Harvard Data Science Review • Issue 5.2, Spring 2023

Software Engineering Practices in Academia: Promoting the 3Rs—Readability, Resilience, and Reuse

Andrew Connolly[1] Joseph Hellerstein[1] Naomi Alterman[1] David Beck[1] Rob Fatland[1] Ed Lazowska[1] Vani Mandava[1] Sarah Stone[1]

[1]eScience Institute, University of Washington, Seattle, Washington, United States of America

Published on: Apr 27, 2023

DOI: https://doi.org/10.1162/99608f92.018bf012

ABSTRACT

Over the past decade as data science has become integral to the research workflow, we, like many others, have learned that good data science requires high-quality software engineering. Unfortunately, our experience is that many data science projects can be limited by the absence of software engineering processes. We advocate that data science projects should incorporate what we call the 3Rs of software engineering: readability (human understandable codes), resilience (fails rarely/gracefully), and reuse (can easily be used by others and can be embedded in other software). This article discusses engineering practices that promote 3R software in academia. We emphasize that best practices in academia may differ from those in industry because of substantial differences in project scope (most academic projects have a single developer who is the sole user) and the reward systems in place in academia. We provide a framework for selecting a level of software engineering rigor that aligns well with the project scope, something that may change over time. We further discuss how to improve training in software engineering skills in an academic environment and how to build communities of practice that span across disciplines.

Keywords: software engineering, reproducibility, reuse, data science education

Media Summary

Data science is about acquiring, interpreting, and using data — all of which rely on software. Industry has skilled professionals who can provide the software necessary for high quality data science. But academic data science projects are often staffed with researchers with few skills in software engineering, especially the ability to produce readable, resilient, and reusable software. This is what we call the 3Rs of software engineering.

For the last decade, the eScience Institute at the University of Washington has been training data scientists and developing data science technologies. Thousands of undergraduate and graduate students, along with numerous faculty, have participated in our programs, training, and formal classes. As a result, we have developed a considerable understanding about the software engineering principles that promote effective data science in academia.

We find that while *some* academic projects require industrial strength software engineering, most projects do not. We recommend adjusting software engineering practices to the scope of the project. We classify the scope of academic projects as: solo (a single researcher creates and uses the project); lab (the developers and users know each other and are in close contact); and community (developers have limited knowledge of the users).

The centerpiece of this article is how to adjust software engineering practices to the project scope. For example, a solo project should include unit tests, but need not invest in packaging for software distribution. A lab project should include packaging considerations, but may not need formal design documentation. A

community project should abide by the same practices as those employed by industry software engineers. These recommendations are detailed in our discussion, with references to help with applying these practices.

The eScience Institute is developing a centralized team of software engineers who provide strategic support to research projects. In the future, we will report on the effectiveness of our efforts, and the best practices and challenges with developing and operating such a team.

---

1. Good Data Science Requires High-Quality Software Engineering

As data science has become ever more integrated within research, our reliance on software and its development has grown substantially. The 'data' part of data science relies heavily on the development of readable, resilient, and reusable software to create, analyze, and visualize data (e.g., Jansses, 2014). The research we undertake and the discoveries we make are becoming ever more dependent on robust and reproducible software frameworks and development processes. This reliance is, however, not unique to data science. In a study of researchers in the United Kingdom (Osimo & Switters, 2019), 70% of respondents said that their research would be impossible without the use of software, and 56% said that they develop their own software. Indeed, Hettrick (2014) found that without software, more than 80% of the academic respondents would either be unable or would find it extremely difficult to do their research.

Despite the rapid evolution in software-driven science, academic institutions have not kept pace with the growing need for software engineering skills. In a survey of principal investigators at the University of Washington, 30% of respondents said they address their software development needs by hiring graduate students, often without any formal training in software engineering. Hiring students new to the best practices for writing software as software engineers does not, in our experience, typically produce robust software that can live beyond the lifespan of a typical PhD. Further, software developed in this way often cannot be used by a broader community. Training in software engineering for non-CS (computer science) majors has begun to emerge within some university CS programs, but as we note in later sections, these programs are not focused on the needs of non-CS researchers because they do not emphasize the practice of software engineering. Addressing this challenge has implications for workforce development within the United States and the creation of a software engineering talent pool within the research community.

At times, our discussion uses technical terms that may be unfamiliar to some readers. To make this article accessible to a wider audience, we include a glossary of terms in Appendix A. When a glossary term is first introduced, it is underlined (e.g., engineering practice).

The lack of software engineering expertise on campuses will likely impact research productivity over the coming years. To address this, we believe that researchers should have the skills to develop what we refer to as **the 3Rs of software engineering**:

- **Readability**. We mean that software is written to promote understanding by others (e.g., good comments and naming conventions). Readability is essential to providing scientific results that are reproducible by others. Readability is also required to make software extensible and reusable.
- **Resilient**. We mean that a system fails rarely or, in the context of software systems, that when it does fail due to adverse inputs or failures in components of the system, it does so 'gracefully' (e.g., it operates even when the system is degraded). Resilience requires testing for common error conditions (e.g., good unit tests). Resilience is required to give users and developers confidence that they can rely on the software.
- **Reusable**. We mean that others can make use of the code as is, without extensive rewriting (e.g., through the use of modular structures). Beyond readability and resilience, reuse requires that software be modular and distributed in a way that is easy to install and use.

As we discuss in this article, not all academic software needs to be at the high end of the 3Rs, but all developers of academic software should have the understanding to produce software with the 3Rs.

A theme throughout this article is *employing an appropriate level of software engineering for the project scope*. Milewicz et al. (2019) notes that the vast majority of academic software projects have a single developer and a single user. Such solo projects are common in student thesis work and for software written to analyze data for a publication. A modest number of projects have multiple users, typically software written for a research lab such as a data reduction pipeline. A small fraction of projects are in widespread use in a research community. For these projects, significant software engineering rigor is necessary.

