Better Architecture, Better Software, Better Research

Stephan Druskat, *German Aerospace Center, 12489, Berlin, Germany*
Nasir U. Eisty, *University of Tennessee, Knoxville, TN, 37916, USA*
Robert Chisholm, *University of Sheffield, S1 4DP, Sheffield, U.K.*
Neil P. Chue Hong, *University of Edinburgh, EH8 9BT, Edinburgh, U.K.*
Ryan C. Cocking, *University of Sheffield, S3 7RH, Sheffield, U.K.*
Myra B. Cohen, *Iowa State University, Ames, IA, 50011, USA*
Michael Felderer, *German Aerospace Center and University of Cologne, 51147, Cologne, Germany*
Lars Grunske, *Humboldt-Universität zu Berlin, 10099, Berlin, Germany*
Sarah A. Harris, *University of Sheffield, S3 7RH, Sheffield, U.K.*
Wilhelm Hasselbring, *Kiel University, 24098, Kiel, Germany*
Thomas Krause, *Humboldt-Universität zu Berlin, 10099, Berlin, Germany*
Jan Linxweiler, *Technical University Braunschweig, 38106, Braunschweig, Germany*
Colin C. Venters, *European Organization for Nuclear Research, 1211, Geneva 23, Switzerland*

Better software drives better research, a fundamental principle in the research software engineering community. Similarly, better architecture underpins better software, a core belief in the software engineering research community. Therefore, we advocate and emphasize the importance of designing robust architectures for research software to elevate the quality of research outcomes, and illustrate this with two case studies.

Research software—the software used in the generation of research results—is used across all disciplines in different ways. From modeling and simulation, data management, analysis, and visualization to experimental management, software has become central to the production of research, the advancement of knowledge, and the decisions that are based on it.

Research software has been defined as including "source code files, algorithms, scripts, computational workflows and executables that were created during the research process or for a research purpose."[1] It can fulfill many different roles: instrument, model, tool, infrastructure, object of research, and research output in its own right.

The critical role that research software holds for reproducibility and confidence in results puts requirements on its development, maintenance, and availability. Research software must be of sufficient quality to ensure that it produces correct results. It must also be FAIR—i.e., findable, accessible, interoperable, and reusable[2]—to ensure that it can be reused, and supports use in interconnection with other software from the same, as well as other, domains.

Research software is often long-lived, becoming more complex as it is reused and extended. Due to the nature of research culture and funding, development teams change frequently. Each new generation of researchers (often Ph.D. students) and research software

Digital Object Identifier 10.1109/MCSE.2025.3573887
Date of publication 29 May 2025; date of current version 24 July 2025.

engineers (RSEs) working on the software brings their own culture, experience, and domain background with them. This may benefit the software but can also lead to loss of knowledge and accrual of technical debt[a]; lack of effort or documentation may result in "accidental" architectures that hinder, rather than support, the maintenance, extensibility, and evolution of the software. The negative effect of accidental architectures may be amplified by the research context, where value is placed on new features that enable novel research, not on maintainability and refactoring. Similarly, research funding rewards novelty and provides few resources for maintenance, refactoring, and improving architecture. Reusability is not a static condition of a "final" version but a condition of adaptability to changing needs that requires as few resources as possible, even when adapting to new hardware architectures, such as GPU programming models.

In this article, we advocate and emphasize the importance of designing robust architectures for research software. We present two case studies that analyze software maintainability through the use of static code analysis metrics. They highlight the importance of modularization for software quality and provide preliminary insights into the usefulness of metrics for assessing the maintainability of research software as an architectural trait.

SOFTWARE ARCHITECTURE

In software engineering, complexity is "anything related to the structure of a software system that makes it difficult to understand and modify." Software design is a key component in addressing complexity, which starts with software architecture, the process of making design decisions that lead to a set of structures needed to reason about a software system, outlining the system in terms of its elements, their properties, and relationships, which distinguishes itself from the details of a fine-grained system description.[3] For large, complex software systems, the design of the overall system structure presents a critical challenge. This level of design has been addressed in a number of ways, including informal diagrams and descriptive terms, module interconnection languages, and application domain-specific frameworks.

The software architecture establishes a link between the software systems requirements and the system design, providing a rationale for design decisions and encompassing decisions about architecturally significant requirements, which can then be evaluated early in the development process. For example, system performance is heavily impacted by the complexity of coordination and communication, especially when components are distributed across a network. In contrast, "organically grown" accidental software architectures lead to a range of symptoms, also known as "design smells," including software rigidity, fragility, immobility, opacity, and viscosity, which results in high-maintenance and evolution costs, the foundation for software collapse, and death in all software investment.[4] Additionally, accidental architectures make it harder to reason about the correctness of the code and its results.

The primary challenge in software engineering is the criteria for decomposing a large, complex system into modules. These modules should exhibit well-defined and stable interfaces and have high cohesion, i.e., the parts of a module are closely related and work well together. Additionally, dependencies between modules should be as low as possible, leading to a low coupling between the modules, which benefits the testability and reusability of components and services. Designing software to be stateless also has positive effects on testability and maintainability, and avoids side effects. In research software, this becomes relevant when code needs to be parallelized, e.g., to run on high-performance computing clusters.

When a new RSE begins work on an existing software system, the architecture should typically be the first element they review. This can be extremely challenging if the architecture is undocumented, which is often true. In such cases, reverse engineering the architecture from the code (architectural recovery and reconstruction) can aid in understanding the program; however, this is not without its own challenges and is an established area of research. As an embodiment of initial design decisions, establishing a sustainable software architecture is critical as it includes choices that are costly and difficult to alter later, making them crucial for careful consideration.[5]

RESEARCH SOFTWARE ENGINEERING

The production, operation, and maintenance of software that meets these standards requires professional research software practice, i.e. expert application of software engineering. The application of software engineering methods, tools, and practices in research is called *research software engineering*, and its practitioners *research software engineers*. It is important to

[a]The concept of technical debt refers to the cost of tradeoffs incurred when a development team consciously or unconsciously makes suboptimal technical decisions to achieve short-term goals at the expense of creating a technical solution that minimizes complexity and cost in the long-term.

note that not all research software, perhaps not even the majority, is developed by RSEs. Some domain researchers, or "researcher developers," also write code to support their research, albeit often without the application of software engineering methods.

