen problems. However, the evidence suggests that awareness of other's expertise alone is not enough: there must be ongoing, sustained contact.

Meanwhile, we found that digital communications were effective at solving some challenges while fueling others. Private email makes it easier to keep track of how other people's distant changes may affect your code ($p_{changeothers}$,$r = -0.372$,$p = 0.430$), but it is also a high-bandwidth channel of communication that is a frequent source of interruptions ($p_{dividedattention}$,$r = 0.437$,$p = 0.008$). Likewise, video conferencing has made it possible to keep people involved in many different projects, even when separated over great distances, but that certainly doesn't help with the divided attention problem ($r = 0.451$,$p = 0.006$). Meanwhile, team collaboration software (e.g. Jira, Confluence, etc.) has made it easier to track who is responsible for work items ($p_{ownership}$,$r = -0.329$,$p = 0.50$) and to piece together the origins of bugs ($p_{bugrelatedcode}$,$r = -0.383$,$p = 0.021$), but unaccounted slack which could be spent on rearchitecting and refactoring activities may be harder to come by ($p_{convincingmanagers}$,$r = 0.401$,$p = 0.015$). Lastly, on a purely positive note, we found that documentation was a valuable tool in bug finding ($p_{bugrelatedcode}$,$r = -0.396$,$p = 0.016$).

Finding (F9): Digital communication strategies are useful for protecting modularity and understanding the links between artifacts, but the communication overhead also introduces new challenges.

V. Discussion

A. Summary of Results

As we attempt to scale up the production of scientific software to meet the demands for innovation, we are beset by problems of understanding, coordination, and awareness. In general, they are not cured by time or experience (**F2**). We found no proof to suggest that additional domain or software development training will make them go away (**F5**). What evidence we do have tells us that they are complex and multi-faceted issues for which there can be no single solution (**F1**). In our case study, we found that the number of problems that respondents reported was, in some sense, a reflection of their "embeddedness" in the team (**F3**), which motivated further investigation into how expertise is situated and accessed.

The most useful forms of expertise were those that allowed respondents to position themselves between domains of activity, namely between the code and the machine (execution knowledge) and between the developers and the clients (operational knowledge)(**F4**). The benefit of that knowledge is indirect. For example, people who know more about compilers had fewer problems identifiying code ownership, but this is likely because compiler experts find it more important to keep track of where different code is coming from. Likewise, people with a high knowledge of design had more attention problems, and this is probably because those who do design work spend more time supporting and coordinating different people's activities.

Most of the problems are not able to be solved by individuals in isolation, and we found that this was a strong motivator for respondents to seek out contacts across different areas of expertise (**F6**). This was especially important for people having to negotiate with others to carry out work and for those trying to maintain awareness of code written by others. Interestingly, seeking out help did not reduce the frequency with which respondents reported problems, which implies that this is a risk mitigation rather than a risk reduction strategy. From this reading of the data, we can conclude that mere awareness and/or periodic contact is not enough to reduce the occurrence of issues: several of the problems we studied depended upon the quantity and quality of contact with others. Frequent communication was found to increase individual knowledge across the board (**F7**). With respect to particular problems, frequent face-to-face communications were important for locating and using other people's expertise (**F8**). Meanwhile, digital communications were found to help with change awareness and bug-finding, but also had the potential to exacerbate attention problems and create new bureaucratic barriers (**F9**).

B. Recommendations

The problems described in this paper are very common among large software development projects. However, not all solutions readily translate to the scientific software domain. Sletholt et al. 2012, a literature review on the use agile practices in scientific software development, found support for

TABLE VI: The number of problems in each category that have a statistically significant relationship with the factors studied in our survey.

Problem Area	Backgrounc	What They Know	Knowing Who Knows	How They Commu-nicate
Code Understanding	(2/4)	(1/4)	(3/4)	
Switching Tasks		(2/3)	(3/3)	(1/3)
Modularity	(1/2)		(1/2)	(1/2)
Links Between Artifacts	(2/5)	(3/5)	(2/5)	(2/5)
Team			(1/2)	(1/2)
Expertise Finding	(1/3)	(1/3)		(3/3)

some agile methods but not others [24]. The authors also caution that their evidence is strongest when considering "small projects with relatively few team members". For example, in a project like Trilinos, where the number one complaint among developers is having too many projects and not enough time, daily stand-up meetings may not be a realistic solution for everyone. This being said, we have identified several well-supported solutions that we believe may be a good fit for large scientific software teams like Trilinos.

Empowering knowledge brokers: Boden et al. 2009 calls attention to the role of knowledge brokers in distributed software development, that is, people capable of acting as bridges between different teams and domains of expertise; knowledge brokers are considered critical to enabling the flow of information between different sites[25]. Brokers tend to play an informal role in filling in structural holes in social networks, but works like Parise et al. 2006 argue that organizations should give formal recognition and power to these people[7]. In our case study, almost all of the Trilinos team is located within the same research building, but this is not a good guarantee of team cohesion: a recent study of R&D organizations found that the frequency of scientific collaboration drops off given 100 feet of distance between offices[26]. Along these lines, we note that 33% of our respondents indicated that they knew no one they could turn to for help in any of the knowledge areas while having an average of 5 different problems that could potentially be mitigated by having useful contacts; this is a situation where brokers could be helpful.