Project scope can change, both expanding and contracting. As such, we strongly believe that all developers of academic software should know how to build 3R software, even though high-end 3R software is unnecessary in most academic projects. Unfortunately, 3R skills are difficult to acquire in current academic course work outside of CS and related fields (Fox & Patterson, 2012).

In this article, we further address how to bring more software engineering skills into academia through improved training programs, and the creation of career trajectories for research software engineers (RSEs), professional software engineers working in an academic setting. There is a growing recognition of the need for RSEs. One indication of this need is the creation of the United States Research Software Engineering organization (USRSE, n.d.). Feeding the RSE pipeline requires universities to develop courses and programs to train students in software engineering practices. Here we consider approaches and challenges with developing these courses at the University of Washington and elsewhere. Finally, we explore what a future software engineering and a software engineering community might look like at a research university.

2. A Tale of Two Academic Software Projects

To gain more insight into the nature of and challenges with the development of academic software, we discuss the history of two academic software projects. These projects are atypical in that their software is in use by a broad community, and both evolved over a period of two decades. As noted previously, most academic

software is developed and used by a single researcher. That said, the success of the projects described below is in part due to adopting tools and best practices for creating 3R software. Solo projects should be aware of how they can evolve their software to potentially address a broader community.

2.1. Tellurium

With the advent of inexpensive genomic sequencing and related advances in laboratory techniques, biological science and engineering has increasingly focused on quantitative techniques. One aspect of this is the development of mechanistic models that predict cellular processes, like the concentration of glucose and related metabolites. These models are expressed as a network of chemical reactions. Each reaction describes the transfer of mass from reactants to products and the rate at which this occurs. Evaluating such a model requires evaluating a system of nonlinear ordinary differential equations. Historically, such models were developed as special purpose simulators written in the C or FORTRAN programming languages. This was problematic because (a) few modelers could program in C or FORTRAN and (b) there was little reuse of common components (e.g., transforming a reaction network into a system of differential equations). As time passed, there was a recognition that biochemical simulators could be developed more rapidly if there were tools that incorporated commonly used capabilities.

The Tellurium project (Choi et al., 2018) is the latest evolution of a long-standing effort by the Sys-Bio Research Group at the University of Washington to develop mechanistic models of biochemical networks. The JARNAC project (Sauro et al., 2003) tried to make this modeling accessible to bench scientists by allowing users to enter reactions in chemical notation instead of having them write computer codes. Unfortunately, the system was quite slow. Thus, the project evolved into two parts: the RoadRunner execution engine (Somogyi et al., 2015) and the Antimony (Smith et al., 2009) reaction specification language. This strategy proved successful. The project expanded to support Windows, Mac, and Linux; and it provides clients for Python and C.

As the project scope evolved, so did the people within it. In the beginning, this was a solo project by Herbert M. Sauro (2018). It was a critical period because the future success of the project depended on a single person. Later, another person was added to build Antimony. Subsequently, others were included to support multiple platforms.

Software engineering practices evolved in parallel with the project scope. This was because as more people joined the team, more formality was required to maintain code integrity (e.g., manage source code conflicts). A central consideration was controlling versions of software contributed to by many developers. The ‘branch’ capabilities of git (Loeliger & McCullough, 2012) proved critical here. Early on, unit tests were not used at all, but the project has added extensive unit tests in recent years, and this has improved resilience. Since Tellurium is intended for programmatic use, detailed documentation is essential. A very recent change is using Read the

Docs (n.d.) for software documentation along with an automated system for documenting Tellurium application programming interfaces.

2.2. Astropy

Astrophysics has a long history of community software development extending back to the 1980s. These initiatives have been orchestrated by large organizations (e.g., those funded through government support such as IRAF [Image Reduction and Analysis Facility]; Tody, 1986, and Starlink; Currie, 2014) and by smaller groups of users who built domain-specific applications (e.g., the analysis package for gamma-ray astronomy, gammapy; Deil et al., 2017). More recently, a third development strategy has emerged in which packages (or frameworks) of larger scope are created through the integration of many smaller packages. Astropy (Price-Whelan et al., 2018), created in 2011, is an example of such a strategy.

Astropy was motivated by the rise of Python as a lingua franca in astronomy. A central goal of the Astropy Project was to provide consistency and completeness for common calculations and tools used by astronomers. Examples of these tools include unit conversions, the manipulation of sky coordinates (e.g., transforming from Galactic coordinates to Right Ascension and Declination), and software to read and write common astronomical data formats.

The packages integrated into Astropy were, in large part, developed by researchers well versed in software engineering practices. For example, constituent packages made use of version control via GitHub, had unit tests for the core libraries and functions, and issued pull requests as a means of developing new features. Packages were distributed using common software repositories such as PyPI (Python Package Index). Tools for continuous integration (e.g., Travis CI and Jenkins) were adopted early in the development of Astropy to improve the reliability and robustness of the software. This solid engineering foundation greatly facilitated the construction of Astropy and its adoption by the community.

Even with this engineering foundation, package integration was nontrivial because of the need for common abstractions across packages. An example of this is the sub-package astropy.units, which provides a representation of physical units used in astrophysics, enables translation between units, and has the ability to decompose complex parameters (e.g., the Hubble parameter) into their base units (i.e., inverse time). As with many early Astropy packages, the units package was developed from an existing application that had introduced units to cosmological simulation software and was then extended to support the needs of the broader astronomical community. The lack of existing standards within astronomy for units led to the inclusion of all available standards within the package to make it as general as possible. Functionality to translate between conventions enabled the units package to provide general support without forcing the community to agree on a set of standards. This ‘ease of use’ philosophy underlined many of the Astropy design choices.

Building the community of Astropy users, maintainers, and developers required convincing astronomers with little or no formal training in software engineering to adopt these standard tools and procedures if they wanted

to contribute to the code base. With little financial resources, the training and education of the user community was supported by the Astropy developers themselves. The availability of GitHub, on which Astropy was built, provided the tools and infrastructure on which to develop community-agreed engineering practices for version control, issue tracking, and communication. The availability of Infrastructure tools and repositories such as GitHub are key to the sustainability of software projects in astronomy and enable common approaches to be adopted within a community.