RSEs employ research software engineering across many different scholarly disciplines. In addition to being software experts, they also sufficiently understand the research in the respective disciplines to enable its translation into software. RSEs are as diverse as the disciplines they work in, and around 54.5% hold a doctorate themselves.[6] They may work embedded in fixed-term research projects, in a particular research group, or as part of a central institutional team. They may also specialize in a particular subdiscipline of software engineering, e.g., modeling and simulation, optimization, or application development. Although RSEs are not necessarily formally trained software developers, and only around 23% studied computer science, they have had, on average, more than six years of software development experience.

SOFTWARE ARCHITECTURE IN RESEARCH SOFTWARE ENGINEERING

Research software is often developed with limited resources and without a long-term maintenance vision, which can lead to organic growth of accidental architecture and ultimately to software collapse.

Research software always has an architecture, but it may not always be formally defined or standardized. Research often involves exploring new ideas, methodologies, and data. Unlike in other domains, requirements may be unknown or change fundamentally and frequently. This can lead to rapid prototyping and frequent software design changes, making settling on a structured architecture challenging. Research typically prioritizes producing results and advancing knowledge over applying software engineering principles and practices.[7] As a result, researchers may prioritize functionality and performance over architectural concerns.

Researchers' diverse backgrounds, expertise in software development,[6] different coding styles and practices make enforcing standardized architectures across projects difficult. Available resources in projects for activities, such as architectural design and documentation, are limited, and further disincentivized by academic reward systems that value novelty over sustainability and reusability. Differing experience with and perspectives on software architecture between interdisciplinary collaborators or changes in code ownership over time can lead to accidental architectures. Additionally, research software varies widely in codebase size, lifetime, development team size, and purpose in research. Between applications in modeling, simulation and data analytics, prototyping in technology research, and research infrastructure,[8] it is challenging to establish unified architectural approaches for research software. All of the aforementioned factors influence the feasibility of applying architectures at a specific point in the software lifecycle, or applying architectures at all.

Agile development fits well with how most researchers work. Finding the right balance for architecture work is the art of agile architecture ownership. We can expect a coalescence of architecture work and agile software development practices, e.g., where robust architecture makes agile refactoring possible or easier. Architecture owners should make decisions at the *most responsible* moment, not the *last possible* moment.[9]

Despite its emphasis on flexibility and experimentation, research software can benefit significantly from a well-defined architecture that enhances the software's *maintainability* by making it easier to understand, extend, and modify over time. As researchers frequently revisit their software for future experiments or publications, a clear architectural design provides a solid foundation for these endeavors. Moreover, a modular architecture promotes *reusability*, allowing components of the software to be shared across projects or among researchers, thus fostering collaboration and accelerating progress in the field.

As projects grow in complexity or scope, a well-structured architecture facilitates the *scalability* of the software to handle larger datasets, more sophisticated algorithms, or increased computational demands. Additionally, a robust architecture contributes to the software's *reliability* and robustness, reducing the likelihood of errors or unexpected behavior during experiments or data analysis. This reliability is essential for ensuring the validity and integrity of research results.

Furthermore, a well-designed architecture serves as a foundation for documentation and *reproducibility*. Architectural design descriptions document the software's structure, functionality, and dependencies, enhancing the reproducibility of research results by enabling other researchers to understand and replicate the software environment and workflows. Clear architectural guidelines also facilitate collaboration among team members by providing a common framework for understanding and contributing to the software. This collaboration is essential for leveraging collective expertise and accelerating research progress while easing the onboarding process for new team members who need to understand the software's structure and design principles.

EVALUATING RESEARCH SOFTWARE ARCHITECTURE USING SOFTWARE METRICS

Software engineering research has developed metrics and tools to evaluate different aspects of software quality. Metrics, often derived using static code analysis, are generally helpful in gauging overall software quality. This is especially relevant for developers joining an existing or legacy software project. These insights can serve as starting point for an in-depth investigation of the software architecture that considers design documentation and original developers. Metrics can highlight potential architectural issues whose resolution should be prioritized to increase software maintainability.

The National Aeronautics and Space Administration Software Assurance Technology Center argues that effective evaluation of software maintainability is a combination of size (lines of code or LOC) and cyclomatic complexity.[b] While maintainability is complex and depends on less easily measurable factors, such as human cognition, knowledge retention, and team culture, metrics can provide valuable indicators of the quality and maintainability of a codebase. Software components with both a high cyclomatic complexity and a large size tend to have low maintainability. Components with low size and high complexity are also a reliability risk because they tend to be very terse code, which is difficult to change or modify. Other common metrics that static code analysis provides include code smells, bugs, vulnerabilities, and test coverage.

Some metrics are related to architecture, others are not. For an examination of software architecture, it is therefore important to understand what a metric expresses and how it is related to the quality attribute that is observed.

Some of the mentioned metrics are related to code-level attributes, not software architecture, such as code smells, adherence to language-dependent coding rules, or styles. Others, such as cyclomatic complexity, cognitive complexity,[c] and code duplication relate to the maintainability and adaptability of the software, and thus aim at the same quality attributes as architectural design. Yet other metrics, such as coupling and cohesion, are directly related to architectural traits, such as modularization, or they are a potential proxy for architectural attributes.

One example of proxy metrics is *change coupling*. Change coupling degrees quantify which parts of a codebase are frequently changed together, e.g. in the same commit. High change coupling degrees may uncover latent relations between different parts of the codebase, and help identify architectural patterns that break modularization and clean code principles. The metric may also show semantic or domain relatedness of different code parts, and work as proxy metric for lack of cohesion (see next paragraph). At the same time, it could also be an artifact of a specific development paradigm or workflow, and thus unrelated to architecture altogether. Change coupling is also potentially linked to *coupling*, which expresses the degree of dependence of one component on another. Coupling usually occurs as use of parts located in other modules, but also as use of global variables. High coupling may necessitate changes in one component whenever another changes, or lead to defects in one component whenever a defect occurs in another.