Cultivating organizational awareness: A benefit of having strong networks is the potential for serendipitous encounters. Santos et al. 2012 points out that the most effective knowledge sharing in complex R&D projects often happens in bars after work[9]. While strategies such as having knowledge brokers can help people locate specific expertise on demand, passive and casual exchanges of knowledge can clue people in to opportunities they might not have known about otherwise. This is echoed by Schossau and Wilson 2014, who found that one of the "completely unanticipated" benefits of Software Carpentry workshops was that they promoted awareness of technologies and methods, even if that information was not immediately useful[27]. It is possible to create these conditions through events such as interdepartmental lunches and seminars.

Encouraging integrative work: As our survey data shows, quality (not just quantity) of interactions matters for problems such as expertise finding. This echoes the findings of Hara et al. 2003, which distinguished between complementary and integrative collaborations in research groups, the former requiring awareness and the latter requiring frequent, close communication[14]. In our case, we found that 36% of respondents reported having no daily face-to-face interactions with other coworkers; the value of quiet isolation notwithstanding, there is also much to be said for close collaboration. However, the solutions in this category may be more demanding on individuals. Pair programming, for instance, has been shown to have great potential in conventional software development, but it has seen only limited adoption among scientific software teams[24]. Another strategy commonly employed in industry is to occasionally rotate members between different teams in order to disseminate best practices[28].

VI. Threats to Validity

There are several potential threats to internal validity. As with all surveys, our work is vulnerable to response biases. One concern in crafting this survey was social desirability bias, as our survey asks participants about their strengths and their weaknesses. This has come up in other surveys of scientist-developers such as Carver et al. 2013, which found that scientists tend to overestimate their own software development abilities[29]. We attempted to control for this by having a vetted protocol for collecting and storing survey data to protect the anonymity and confidentiality of responses; in general, we found that respondents were eager to volunteer information. Another concern was non-response bias because scientist-developers are notoriously preoccupied, but nevertheless we were able to get a sufficient number of responses. Moreover, as this was an organizational survey, we were able to precisely quantify the number of non-responses.

While our sample size is representative of the population, the population itself is a single team, and this raises questions about external validity. Most scientific software teams do not operate at the size and scale of Trilinos; Pinto et al. 2018 found that 95% of scientific software teams they surveyed had five members or fewer[30]. However, large-scale projects (e.g. libraries) are foundational for the scientific software ecosystem, and the problems we studied are universal to large software projects regardless of domain.

VII. Related Work

Szymczak et al 2016. argues for a rational, document-driven approach to codifying the knowledge surrounding scientific software development, and introduces Drasil, a platform for accomplishing this[31]. Along these lines, Smith et al. 2016 provides a series of case studies on the application of document-driven design to scientific software[32]. We recognize the value of this approach, especially when it comes to improving usability and verifiability, but we caution that most software development knowledge is tacit and unable to be codified; many of the specific problems we have described

in our work are not easily addressed by knowledge capture strategies.

On the subject of training and education, Gil et al. 2014 notes that there is "a very limited focus on issues of collaborative software development" in the education of early-career scientists[33]. Our work suggests that such training is of special importance to large-scale scientific software development.

VIII. Conclusion

In this work, we studied problems of communication and awareness in realizing large-scale scientific software by conducting a survey of developers of the Trilinos project, a key software library at Sandia National Laboratories. Our takeaway is that many widespread development problems can be related to the way in which expertise is situated and communicated within the team, and presented several preliminary recommendations. Our findings underscore the need for more investigation into development methodologies suitable for large-scale scientific software development. To that end, our future work will include ethnographic research into the work practices of scientist-developers.

References

[1] S. Hettrick, "Research software sustainability: Report on a knowledge exchange workshop," 2016.

[2] J. Cohen, C. Cantwell, N. C. Hong, D. Moxey, M. Illingworth, A. Turner, J. Darlington, and S. Sherwin, "Simplifying the development, use and sustainability of hpc software," *Journal of Open Research Software*, vol. 2, no. 1, 2014.

[3] D. Dennehy and K. Conboy, "Going with the flow: An activity theory analysis of flow techniques in software development," *Journal of Systems and Software*, 2016.

[4] K. Agelastos, M. Rajan, N. Wichmann, P. Lin, R. Baker, S. Domino, E. Draeger, S. Anderson, J. Balma, S. Behling, M. Berry, P. Carrier, M. Davis, K. McMahon, D. Sandness, K. Thomas, S. Warren, and T. Zhu, "Performance on trinity phase 2 (a cray xc40 utilizing intel xeon phi processors) with acceptance-applications and benchmarks (sand2017-6390pe, uur)." Sandia National Laboratories, 2017.

[5] N. B. Moe, D. Šmite, A. Šāblis, A.-L. Börjesson, and P. Andréasson, "Networking in a large-scale distributed agile project," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 12.