3. 3R Software Engineering Practices

The foregoing academic software projects, while different in size and scope, show how good software engineering practices can increase the adoption and trust of software packages beyond the developers who wrote them. Based on this and our experiences in software development, we have developed a set of recommendations for software engineering practices that aid in the development of 3R software. We do this with great humility since software engineering is a field with a long history and a vast literature (see Boehm, 1976; Glass et al., 2002). A good starting point for this literature is on artifact sharing (Timperley et al., 2020). By engineering practice, we mean a collection of related activities used in building, evolving, and managing software systems. Examples of engineering activities include coding, quality assurance, and distributing software. We use the term artifact to refer to work products of software engineering, especially code, documentation, and data. We note in passing that data science produces other artifacts, such as predictive models and analysis pipelines. These artifacts are beyond the scope of this article.

Our recommendations come in part from the experience of members of the eScience Institute at the University of Washington and discussions. eScience has more than 20 technical staff, almost all of whom have a PhD in a primary discipline such as Computer Science, Physics, Human Centered Design, Statistics, and Chemistry. Technical staff are engaged in many educational programs. Some teach formal courses in their departments. Others orchestrate and/or instruct Software Carpentries. eScience oversees the development of data science curricula and courses at the graduate and undergraduate level. We also have an extensive outreach program for on-campus researchers, our Winter Incubator in which domain researchers dedicate 16 hours per week for a quarter to work with a member of eScience technical staff. In the summer we run a program for matriculating undergraduates and graduate students from across the county (and even globally) to undertake data science projects for social good (DSSG). Beyond this, we offer approximately 20 hours per week of office hours to researchers who seek focused consultations with technical staff. These interactions provided us with extensive insights into the challenges encountered in academic projects of varying size and duration.

The engineering practices we propose combine the collective insights of eScience technical staff with feedback from software engineers in industry and at national laboratories as well as researchers who teach software engineering in an academic environment. Where possible, we point to published recommendations and draw on work from the field of software architecture and development practices. The vast majority of this literature focuses on team processes for complex projects, with only a modest discussion of software development

(actually building software) and almost nothing on maintaining and extending existing academic software. Formal publications in this area include: Beck et al. (1983), Feller et al. (2007), Hooley et al. (2021), and Poppendieck & Poppendieck (2003). In addition, detailed guidance is available in numerous blog posts (Postan, 2021) and Stack Overflow replies (Stack, 2022), but the provenance of this advice is often limited.

From our experience, since many academic projects are short-term and of small scale, it is appropriate to apply a level of engineering rigor that is commensurate with the project scope. An important consideration is, however, to highlight the requirements for transitioning a project to a larger scope. We provide recommendations of best practices and how these practices may evolve as a project moves from a single developer, to use within a self-contained team, and then possibly to broad adoption by a research community. Since our interest is in data science, we focus on Python and R, the most widely used languages in data science.

3.1. Engineering Practices and Their Interactions

A central theme in this article is that engineering practices should be scaled to the project scope. That is, in smaller projects, a practice may be greatly simplified or absent altogether. However, if a project grows, there must be awareness of how to incorporate engineering practices that were not considered previously. In the following, we describe how engineering practices need to evolve as projects grow. Although we have recommendations for what practices to change, there is less agreement from the use cases we have studied about what should trigger a change in engineering practices or how to build consensus within a developer community to adopt these changes. We refer readers to studies on this topic related to the adoption of developer tools (Brooke Jordan, 2014) and security analysis (Jaspan et al., 2007).

Broadly, there are technical and people management activities (which we use synonymously with practices) within software engineering. Technical practices produce code, data, and documentation of the software internals. Refining this further, the production of code and data includes design, quality assurance (e.g., testing), and packaging and deployment. People management activities address coordination and communication within the project and communication between project developers and users. The people management practices have associated artifacts as well, such as project plans and prioritized lists of features and fixes.

The nature of engineering practices depends strongly on the scope of the project. An example of a project with a small scope is a short-term exploratory effort by a single researcher. In contrast, a project with large scope often involves multiple teams at different locations. We consider three project scopes:

- **Developing for your own use (solo)**. Our experience is that the vast majority of academic software projects consist of a single developer who is the sole user of the software. These projects are often undertaken as part of a research exploration. Few academic projects advance beyond this stage.
- **Developing for your research lab (lab)**. Many researchers work in teams. They often find that the problem solved by their software can be used by others in their team. In these projects, developers and users are in

frequent contact.

- **Developing for a broad research community (community)**. There are a small number of projects that are used by a broader research community and/or of sufficient technical scope that a large team is required.

We emphasize that the boundaries between these scopes are fluid. For example, a solo project may evolve into a lab project, and the reverse can happen as well.

3.2 Details of Engineering Practices

There is a vast literature on software engineering and engineering practices. In this section, we describe a subset of these practices that we feel are most relevant to data science. These practices relate to: version control, design, coding, quality assurance, packaging and deployment, user documentation, team management, and user engagement. We organize the discussion by project scope (including some references to more in-depth discussions of these software engineering practices). For each topic, there are two bullets. The first describes what the practice is and why it is important; the second bullet outlines some recommended tools and best practices.

We begin with ***version control*** (Blokdyk, 2022).

- **What**: Version control deals with tracking changes to artifacts (e.g., code, documents, data) in shared collections of files called repositories. Version control is an essential part of making software resilient and reusable.
- **How**: For software, services such as GitHub (Ponuthorai, 2023) (Wikipedia, n.d.-e) allow users to have a code repository where changes can be viewed. Commonly used features are (a) undoing a change that introduced an error and (b) coordinating changes among multiple developers. Another widely used feature is a version control 'branch' that allows developers to make changes in parallel (and also facilities managing experimental data). A solo project needs version control to ensure that the code is not lost and to revert to previous versions if a bug is introduced. They enable experimentation and exploration of new ideas without impacting the primary or main branch. A lab project has additional requirements, such as resolving 'change conflicts' (changes to the same line in a file by different developers). In a community project, more formal coordination is done to manage releases, develop new packages or features based on the initial code base (i.e., forks), integrate codes from other groups, and handle urgent bug fixes ('hot fixes') that are done between formal releases.