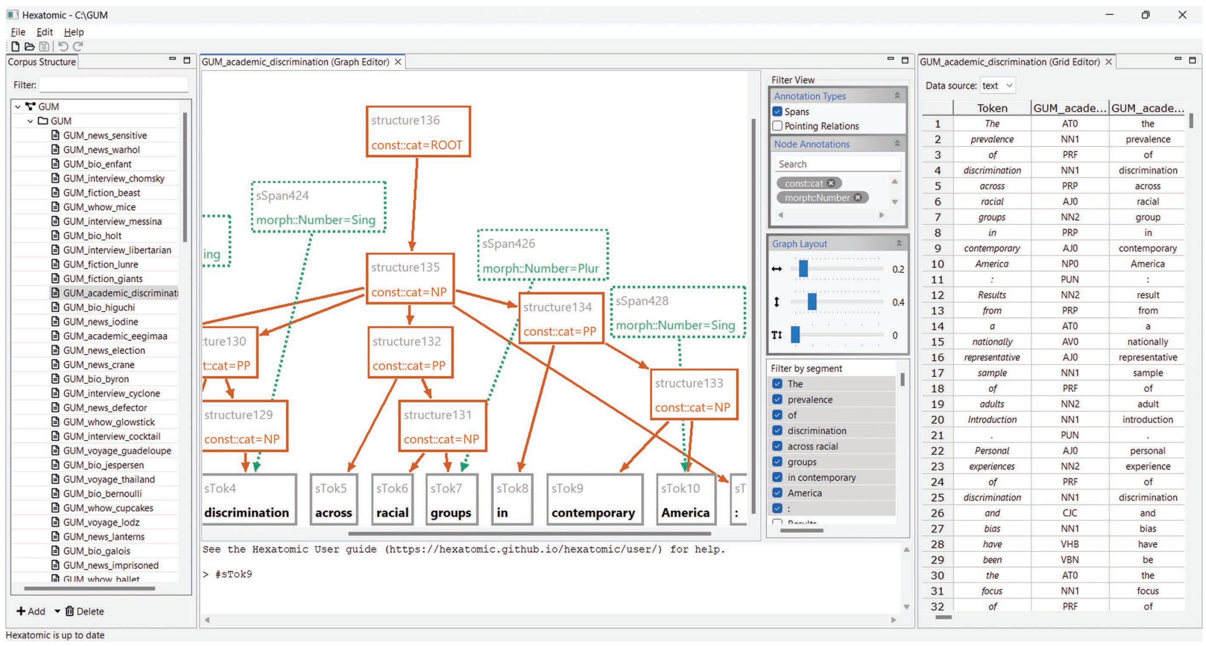
Cohesion expresses the degree to which elements within the same component "belong together." Cohesion can be high if there is a strong relationship between the functionality of a component and the data it processes, or a common purpose in the functionality of a component. Cohesion increases if related code or components are grouped. High cohesion, in contrast to high coupling, is preferable, as cohesion is taken to improve the maintainability and robustness of a software component. Good software architecture combines low coupling with high cohesion in well-modularized software.

These useful but complex measures for good software architecture are heavily context-dependent, not easy to generalize, and hard to implement as rules for static code analysis. In fact, metrics for coupling and cohesion, such as "lack of cohesion in methods" or "coupling between object classes," are not implemented in general-purpose static analysis tools, such as SonarQube or CodeScene.

We suggest that software metrics can be useful to evaluate some aspects of research software architecture. We argue they should be approached with caution and with awareness that they must be carefully interpreted with regard to what they measure and

[b]Cyclomatic complexity quantifies linearly independent paths through source code, i.e., the number of decision points in the code.

[c]Cognitive complexity is a measure of the effort it takes to understand code.



```
See the Hexatomic User guide (https://hexatomic.github.io/hexatomic/user/) for help.

> #sTok9
```

	Token	GUM_acade...	GUM_acade...
1	*The*	AT0	the
2	*prevalence*	NN1	prevalence
3	*of*	PRF	of
4	*discrimination*	NN1	discrimination
5	*across*	PRP	across
6	*racial*	AJ0	racial
7	*groups*	NN2	group
8	*in*	PRP	in
9	*contemporary*	AJ0	contemporary
10	*America*	NP0	America
11	*:*	PUN	:
12	*Results*	NN2	result
13	*from*	PRP	from
14	*a*	AT0	a
15	*nationally*	AV0	nationally
16	*representative*	AJ0	representative
17	*sample*	NN1	sample
18	*of*	PRF	of
19	*adults*	NN2	adult
20	*Introduction*	NN1	introduction
21	.	PUN	.
22	*Personal*	AJ0	personal
23	*experiences*	NN2	experience
24	*of*	PRF	of
25	*discrimination*	NN1	discrimination
26	*and*	CJC	and
27	*bias*	NN1	bias
28	*have*	VHB	have
29	*been*	VBN	be
30	*the*	AT0	the
31	*focus*	NN1	focus
32	*of*	PRF	of

FIGURE 1. Screenshot of the Hexatomic graphical user interface, showing simultaneous annotation of the same linguistic corpus data in an editor for annotations represented as graphs and annotations represented as spans over tokenized text.

what they contribute to an understanding of the architectural trait of interest.

Below, we provide two examples for evaluating the *maintainability* of research software using software metrics. We use three different sets of metrics provided by two different static analysis platforms. SonarQube Cloud (https://sonarcloud.io, SonarSource SA, Geneva, Switzerland) can continuously detect issues in source code. It supports 36 programming languages, including C, C++, COBOL, C#, Java, and Python. CodeScene (https://codescene.io, CodeScene AB, Malmö, Sweden) also provides insights into higher-level attributes of analyzed software projects, including architecture. It supports 32 programming languages, including Rust and the ones mentioned above, with the exception of COBOL. For this work, we have used the free tier of SonarQube Cloud for open source projects, and a trial version of the CodeScene Pro plan using CodeScene Enterprise (version 6.7.3).

Our analysis focuses on:

- SonarQube's code-level *issues*
- CodeScene's *refactoring targets*: prioritizes components with low code health and therefore high return on investment (ROI) for refactoring
- CodeScene's *architectural hotspots*: prioritizes architectural components based on understandability, maintainability and evolvability.

Hexatomic

Hexatomic[d] is an open source extensible graphical platform for manual and semiautomated multilayer annotation of linguistic corpus data[10] (see Figure 1). Its original development was funded in the call "Research Software Sustainability" by the German Research Foundation (DFG) to investigate minimal infrastructure requirements for the sustainable provision of research software.

Hexatomic was initially developed by two RSEs [0.5 full-time equivalent (FTE) each over three years] and two student assistants (0.25 FTE each over one year). One of the RSEs holds a Ph.D. degree in computer science (but with no research focus on software engineering), and the other is self-taught and holds an M.A. degree in literature and language subjects.