[6] T. H. Davenport and L. Prusak, *Working knowledge: How organizations manage what they know*. Harvard Business Press, 1998.

[7] S. Parise, R. Cross, and T. H. Davenport, "Strategies for preventing a knowledge-loss crisis," *MIT Sloan Management Review*, vol. 47, no. 4, p. 31, 2006.

[8] D. Kelly, "Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software," *Journal of Systems and Software*, vol. 109, pp. 50–61, 2015.

[9] V. R. Santos, A. L. Soares, and J. Á. Carvalho, "Knowledge sharing barriers in complex research and development projects: an exploratory study on the perceptions of project managers," *Knowledge and Process Management*, vol. 19, no. 1, pp. 27–38, 2012.

[10] E. S. Mesh and J. S. Hawker, "Scientific software process improvement decisions: A proposed research strategy," in *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*. IEEE, 2013, pp. 32–39.

[11] D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science: A systematic literature review," *Information and Software Technology*, vol. 67, pp. 207–219, 2015.

[12] J. Segal, "Scientists and software engineers: A tale of two cultures," 2008.

[13] D. F. Kelly, "A software chasm: Software engineering and scientific computing," *IEEE Software*, vol. 24, no. 6, pp. 120–119, 2007.

[14] N. Hara, P. Solomon, S.-L. Kim, and D. H. Sonnenwald, "An emerging view of scientific collaboration: Scientists' perspectives on collaboration and factors that impact collaboration," *Journal of the Association for Information Science and Technology*, vol. 54, no. 10, pp. 952–965, 2003.

[15] C. H. Jakobsen, T. Hels, and W. J. McLaughlin, "Barriers and facilitators to integration among scientists in transdisciplinary landscape analyses: a cross-country comparison," *Forest Policy and Economics*, vol. 6, no. 1, pp. 15–31, 2004.

[16] K. Hinsen, "Technical debt in computational science," *Computing in Science & Engineering*, vol. 17, no. 6, pp. 103–107, 2015.

[17] M. S. Anderson, E. A. Ronning, R. De Vries, and B. C. Martinson, "The perverse effects of competition on scientists work and relationships," *Science and engineering ethics*, vol. 13, no. 4, pp. 437–461, 2007.

[18] H. P. Van Dalen and K. Henkens, "Intended and unintended consequences of a publish-or-perish culture: A worldwide survey," *Journal of the Association for Information Science and Technology*, vol. 63, no. 7, pp. 1282–1293, 2012.

[19] V. Ratcheva, "Integrating diverse knowledge through boundary spanning processes–the case of multidisciplinary project teams," *International Journal of Project Management*, vol. 27, no. 3, pp. 206–215, 2009.

[20] D. Post and L. Cook, "A comparison of software engineering practices used by the llnl nuclear applications codes and by the software industry," in *Nuclear Explosive Code Developers Conference. Oakland, CA, Lawrence Livermore National Laboratory*, vol. 18, 2000.

[21] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[22] L. J. Cronbach, "Coefficient alpha and the internal structure of tests," *psychometrika*, vol. 16, no. 3, pp. 297–334, 1951.

[23] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 607–614.

[24] M. T. Sletholt, J. E. Hannay, D. Pfahl, and H. P. Langtangen, "What do we know about scientific software development's agile practices?" *Computing in Science & Engineering*, vol. 14, no. 2, pp. 24–37, 2012.

[25] A. Boden and G. Avram, "Bridging knowledge distribution-the role of knowledge brokers in distributed software development teams," in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE Computer Society, 2009, pp. 8–11.

[26] F. W. Kabo, N. Cotton-Nessler, Y. Hwang, M. C. Levenstein, and J. Owen-Smith, "Proximity effects on the dynamics and outcomes of scientific collaborations," *Research Policy*, vol. 43, no. 9, pp. 1469–1485, 2014.

[27] J. Schossau and G. Wilson, "Which sustainable software practices do scientists find most useful?" in *Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2)*, 2014.

[28] V. Santos, A. Goldman, and C. R. De Souza, "Fostering effective inter-team knowledge sharing in agile software development," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1006–1051, 2015.

[29] J. Carver, D. Heaton, L. Hochstein, and R. Bartlett, "Self-perceptions about software engineering: A survey of scientists and engineers," *Computing in Science & Engineering*, vol. 15, no. 1, pp. 7–11, 2013.

[30] G. Pinto, I. Wiese, and L. F. Dias, "How do scientists develop scientific software? an external replication," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 582–591.

[31] D. Szymczak, S. Smith, and J. Carette, "Position paper: A knowledge-based approach to scientific software development," in *Software Engineering for Science (SE4Science), IEEE/ACM International Workshop on*. IEEE, 2016, pp. 23–26.

[32] S. Smith, T. Jegatheesan, and D. Kelly, "Advantages, disadvantages and misunderstandings about document driven design for scientific software," in *Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2016 Fourth International Workshop on*. IEEE, 2016, pp. 41–48.

[33] Y. Gil, E. Moon, and J. Howison, "No science software is an island: Collaborative software development needs in geosciences," in *Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2)*, 2014.