Software design (Keeling, 2017) is at the core of creating resilient (e.g., by early consideration of error conditions) and reusable software (e.g., by a modular design).

- **What**: There are multiple components to software design of which we consider two critical for data science applications. First is the design of the user experience or use cases, often referred to as functional design.

This is about how a user interacts with the system to accomplish their objectives. The second is component design. This specifies how to create and interconnect software artifacts that perform the use cases.

- **How**: Appendices B and C contain simplified templates that we developed for functional and component design that we use in CSE 583 (University of Washington, 2023-b) and DATA 515A (University of Washington, 2023-c) at the University of Washington and in CHEME 545 and 546 (University of Washington, 2023-a). The functional design specifies a set of use cases that are detailed descriptions of user interactions with the software system. Appendix B contains a template for functional design. A component design can be expressed in many ways, such as: a data flow diagram (Li & Chen, 2009), UML diagrams that describe objects with properties and behaviors (Fowler, 2003), and entity-relationship diagrams (Li & Chen, 2009). Appendix C contains a template for component design. In solo projects, design may be done informally (e.g., in a notebook). In a lab project, there is often some discussion that requires a shared white board and sometimes an informal write-up. In community projects, more formality is required, such as a standard template for function and component design documents.

Computer programming or coding (Kerninghan & Pike, 1999) is the process of writing detailed instructions so that a computer can perform a desired task.

- **What**: Coding practices encompass a wide range of decisions, such as: (a) the choice of names for variables, functions, and modules; (b) documentation practices in modules and functions; (c) the use language features (e.g., the choice between a `for` statement versus a comprehension in Python); and (d) the choice of data structures (e.g., when to use a `list` vs. an `array` vs. a `dict`). A further consideration are software licenses (Laurent, 2004).
- **How**: Over the last 50 years, programming has evolved into a systematic engineering activity with powerful productivity tools. Examples of such tools are integrated development environments (IDEs) for Python (e.g., PyCharm; Nguyen, 2019) and R (e.g., Allaire, 2011). In recent years there has been a trend toward "literate programming" (Knuth, 1992), especially 'notebooks' (e.g., Jupyter) that intermix code with text to provide a narrative for an analysis. For a solo project, readability is greatly improved by providing notes about decisions made (e.g., a GitHub README file or within the notebook) as well as the use of consistent naming conventions to facilitate understanding codes written months earlier. In a lab project, readability and resilience are enhanced agreement on common data structures and coding styles (with tools such as linters to enforce style). A community project often takes this a step further by having code reviews in which developers explain their motivations for engineering decisions and reviewers advise on approaches to improve reuse.

In industry, ***quality assurance*** (Patton, 2005) is a very broad term that encompasses the entire engineering process.

- **What**: Quality assurance is about ensuring resilience, good performance, security, and privacy.

- **How**: For academic projects, the focus is mostly about testing for errors at various levels. For a solo project, it likely means implementing unit tests (codes that check for errors in functions and methods) for key elements of the project. Excellent open source packages are available to enable these tests (e.g., Python unittest and R testthat). For a lab project, unit tests are more extensive, and there is continuous integration (e.g., run all unit tests after every commit to the software repository). A community project likely includes additional quality tests for each software release to ensure there is no 'regression' in future releases. (See Nielsen, 2000, for a more detailed discussion.)

Packaging and deployment (e.g., Waldon, 2012) are activities that make software available to users, an essential element of making software reusable.

- **What**: The goal is to make software developed by one user available to other users. This often requires that the software developer structure their codes into a package that can be shared with other users. Users may have different software installed on their computers, even different operating systems. So, the package must specify its dependencies, such as a particular version of the Python library `numpy`. This raises a further challenge that two packages may have conflicting requirements, such as different versions of `numpy`.
- **How**: Most academic projects use an install model of package deployment in which the user's computer is updated to incorporate the software. PyPI is the most common mechanism for distributing Python packages, and CRAN is widely used for R packages. Other software repositories include Conda and SourceForge. There are also service models for software distribution in which the software runs on servers owned by the provider and users are not aware of the software updates (e.g., Gmail). Still another approach is container-based distribution (e.g., Docker). For a solo project, there may be no packaging and deployment since the code runs on a single machine for a single user, but to support the reproducibility of the research a well-defined and reproducible development environment can be critical. For a lab project, it is common that all machines in the lab run an almost identical software stack (e.g., the same version of Linux and Python packages), and often codes are relatively machine independent (e.g., Python, R); so, deployment is done via PyPI for Python and CRAN for R (with their associated packaging requirements). A recent trend is to use virtual machines in the cloud so that even if physical machines have different software, the virtual machines are identical. A community project often involves multiple languages and hardware platforms, and so packaging and distribution is more complex. One such complexity is that quality assurance must include testing of packaging and installs.

User documentation (Bhatti, 2021) is the written descriptions that accompany a software package so that nondevelopers can effectively use the software, a key consideration in building reusable software.

- **What**: User documentation covers installation, basic usage, and a detailed reference manual for advanced users. For example, a screen scraper application might specify a command line to install the tool, illustrate its usage on a page from *The New York Times*, and point to detailed documentation on options for different kinds of web pages.

- **How**: Solo projects have modest needs here, mostly to ensure that the developer can easily recall how to use their software some months after it was written (and the methods or research papers underlying its development). In a lab project, developers may provide a 'help' option for command line tools and/or a one-page summary of usage (a 'manual page,' Linux, 2009.) or a Jupyter Notebook (Ragan-Kelley et al., 2014). In community projects, there are more extensive capabilities (e.g., Read the Docs; Cotton, 2016) that contain detailed descriptions of the software features, examples, and capabilities for searching documentation.

We use the term ***team management*** (Project Management Institute, 2017) to refer to those aspects of project management that address the internals of the project.