Hexatomic reimplements an architectural proof-of-concept for integrating existing linguistic infrastructure software built in Ruby, using a Java-based application framework, the Eclipse Rich Client Platform, with a strong focus on modularization and separation of concerns. Choosing it early in the development process facilitated architecturally aware design and implementation over the complete course of the project.

[d]Hexatomic source code: https://github.com/hexatomic/hexatomic.

During initial funding [1 October 2018 (t_0) – 31 December 2022 (t_2)], the software architecture was documented from 5 May 2020[e] to support developers, and static code analysis with SonarQube Cloud was introduced on 28 May 2020 (t_1)[f] to assert the quality of all patches merged into the codebase. SonarQube Cloud has been continuously used on pull requests for Hexatomic until t_2.

Figure 2 presents software metrics for versions of Hexatomic produced between t_1 and t_2. Figure 2(a) shows the number of "issues" detected by SonarQube Cloud for tagged versions of Hexatomic. Analysis of the first tagged version after the introduction of SonarQube Cloud (version v0.4.3) yielded 145 issues. In subsequent versions, the RSEs removed code smells from the codebase. Version v0.5.1 showed four issues and number of issues remained low until t_2 (v1.0.1).

We retrospectively analyzed Hexatomic versions between t_1 and t_2 with CodeScene to evaluate its maintainability as an attribute of software architecture, and to compare architectural hotspots with issues related to maintainability. Figure 2(b) and (c) show CodeScene's visualization of refactoring targets at t_1 and t_2, respectively. At t_1, Hexatomic includes four components ("bundles", 8811 LOC). CodeScene identified five Java files as "refactoring targets" and four "priority refactoring targets" with low code health and high ROI for refactoring. At t_2 (18,920 LOC, five bundles), CodeScene identified one priority refactoring target due to complex conditionals and nested conditional logic: `ProjectManager.java` (466 LOC) contains logic for managing linguistic corpus projects in Hexatomic. The file has "change coupling" degrees of 55% and 37% with the project management GUI (Figure 1, left), and the handler code for opening projects.

To find out if these instances of *change coupling* relate to factual *coupling between objects*, we analyzed `ProjectManager.java`[g] with PMD[h] (version 7.8.0)[i] , a multilingual static code analyzer. The respective PMD analysis rule (category/java/design.xml/CouplingBetweenObjects) found 27 coupling instances in the class (default threshold: 20), a potential architectural issue. The class contains logic for corpus creation *and* opening projects, and could be refactored into two classes. `ProjectManager.java` uses one constant of one of two highly change-coupled classes, `OpenSaltDocumentHandler.java`, three times. The constant refers to document identifiers for which persisted state is retrieved in the project manager class. Arguably, this constant is semantically closer to project management than handling logic, and could be declared in `ProjectManager.java` to reduce coupling. For `ProjectManager.java` the number of coupling issues related to change coupling is low, suggesting that change coupling does not function as proxy metric for coupling in this case.

Figure 2(d) and (e) show CodeScene's "architectural hotspots" visualizations of Hexatomic bundles at t_1 and t_2. Architectural hotspots compromise the understandability, maintainability, and evolvability of code. Between t_1 and t_2, the number of components that require extra maintenance effort, which may slow development (yellow in Figure 2) has been reduced from 4/5 to 2/6. This suggests that architecture health has improved over time.

However, the architecture of Hexatomic has not changed between t_1 and t_2, while one bundle handling corpus formats was added. Although there seems to be correlation between the existence of code-level issues in the codebase and overall architectural quality, we cannot assume causation. We argue that issues and architectural hotspots relate to the same target, *maintainability*, and thus emit correlated results. This points to a shortcoming in static code analysis platforms. These analyze *measurable* attributes of a codebase (here: maintainability), but not necessarily more complex architectural attributes, such as scalability and reproducibility.

In addition to refactoring, using architecturally strong frameworks can improve software architecture. Hexatomic is based on the Eclipse Rich Client Platform, which prescribes a highly modular architecture based on the OSGi specification[j]: *Bundles* encapsulate functionality and provide interfaces. Functionally related bundles are collected in *features*, which are combined in an *application*. Hexatomic uses this structure to separate the GUI from project management, format handling, data viewers, and annotation editors.

We argue that the architecture of Hexatomic at t_2 is improved as a combined effect of strong modularization and using service/event-based architectural patterns, and refactoring efforts. Without the high degree of modularization, refactoring would have been harder or impossible to achieve. With regard to coupling, the implementation overall follows good architectural practice, with some lack of separation of concerns.

[e]See https://archive.softwareheritage.org/swh:1:rev:355fce8624fda08e9d1611aae2f9280cd9b811be.

[f]See https://archive.softwareheritage.org/swh:1:cnt:1e9944a8ad5e0c8476c6d5aea9773974f1562059;path=/.travis.yml;lines=46-56.

[g]See https://archive.softwareheritage.org/swh:1:cnt:1fff97af003af2d720095d5ba27a82654bf5a239.

[h]PMD website: https://pmd.github.io/.

[i]PMD version 7.8.0: https://archive.softwareheritage.org/swh:1:rel:ac04c37bfb25f5b2ad24ebc506978aca8707f464.

[j]https://docs.osgi.org/specification.

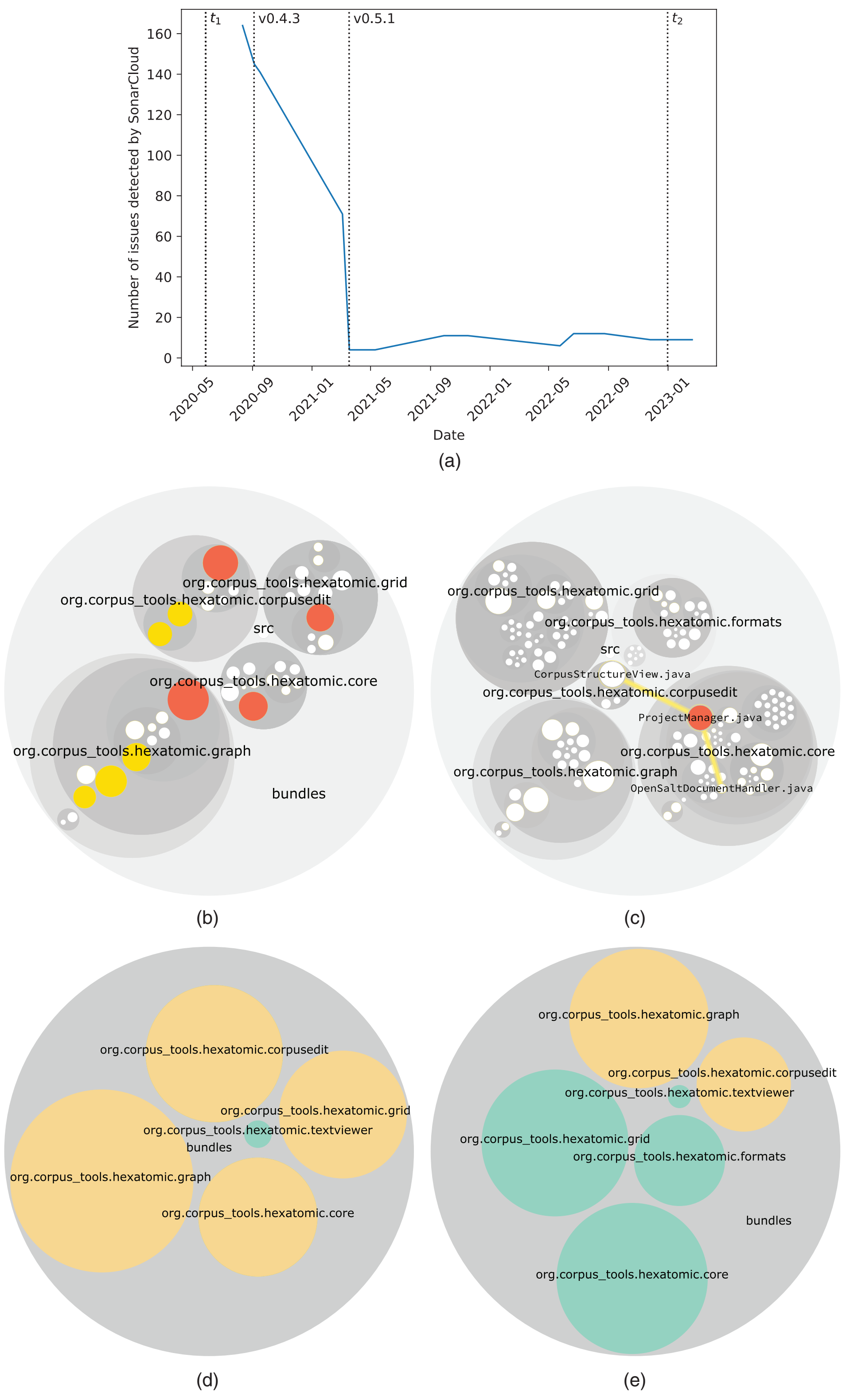


FIGURE 2. Hexatomic: (a) SonarQube issues, (b) and (c), differences in prioritized refactoring targets, and (d) and (e) differences in architectural hotspots in `bundles` between SonarQube Cloud introduction (t_1) and end of initial funding period (t_2). Colors: (b) and (c) White: not a target; yellow: refactoring target; red: priority refactoring target. (d) and (e) Yellow: problematic; green: healthy.

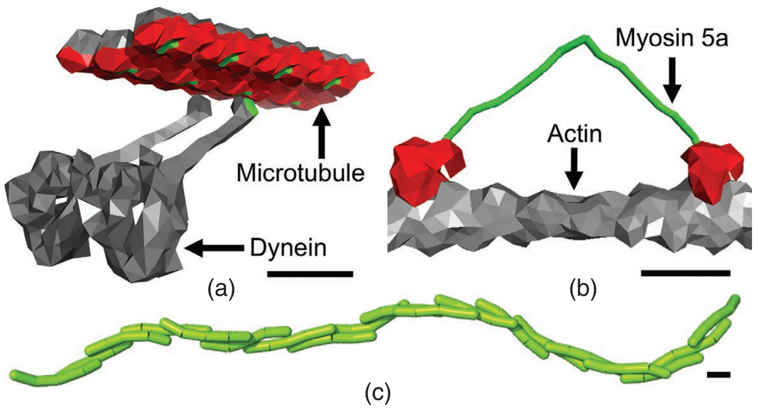


FIGURE 3. Representative snapshots of FFEA simulations. (a) Cytoplasmic dynein motor protein attached to its microtubule track. (b) Myosin 5a motor protein attached to its actin filament track. (c) Fibrin blood clotting protein, assembled into a protofibril. (a) Contains only blobs, (b) contains blobs and rods, and (c) contains only rods. Scale bars: 10 nm.

The development context of Hexatomic provided a clear focus on software quality, which has likely contributed to an overall maintainable, well-structured codebase as evidenced through static code analysis [Figure 2(e)].

Fluctuating Finite Element Analysis

Fluctuating finite element analysis[k] (FFEA) is a biomolecular modeling program designed to perform continuum mechanics simulations of globular and fibrous proteins.[11] A unique aspect of FFEA is that finite element meshes, which represent the deformable protein structure, are subject to thermal fluctuations during a simulation. This feature is essential to capturing protein behavior because they are nanoscale objects. Protein dynamics are saved to a trajectory file that can then be visualized; Figure 3 displays representative snapshots of four biomolecular systems.

The FFEA software was inspired by the "resolution revolution" in cryoelectron microscopy that has provided structural biologists with information about increasingly large biomolecular complexes, which are outside the regime accessible to conventional atomistic molecular simulation tools.[12] The software has been in development since 2010, by nine authors, leading to 11 publications. It consists of a C++ simulation platform and a Python package (FFEATools) that provides additional tools for managing simulations.

Typical of research software, the majority of FFEA's code has been developed by Ph.D. students and postdoctoral researchers, with expert domain knowledge but lacking formal training in software engineering. Each researcher focused on implementing new functionality required to achieve their specific research objectives over a period of two to three years, with limited access to guidance from previous developers who had since moved on. They typically followed their own development style without fully familiarizing themselves with, or adhering to, a standardized approach for the entire codebase, resulting in a highly inconsistent architecture.

In April 2024, the FFEA C++ codebase was 24,429 LOC, with scant developer documentation and only 62 test cases. These tests offered limited coverage, with some consistently failing and others considered "flaky," yielding inconsistent results across runs. Over more than a decade of mostly independent development, FFEA's scope expanded organically, leaving earlier components undermaintained.