- **What**: Examples of team management include: agreeing on common objectives, developing a plan (tasks, people, deadlines), progress monitoring, and plan evolution.
- **How**: Agile practices are widely used for managing software projects (Martin, 2003), an approach that iteratively delivers prototypes, an approach that applies to all project scopes. Little is required for a solo project beyond an individual prioritizing their activities. For a lab project, lab managers (e.g., principal investigators) may find it useful to have a spreadsheet that describes who is working on which feature and the expected completion dates. For a community project, there is often coordination across physical locations. Managing the dependencies between teams may demand the use of project management software (Nieto-Rodriguez, 2022) as well as a designated project manager or package maintainer who tracks progress of the project plan.

User engagement (Cagan, 2018) addresses interactions between the software developers and users of the software.

- **What**: Sometimes this is included in project management or product management (delivering products customers want). We separate this aspect because the role often goes unnoticed as a lab project grows into a community project.
- **How**: In a solo or lab project, user engagement typically involves a hallway conversation with a peer researcher. However, in a community project, communicating with users may require using GitHub issues, a designated email account, and even periodic user group meetings.

Table 1 summarizes the foregoing discussion providing examples of software packages that can support the software engineering practices (e.g., linters, and unit test frameworks). The rows are software engineering practices, and columns are the three project scopes: solo, lab, and community. The rigor of engineering practices increases as the scope of the project progresses from solo to community.

We use this table to recommend engineering activities for an academic environment. Our expectation is that most projects are not adequately characterized by a single column, and so we expect that projects may adjust

their practices by employing recommendations from more than one column. We note that the table includes a number of technical terms. Rather than defining these inline, we have included a glossary as Appendix A.

Table 1. Software engineering practices and project scopes.

ENGINEERING PRACTICE	*Increasing software engineering rigor →*		
	Developing for your own use (*solo*)	**Developing for your research group (*lab*)**	**Developing for a broad research community (*community*)**
Version control	• Commit source code to a repository (local or remote) • Use meaningful commit messages to aid code comprehension and debugging • Implement backups, in particular, for local repositories.	• Isolate features being developed into separate commits (e.g., use branches for individual features) • Keep commit history concise (e.g., use squash, merge) • Document the development workflow (e.g., define the expected size of commits etc.).	• Create policies for fixes, new features, and spin-off projects. • Implement review criteria for accepting commit requests. • Create policies for version control mechanisms (e.g., creating forks, branches, pull requests).
Software Design	• Brainstorm with colleagues	• Review user experience via lab presentations of mock ups. • Apply best practices for software design to promote extensibility and reuse (e.g., object oriented). • Adopt code reviews (e.g., multiple developers check coded commits).	• Write functional specifications that describe: the problem addressed, user profiles, use cases. • Document components and their interactions (e.g., class diagrams and interaction diagrams). • Review functional and component specifications.

Coding	• Ensure the repository has a README file. • Use a consistent naming convention for language elements (e.g., objects, functions, classes).	• Use common data definitions (e.g., DNA sequence, light source). • Adopt a software license. • Create practices for coding style. • Use a linter to enforce coding practices (e.g., Pylint, Flake8). • Use build tools (e.g., make). • Use published standards for coding style, especially for tool support (e.g., use PEP 8 and Pylint). • Define templates for packages so that they have a common layout (e.g., directory structure).	• Implement code reviews to ensure standards, style consistency, adequate tests, and use of common code. • Create documentation reviews. • Implement pre-commit/pull request coding style checks in a linter.
Quality Assurance	• Write unit tests for critical components (e.g., using the pytest framework). • Automate local test execution (e.g., using a build-tool target)	• Develop extensive unit tests and measure test coverage (e.g., using coverage.py). • Develop system tests (e.g., end-to-end tests for use cases). • Implement continuous integration for the development platform.	• Implement regression tests for each release. • Where appropriate, implement tests for performance, security, and so on. • Test on all supported platforms. • Evaluate the software package by users of the package.
Packaging & Deployment	• Deploy software through git clone. • Version software via commit labels/tags. • Deploy to established package managers (e.g., pip, conda, conda-forge)	• Use semantic versioning (or tagging) conventions to indicate the magnitude of change from previous versions. • Tag and release versions of the code.	• Create release documentation. • Implement automatic documentation builds. • Use automatic builds for container deployments.

User Documentation	• Document or comment the code. • Document complicated math in a long comment or a reference to a paper. • Provide Jupyter Notebooks to demonstrate the code.	• Implement more extensive developer documentation (e.g., Sphinx). • Create user documentation (e.g., details of usage and options). • Provide citation information for the package (e.g., a DOI).	• Publish documentation (e.g., Read the Docs) that includes examples. • Create email lists for developers and users. • Create tutorials. • Publish on-boarding documentation.
Team Management	• Ad hoc	• Create a plan for each release that specifies delivery date, features, and deliverables for each team member, or at least a prioritization of changes and their estimated level of difficulty.	• Technical leads/contributors create release plans or create project plans with task assignments. • Implement regular “stand ups” to share progress and blockers. • Undertake technology reviews when considering new dependencies. • Define a governance structure (clear rules for decision-making).
User Engagement	• Ad hoc	• Make team presentations on proposed features. • Use issue tracking software (e.g., GitHub Issues).	• Provide a forum whereby users can request as well as give feedback on the prioritization of features and fixes. • Notify users of prioritization of issues. • Provide users with a project roadmap, training at community events, major release changes warnings, and deprecation warnings.

A particular challenge for software engineering for data science is the widespread use of computation notebooks in which software is intermixed with an analysis narrative that provides the user with a way to better understand the results as well as conduct further explorations (e.g., Kery & Myers, 2017; Wang et al., 2019). Probably the most widely used computational notebook is Jupyter (2015; Randles et al., 2017). Others have

noted challenges with notebooks as a computational environment (e.g., Chattopadhyay et al., 2020). Our experience from teaching data science is that the quality of a computation notebook is greatly improved by two practices. The first is documentation. All codes should have documentation that specifies: (i) what the function does, (ii) the types of arguments to the function, and (iii) the types of values returned by the function. A second consideration is to make sure that there is a test for each function, typically within the same cell as the function definition so that the test is run when the function is defined. A final consideration is when to move code from a notebook to a Python module (a file that has extension '.py').