FFEA is capable of modeling two types of components, "blobs" (tetrahedral meshes) and 1D "rods" (Figure 3), along with interactions between the two types. These two components are largely independent within the codebase, with limited shared architectural design. The "blob" architecture follows an object-orientated approach, with errors handled via the return of error codes. In contrast, the "rod" is implemented with many static functions and handles errors using exceptions. Both components heavily utilized manual memory management, typical of C++ written when the project was started.

In July 2024, an RSE (0.36 FTE over five months) with a Ph.D. degree in computer science, collaborated with the most recent developer (one FTE over five months), a self-taught programmer with a Ph.D. degree in biophysics, to modernize FFEA and improve its maintainability for future developers. This work involved

[k] FFEA: https://ffea.bitbucket.io/.

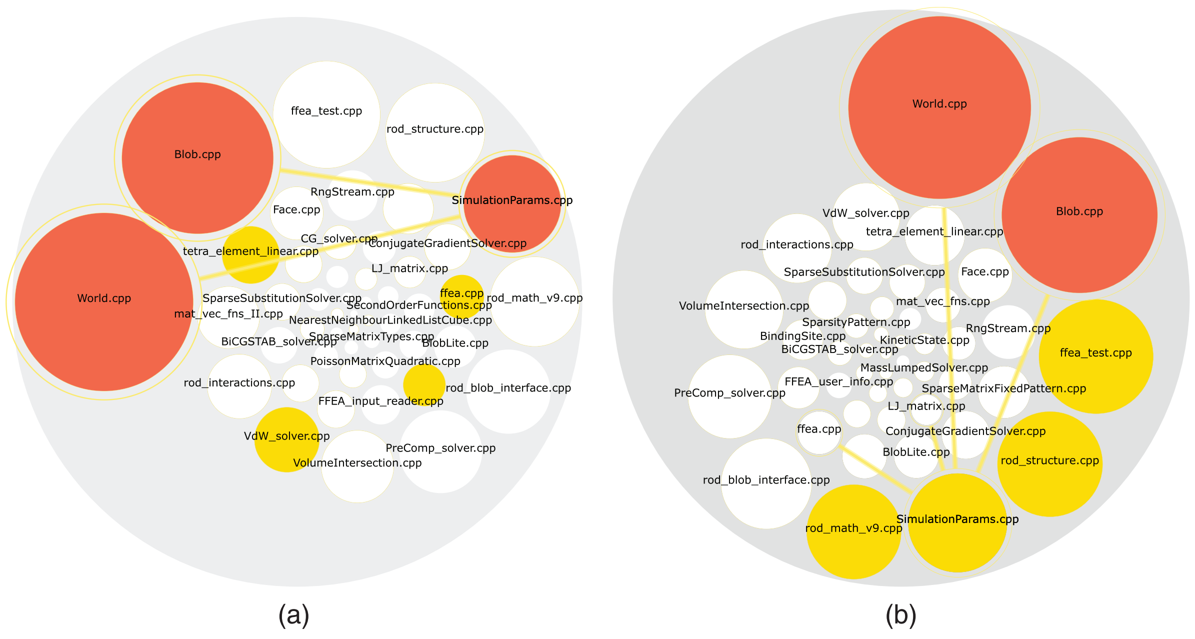


FIGURE 4. FFEA: CodeScene-detected refactoring targets in `ffea` module. Refactoring reduced the number of priority refactoring targets from three (a) to two (b).

updating and simplifying the build configuration, adopting modern C++ standards, resolving failing and flaky tests, addressing memory access violations, and reducing code duplication (Figure 4). However, enhancements to FFEA's high-level architecture were constrained by inherent domain complexity and the absence of adequate documentation or testing, which hindered both a thorough understanding of the codebase and confidence in implementing significant changes.

Static analysis of FFEA's source, with SonarQube Cloud, was performed at the start and end of this project. The number of issues identified within the source code decreased from 5014 to 2233, in particular the number of high priority issues dropped from 1200 to 385. The type of issues identified by this static analysis largely inherit from the C++ Core Guidelines,[l] which provide modern best practices for writing safe, efficient, and maintainable C++ code. They range from high-priority issues, such as using modern techniques that limit code misuse, to low priority style recommendations that add clarity to make the code easier for users to parse.

In contrast, SonarQube Cloud's measures of cyclomatic complexity and cognitive complexity have not changed significantly. For example, `src/Blob.cpp`, which dropped from 640 to 174 issues, only saw its cyclomatic complexity score drop from 663 to 583, and its cognitive complexity drop from 932 to 830. Similarly, test coverage at the end of the project was still low at 61% of lines and 37% of branches. This confirms our interpretation that architectural complexity is largely distinct from code-level smells. Many code smells may reduce the safety and readability of a localized area of code. However, addressing these has limited impact to improving the maintainability of the whole codebase, where issues are far more challenging to address at a late stage.

[l] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

DISCUSSION

Static code analysis tools are helpful in analyzing specific aspects of software architecture. Metrics, such as cyclomatic and cognitive complexity, as well as some types of issues, such as code smells, relate to the understandability and maintainability of a codebase. However, not all relevant metrics are provided by established static code analysis tools. More complex metrics related to architecture are often context-dependent or hard to quantify, or require the use of additional tools providing more in-depth analysis for specific architectural measurements or specific programming languages. Metrics for more easily measurable attributes are often readily available, but their improvement does not necessarily have a positive impact on architecturally relevant attributes. Additionally, while metrics can serve as a starting point for architectural analysis, more comprehensive insights must be gained from other sources, such as design documents or developers.

Considering architecture and software design as important parts of research software engineering is essential through the complete lifetime of a research software project. Failure to do this can lead to accidental architectures and increasingly unmaintainable code, especially in research software projects with a long lifetime. Continuously monitoring the software maintainability is feasible through integrating static code analysis into continuous integration pipelines. Open source static code analysis tools exist (e.g., PMD). Some platforms are free for open source projects or academic use. In our examples, static code analysis tools provided leads for the identification of potential architectural issues, even if they failed to provide the relevant metrics themselves.

Where new developers regularly join the project (e.g., as Ph.D. students), it is important to provide documentation and implement decision and onboarding processes that include a discussion of software architecture, e.g., via lightweight architectural decision records.

Existing architectural patterns and concepts, such as modularization, support maintainability, scalability, and robustness of research software and can have a positive impact on software sustainability and the reproducibility of research results. Using existing frameworks that enforce modularization can be helpful in achieving solid software architecture, although resources must be available to assess advantages and disadvantages of using these frameworks, and to learn how they work.

It falls upon software engineering research to develop generalizable metrics that provide relevant information about software architecture, and develop cost-effective and usable open source tools that help RSEs evaluate the architecture of their software projects. To achieve this, software engineering researchers need to work more closely with RSEs to understand their unique needs with respect to the heterogeneity of RSEs' educational backgrounds, technical experience and domain-relatedness. Additionally, RSEs often operate under resource constraints that do not favor a focus on maintainability, refactoring activities without visible results in the form of research outputs, or the acquisition of expensive tools whose necessity may not be immediately clear to decision makers. However, there is still a pressing need for new tooling to fit today's emergent and dynamic environments, where essential research software is explicitly designed for continuous evolvability and adaptability without incurring prohibitive architectural technical debt.

CONCLUSION

Considering software architecture in research software engineering safeguards its robustness, maintainability, scalability, and evolvability. This is important in research, where teams often change and developers have diverse backgrounds and expertise, and research objectives progress.

We presented two examples for the architecture evaluation of research software projects using software metrics. Our examples showed that refactoring activities to remove code-level issues do not necessarily have significant positive impact on the complexity and maintainability of research software. Metrics can, however, provide leads for identifying architectural issues. Factors that positively influence the architectural quality of software included the use of modularization frameworks, and a consistent focus on software quality in research software projects.

Practical Implications

In more actionable terms, to achieve better software through better architecture, and better research through better software, we recommend for new research software projects that RSEs spend time considering potentially suitable designs and design principles for the software they will implement, taking into account maintainability, extensibility, adaptability, and reusability. This consideration can be supported by the software architecture literature and by assessment of available modularization frameworks. Some programming languages provide standard mechanisms for modularization, such as Python's extension points. Regardless of the technical solution for modularization, modules should encapsulate semantically connected code and use interfaces for interoperation with other modules to minimize coupling and maximize cohesion.

Outcomes of architectural decision processes should be documented in a way that suits the developers. RSEs should not be concerned with getting everything "right" at the beginning of development, but should be aware of balancing the short-term needs of the research with the longer-term implications for the software. Architecture and software design can be iterated alongside the development process, where new insights into the problem domain may trigger a design iteration. The software's *role* in research and the target *technology readiness level*[8] can guide iterations: The higher the technology readiness level and the wider the potential reuse, the more important intentional, conscious software design and architecture become.

To support architecture iterations, we recommend that RSEs use some automation from the start. Static code analysis can be run in continuous integration, e.g., using the platforms mentioned above. Even if analysis metrics do not provide comprehensive architectural insights or flag all relevant design smells, they can uncover issues at code level and act as regular reminders to consider architectural (and code) quality, and to reflect design decisions. One prerequisite for using such metrics is that RSEs reflect the metrics' meaning and expressive value: While they are easy to measure and track, they may not provide a comprehensive picture of architectural quality. Changes in software design and architecture should always be documented in a way that retains design knowledge independently of individuals.

For developers joining legacy research software projects, we recommend that static code analysis metrics be used by RSEs as an entry point to an architectural analysis of the software they will work on. Higher-level architectural views, such as those presented in Figure 2(d) and (e), can support a basic understanding of the modularity of hitherto unfamiliar software. Building on this, they should then consult existing design documentation where available.

Software engineering researchers should work closely with RSEs to understand their unique needs, and develop suitable metrics and usable open source tools to support them.

In any case, considering the design and architecture of research software, learning and following basic good practices, and leveraging existing tools that support this effort is time well spent. It may save RSEs time and effort over the software lifecycle, and helps making research software more maintainable, extensible, adaptable, and reusable, ultimately improving the quality of research outcomes.

ACKNOWLEDGMENTS

The authors thank Ben Hanson (University of Leeds, U.K.) and Molly Gravett (Utrecht University, The Netherlands) for providing the fluctuating finite element analysis models of cytoplasmic dynein and myosin 5a, respectively. The images in Figure 3 were captured using the PyMOL (https://pymol.org) molecular visualisation system. The authors were supported by the German Research Foundation (DFG), Grants 391160252 (S.D. and T.K.) and 528713834 (W.H.), the U.K. Research and Innovation (N.P.C.H., Grants EP/S021779/1 and AH/Z000114/1 for the Software Sustainability Institute), the Wellcome Trust (Grant 222381/Z/21/Z), and the University of Sheffield, and by the U.S. Department of Energy Office of Science, Biological and Environmental Research division Award DE-SC0025510.

REFERENCES

1. M. Gruenpeter et al. "Defining research software: A controversial discussion." Zenodo, doi: 10.5281/zenodo.5504016.
2. N. P. Chue Hong et al., "FAIR principles for research software (FAIR4RS principles)." Zenodo, doi: 10.15497/RDA00068.
3. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Reading, MA, USA: Addison-Wesley Professional, 2021.
4. C. Lilienthal, *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. San Rafael, CA, USA: Rocky Nook, 2019.
5. C. C. Venters et al., "Sustainable software engineering: Reflections on advances in research and practice," *Inf. Softw. Technol.*, vol. 164, Dec. 2023, Art. no. 107316, doi: https://doi.org/10.1016/j.infsof.2023.107316.
6. S. Hettrick et al. "International RSE survey 2022." Zenodo, doi: 10.5281/zenodo.7015772.
7. N. U. Eisty et al., "Use of software process in research software development: A survey," in *Proc. 23rd Int. Conf. Eval. Assessment Softw. Eng.*, New York, NY, USA: ACM, 2019, pp. 276–282, doi: 10.1145/3319008.3319351.
8. W. Hasselbring et al., "Multi-dimensional research software categorization," *Comput. Sci. Eng.*, early access, Mar. 27, 2025, doi: 10.1109/MCSE.2025.3555023.
9. W. Hasselbring, "Software architecture: Past, present, future," in *The Essence of Software Engineering*, V. Gruhn and R. Striemer, Eds., Cham, Switzerland: Springer-Verlag, 2018, pp. 169–184, doi: 10.1007/978-3-319-73897-0_10.
10. S. Druskat, T. Krause, C. Lachenmaier, and B. Bunzeck, "Hexatomic: An extensible, OS-independent platform for deep multi-layer linguistic annotation of corpora," *J. Open Source Softw.*, vol. 8, no. 86, Art. no. 4825, Jun. 2023, doi: 10.21105/joss.04825.
11. A. Solernou et al., "Fluctuating finite element analysis (FFEA): A continuum mechanics software tool for mesoscale simulation of biomolecules," *PLoS Comput. Biol.*, vol. 14, no. 3, Mar. 2018, Art. no. e1005897, doi: 10.1371/journal.pcbi.1005897.