4. Current Practices for Teaching and Training in Software Engineering for Academic Researchers

Key to building a community of researchers capable of contributing to or creating their own open-source packages is providing courses and training opportunities that teach 3R skills for software engineering best practices. Computer science students along with students in related disciplines (e.g., electrical engineering) develop these skills as part of their multiyear degree programs. However, at present, it is very difficult for students outside these disciplines to develop the practical engineering skills required to develop open-source packages.

These difficulties arise from two sources. First, non-CS students have limited time in their schedules to take classes outside their discipline. Our observation is that most non-CS students have time for no more than two courses to acquire the requisite knowledge. Second, existing courses are typically structured so that practical skills are taught only after teaching substantial theory and background. For example, at the University of Washington, the sequence for non-CS students is Computer Programming I (CSE 142), Computer Programming II (CSE 143), Software Design (CSE 331), Data Structures (CSE 332), System and Software Tools (CSE 391), and finally Software Engineering (CSE 403). (We note in passing that the University of Washington Allen School of Computer Science is making extensive changes to this curriculum.)

Other universities have adopted similar approaches. For example, UC Berkeley has the sequence CS61A "The Structure and Interpretation of Computer Programs", CS61B "Data Structures: Fundamental Dynamic Data Structures", and CS 169A "Software Engineering." Although CS61B provides some instruction in version control, this is not part of the curriculum per se. Indeed, there is no course that provides instruction on engineering practice (Armando Fox, Associate Dean, private communication, March 8, 2022). Carnegie Mellon University has addressed the need to provide 3R software engineering skills by creating an undergraduate minor in CS. But a substantial time commitment is required to take the requisite courses. Harvard University's Institute for Applied Computational Science provides a master's degree for students to acquire 3R software engineering skills (IACS, n.d.), but this too is a significant investment of time.

Students do have options outside of formal academic courses to develop 3R skills. Software Carpentry (Becker & Weaver, 2018) and Data Carpentry are successful and scalable ways to introduce introductory-level software

engineering processes to researchers from a broad set of domains. The areas covered in these courses include the UNIX shell environment, version control using git, and an introduction to Python and plotting with Python. More advanced topics such as automating the building of software and the use of databases and SQL are available. Software Carpentry provides much less depth on more advanced software engineering practices such as unit testing and continuous integration. The limited extent of the Carpentry courses (typically a few days) means that it is harder to integrate the processes within the everyday work or research by a student (particularly more advanced practices). Only through repeated use of these practices do they become embedded in the way that we work.

We have two broad thoughts about changes in academic curriculum that are required to address the development of 3R skills. First, we believe that the focus should be on undergraduate courses. One reason is that it provides a scalable mechanism to prepare students for 21st-century careers in academia and research. The other reason is that these undergraduate courses can also be available to graduate and postgraduate students who need 3R skills in software engineering. That is, our focus is on undergraduate courses, but the training will be done at all levels in the university.

Our second thought is that the courses for developing 3R skills need to be radically redesigned. At present, these course sequences are a lightweight version of the material taught to CS undergraduate majors. That is, courses early in the sequence focus on theory; only toward the end of the course sequence do students acquire 3R skills. We recommend that the material be restructured so that 3R skills are taught (and practiced) early on. More advanced courses in the sequence should provide greater sophistication in areas such as programming (e.g., abstraction techniques) and data structures (e.g., complexity analysis). There are a couple of examples of a first course in such a sequence. At the University of Washington, CSE 583 “Software Development for Data Scientists’’ (Beck, 2018) is a one-quarter course on software engineering for non-CS graduate students that covers all of the engineering practices described above and includes a capstone project to practice these skills. At Carnegie Mellon University, Crafting Software (CMU, n.d.) addresses many 3R engineering practices.

We close with more details about the syllabus for CSE 583. The intent of this course is to develop 3R skills for students who have little programming backgrounds. Key topics are: review of Python programming; version control with GitHub; the bash command line; constructing Python modules; unit tests (both what to test and how to use the `unittest` package); creating PyPI packages; continuous integration; and team processes. Team processes include code reviews, technology reviews (how to choose a software dependency), and project planning. After the topics are addressed individually, students gain practice in their use by doing a class project with a team of three to four students.

5. The Future of Software Engineering for Academic Researchers

One major direction we are pursuing at the University of Washington is to develop a community of practice for software engineering, a group of experienced software engineers employed by the university who have focused

engagements with academic software projects. Developing such a community of practice provides resources for: (a) the adoption of resilient software engineering practices in research and (b) teaching software engineering courses on campus. Although funding exists to support software engineering positions, resources in any given research group are often in the form of fractional FTEs (full-time equivalents). It is often difficult to fill fractional FTEs even if we can entice and retain people with the appropriate expertise for these positions. This has motivated us to develop a centralized pool of research software engineers, or **an RSE team**. How to implement such a model is clearly an open question; it is beyond the scope of this article. That said, we highlight a number of existing models and programs in universities.

The eScience Institute at the Universe of Washington is one example of such a model where data scientists work both in service of university programs and in the development of their own research. Such institutes could leverage the size of the university to build software engineering teams as a university-level program to support researchers. In particular, an institute can make use of fractional FTEs of grant support to amortize the cost of skilled software engineers who could not be funded by any one group. Such opportunities have been realized in the United Kingdom with the development of university and national research software engineering programs, and are beginning to be recognized by other universities in the United States. Other examples of nascent programs include: software groups who embed within a project for 6 months to 2 years (e.g., Caltech SASE; California Institute of Technology, 2022), cohorts of master's students with data science and software engineering skills who are matched with research projects for short engagements (e.g., NYU DS3; NYU Center for Data Science, n.d.), embedding professional software engineers in research labs (as Fox reports for UC Berkeley [A. Fox, personal communication, March 8, 2022]), or RSE teams such as the Research Software Engineering Group at Princeton or the National Center for Supercomputing Applications at the University of Illinois.

As noted previously, a core challenge with much of this vision is hiring and retaining skilled software engineers in an academic environment, especially people who are excited about working with researchers who may have minimal training in software engineering. There are multiple avenues for such hires: (a) long-term positions for software engineers already in academia; (b) software engineers in industry who are engaged in open source communities; (c) industry software engineers returning for graduate studies who may find that their software skills provide entry into an exciting research project; (d) early-career engineers interested in research and the impact of their work; and (e) providing opportunities for late-career software engineers looking for a 'second act.' A critical aspect to the success of such a program is the retention of good talent. Carver et al.'s (2022) survey found an overwhelming concern about the lack of career paths for software professionals in academia. This will require careful thought about the career paths for software engineers within the academic environment, an environment that puts a premium on published articles, not software projects. Retaining skilled software engineers will require providing appealing career paths in academic institutions.

We have a few insights as to how to attract experienced software engineers. We have learned much from hiring software engineers for the recently created Scientific Software Engineering Center at eScience. The goal of the center is to apply industry-grade software engineering practices to the development of research software for science. Hence, we mostly targeted industry for sourcing software engineering talent.

We have several observations based on our experience over the last 6 months of hiring. First, it is easier to recruit senior software engineers who have spent a decade or more in industry. They are attracted to the mission, engineering autonomy and scope, and potential to have impact on scientific breakthroughs after spending years on commercial projects that are mainly focused on profit through extremely specific engineering optimizations, often as very small cogs in large engineering-product teams. Second, it is extremely difficult to reach parity with private industry in terms of compensation, which makes it hard to attract junior to mid-level software engineers with industry experience as they are less likely to depart from lucrative careers in the private sector. Third, there is a lack of formal structure for software development in academia. This is both an opportunity for engineers to extend their skills in eliciting software requirements and a challenge as it slows the pace of engineering output due to the high degree of uncertainty when projects are launched. This is sometimes a constraint during recruiting as the uncertainty could be seen as a lack of investment in supplemental roles such as customer success, product design, program/product management, and software ecosystem and servicing—roles that allow software engineers to focus on software coding–related tasks in which they intend to continue growing their skills.

The biggest advantage of software engineering in academia is the culture of openness and the opportunity to change the trajectory of a multiyear investment in science by contributing highly sought after engineering products to a dedicated community of scientists and researchers. This community impact goes beyond the organization or region to benefit society at large. We expect this to be a primary factor in retaining software engineers in academia, and in establishing the perception of research software engineering in academia as a highly fulfilling career path.

6. Conclusions

Our experience at the eScience Institute is that successful data science projects create software that is readable by others, resilient to variations in usage, and reusable by embedding within other software. We refer to these considerations as the 3Rs of software engineering.

This article addresses engineering practices that create 3R software. By engineering practice, we mean much more than coding, although coding is an important element. Among the engineering practices we discuss are: version control, design, quality assurance, packaging, documentation, and project management.

There are robust industry practices for creating 3R software. However, many of these practices are skills-intensive and time-consuming. Further, although application of these practices can result in a high level of 3R capabilities, this outcome is poorly matched with the needs of most academic projects. Most academic projects

are quite small; they consist of a single researcher who is the sole user of the software. A modest number of academic software projects address multiple users in the same lab. Very few academic projects are directed at a large research community. Often the transition from a single-user application to community-developed software arises organically rather than from a decision at the start of a project. These considerations led us to restructure software engineering practices into a progression of increasing rigor to better match the needs of academic projects with different scopes.

The need for 3R skills for academic software led us to examine teaching and training of software engineering. We provide an in-depth analysis of our institution, the University of Washington, and we provide some insights into the situations at Carnegie Mellon University and the University of California at Berkeley. We conclude that undergraduates outside of CS (or related departments, such as electrical engineering) face significant challenges with acquiring 3R skills because of the limited time available in undergraduate majors to take prerequisite courses and the competition to take these courses.

We touch on another path to creating 3R software—building a ‘community of practice.’. This is a team of experienced research software engineers (i.e., an RSE team) who apply engineering best practices to research projects. This is not an alternative to teaching and training, rather it complements those efforts. One example of an RSE team is LINCC Frameworks (n.d.) a joint project between the University of Washington, Carnegie Mellon University and the LSST Corporation to develop scientific software to analyze data from the Rubin Observatory Legacy Survey of Space and Time (LSST). A broader initiative is the recently announced Virtual Institute for Scientific Software (VISS) (Boyle, 2022) that seeks to accelerate scientific discoveries through the development of 3R software for a diverse set of academic projects.

A further consideration is cultural. In academia, the criteria for success is the publication of the results. In contrast, success in a software engineering culture is creating software that is widely used, and has a reputation for good quality. These cultural differences can create an ‘impedance mismatch’ that may present challenges for an RSE team and promoting 3R software.

If we can address these challenges, we have an opportunity to increase the readability, resilience, and reuse of research software in the United States and throughout the world. Doing so will accelerate the progress of research. It will also aid in workforce development by having more undergraduates trained in software development and by providing a community of practice to support the careers and advancement of those software developers in academia.

Acknowledgments

Andrew J. Connolly and Joseph L. Hellerstein contributed equally to writing this article. We thank Vaughn Iverson for suggesting the idea of project scopes. We are indebted to the editor and reviewers for their detailed

comments that greatly improved earlier drafts of this article. We greatly appreciate the thoughtful and thorough comments from the reviewers that were essential to creating a much-improved manuscript.

Disclosure Statement

AJC acknowledges support from the Department of Energy through award DE-SC0011665 and the National Science Foundation through award AST-2107800. This work was supported by the Washington Research Foundation and by a Data Science Environments project award from the Gordon and Betty Moore Foundation (Award #2013-10-29) and the Alfred P. Sloan Foundation (Award #3835) to the University of Washington eScience Institute. The University of Washington Scientific Software Engineering Center (SSEC) and LINCC Frameworks are supported by Schmidt Futures, a philanthropic initiative founded by Eric and Wendy Schmidt, as part of the Virtual Institute for Scientific Software (VISS) and the Virtual Institute of Astrophysics (VIA).