12. R. O. Dror et al., “Biomolecular simulation: A computational microscope for molecular biology,” *Annu. Rev. Biophys.*, vol. 41, no. 1, pp. 429–452, Jun. 2012, doi: 10.1146/annurev-biophys-042910-155245.

STEPHAN DRUSKAT is a software engineering researcher at the German Aerospace Center (DLR), 12489, Berlin, Germany, and Fellow of the Software Sustainability Institute. His research interests include research software sustainability, empirical and evidence-based (research) software engineering and research software intelligence. He is a co-founder of the German Society for Research Software, and member of the Society for Research Software Engineering and the German Association for Computer Science, where he co-founded the special interest group on research software engineering. Druskat received his M.A degree in English philology, modern German literature and linguistics from the Free University Berlin. Contact him at stephan.druskat@dlr.de.

NASIR U. EISTY is an assistant professor of computer science at the University of Tennessee, Knoxville, 39716, TN, USA. His research interests include software engineering, AI for software engineering, research software engineering, and software security. Eisty received his Ph.D degree in computer science from the University of Alabama. Contact him at neisty@utk.edu.

ROBERT CHISHOLM is a research software engineer in the School of Computer Science at the University of Sheffield, S1 4DP, Sheffield, U.K. His research interests include performance optimization and parallel computing. Chisholm received his Ph.D. degree in computer science from the University of Sheffield. Contact him at robert.chisholm@sheffield.ac.uk.

NEIL P. CHUE HONG is a professor of research software policy and practice at the University of Edinburgh, EH8 9BT, Edinburgh, U.K. and the director of the Software Sustainability Institute. His research interests include understanding the way in which specialist software used in research is developed and how policy and incentives can be used to improve maintenance and reusability. Hong received his M.Phys. degree in computational physics from the University of Edinburgh. Contact him at N.ChueHong@epcc.ed.ac.uk.

RYAN C. COCKING is a research software engineer in the School of Mathematical and Physical Sciences at the University of Sheffield, S3 7RH, Sheffield, U.K. His research interests include coarse-grained biomolecular simulations and software engineering. Cocking received his Ph.D. degree in computational biophysics from the University of Leeds. Contact him at ryan.cocking@sheffield.ac.uk.

MYRA B. COHEN is a professor and the Lanh and Oanh Nguyen Chair in Software Engineering in the Department of Computer Science at Iowa State University, Ames, IA, 50011, USA. Her research interests are in software testing, search-based software engineering, and correctness of scientific software. Cohen received her Ph.D. degree from the University of Auckland, New Zealand. Contact her at mcohen@iastate.edu.

MICHAEL FELDERER is the director of the Institute of Software Technology at German Aerospace Center (DLR), 51147, Cologne, Germany, and a full professor at the University of Cologne. His research interests include software engineering, artificial intelligence, and systems engineering. Felderer received his Ph.D. degree in computer science from the University of Innsbruck. He is a member of the Association for Computing Machinery and the German Association for Computer Science. Contact him at michael.felderer@dlr.de.

LARS GRUNSKE is a professor in the Department of Computer Science from the Humboldt-Universität zu Berlin, 10099, Berlin, Germany. His research interests include automated software engineering, formal methods and research software engineering research. Grunshe received his Ph.D. degree in computer science from the University of Potsdam (Hasso-Plattner-Institute for Software Systems Engineering). Contact him at grunske@informatik.hu-berlin.de.

SARAH A. HARRIS is a professor of biological and materials physics in the School of Mathematical and Physical Sciences at the University of Sheffield, S3 7RH, Sheffield, U.K. Her research interests include multiscale modeling, biomolecular simulation, and molecular recognition. Harris received her Ph.D. degree in computational biophysics from the University of Nottingham. She and her colleagues developed FFEA to perform biomolecular simulations at mesoscopic length and timescales. Contact her at sarah.harris@sheffield.ac.uk.

WILHELM HASSELBRING is a professor of software engineering at Kiel University, 24098, Kiel, Germany, and an adjunct professor at the University of Southampton, SO17 1BJ, Southampton, U.K. His research interests include software engineering, distributed systems, and open science. Hasselbring received his Ph.D.

degree in computer science from the University of Dortmund. He is a member of the Association for Computing Machinery, IEEE Computer Society, and the German Association for Computer Science, at which he is vice chair of the special interest group on research software engineering. Contact him at hasselbring@ email.uni-kiel.de.

THOMAS KRAUSE is a researcher/research software engineer in the Department of German Studies and Linguistics at the Humboldt-Universität zu Berlin, 10099, Berlin, Germany. His research interests include developing methods and software for corpus linguistics to represent, analyze, and visualize linguistic annotations. Krause received his Ph.D. degree in computer science from Humboldt-Universität zu Berlin. Contact him at thomas.krause@hu-berlin.de.

JAN LINXWEILER is the general manager of the Center for Mechanics, Uncertainty and Simulation in Engineering (MUSEN) at TU Braunschweig and head of IT and Research Services at the University library, Braunschweig, 38106, Braunschweig, Germany. His research interests include research software engineering research, high performance computing, research data management, and open ccience. Linxweiler received his Ph.D. degree in engineering and is a founding member of the German RSE association, for which he also serves as chairman of the board. Contact him at j.linxweiler@tu-braunschweig.de.

COLIN C. VENTERS is a research software engineer at European Organization for Nuclear Research, 1211, Geneva 23, Switzerland associated with the ATLAS experiment. His research interests include sustainable software engineering from a software architecture perspective. Venters received his Ph.D. degree from the University of Manchester, United Kingdom. He is a founding member of the Sustainability Design Alliance, and co-author of the Karlskrona Manifesto for Sustainability Design. Contact him at c.venters@cern.ch.