References

Allaire, J. J. (2011). *RStudio: Integrated development environment for R*. https://www.r-project.org/conferences/useR-2011/abstracts/180111-allairejj.pdf

Beck, D. A. (2018). *Software engineering for data scientists*. University of Washington - Paul G. Allen School of Computer Science & Engineering. https://courses.cs.washington.edu/courses/cse583/

Beck, L. L., & Perkins, T. E. (1983). *A survey of software engineering practice: Tools, methods, and results*. IEEE Transactions on Software Engineering, 5(SE-9), 541–561.

Becker, E., & Weaver, B. (2018, May 7). *Meet the members of the software carpentry CAC*. Software Carpentry. https://software-carpentry.org/blog/2018/05/swc-cac.html

Bhatti, J. (2021). *Docs for developers: An engineer's field guide to technical writing*. Apress.

Blokdyk, G. (2022). *Version control software standard requirements*. 5STARCooks.

Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, *25*(12), 1226–1241.

Boyle, A. (2022, January 22). *University of Washington joins $40M campaign to enlist more software engineers for research*. GeekWire. https://www.geekwire.com/2022/university-of-washington-joins-40m-campaign-to-enlist-more-software-engineers-for-research/

Cagan, M. (2018). *Inspired: How to create tech products customers love*. Wiley.

California Institute of Technology. (2022). *SASE: The Schmidt Academy for Software Engineering*. https://sase.caltech.edu/

Carnegie Mellon University. (n.d.). *Crafting Software*. https://cmu-crafting-software.github.io//assignments/hw1

Carver, J. C., Weber, N., Ram, K., Gesing, S., & Katz, D. S. (2022). A survey of the state of the practice for research software in the United States. *PeerJ Computer Science*, *8*, Article e963. https://doi.org/10.7717/peerj-cs.963

Chattopadhyay, S., Prasad, I., Henley, A. Z., Sarma, A., & Barik, T. (2020). What's wrong with computational notebooks? Pain points, needs, and design opportunities. In R. Bernhaupt, F. Mueller, D. Verweij, & J. Andres (Eds.), *CHI '20: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM.

Choi, K., Medley, J. K., König, M., Stocking, K., Smith, L., Gu, S., & Sauro, H. M. (2018). Tellurium: An extensible Python-based modeling environment for systems and synthetic biology. *Biosystems*, *171*, 74–79.

Cotton, B. (2016). *Making documentation easy to read with Read the Docs*. opensource.com.

Currie, M. J., Berry, D. S., Jenness, T., Gibb, A. G., Bell, G. S., & Draper, P. W. (2014). Starlink Software in 2013. *Astronomical data analysis software and systems XXIII*. ASP conference series, vol. 485. https://aspbooks.org/publications/485/391.pdf

Deil, C., Zanin, R., Lefaucheur, J., Boisson, C., Khelifi, B., Terrier, R., Wood M., Mohrmann, L., Chakraborty, N., Watson, J., Lopez-Coto, R., Klepser, S., Cerruti, M., Lenain, J. P., Acero, F., Djannati-Ataï, A., Pita, S., Bosnjak, Z., Trichard, C., Vuillaume, T., ... Arribas, M. (2017) Gammapy - A prototype for the CTA science tools. 35th International Cosmic Ray Conference (ICRC2017), vol. 301, 766. https://doi.org/10.22323/1.301.0766

Feller, J., Fitzgerald, B., Hissam, S. A., & huff, K. R. (2007). Adopting open source software engineering (OSSE) practices by adopting OSSE tools. In *Perspectives on free and open source software* (pp. 245–264). MIT Press.

Fowler, M. (2003*). UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley.

Fox, A., & Patterson, D. (2012). Crossing the software education chasm. *Communications of the ACM*, *55*(5), 44–49.

Nieto-Rodriguez, A. (2022). Project Management Handbook: How to Launch, Lead, and Sponsor Successful Projects. Harvard Business Review.

Glass, R. L., Vessey, I., & Ramesh, V. (2002). Research in software engineering: An analysis of the literature. *Information and Software Technology*, *44*(8), 491–506.

Hettrick S. (2014, December 4). *It's impossible to conduct research without software, say 7 out of10 UK researchers.* Software Sustainability Institute. https://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers

GeeksforGeeks. (2022, July 3). Introduction to semantic versioning. https://www.geeksforgeeks.org/introduction-semantic-versioning/

Hooley, F., Freeman, P. J., & Davies, A. C. (2021). Ten simple rules for teaching applied programming in an authentic and immersive online environment. *PLoS Computational Biology*, *8*(17), 1–11.

Institute for Applied Computational Science. (n.d.). *Degree programs*. https://iacs.seas.harvard.edu/graduate-programs

Janssens, J. (2014). *Data science at the command line: Facing the future with time-tested tools*. O'Reilly Media.

Jaspan, C., Chen, I.-C., & Sharma, A. (2007). Understanding the value of program analysis tools. In R. P. Gabriel (Ed.), *OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (pp. 963–970). ACM.

Jordan, T. B., Johnson, B., Witschey, J., & Murphy-Hill, E. (2014). Designing interventions to persuade software developers to adopt security tools. In R. Biddle, B. Chu, E. Murphy-Hill, & H. Lipford (Eds.), *SIW '14: Proceedings of the 2014 ACM Workshop on Security Information Workers* (pp. 35–38). ACM.

Jupyter. (2015, November 30). *Project Jupyter*. Berkeley Institute for Data Science. https://bids.berkeley.edu/research/project-jupyter

Keeling, M. (2017). *Design it!* Pragmatic Bookself.

Kerninghan, B. W., & Pike, R. (1999). *The practice of programming*. Addison-Wesley.

Kery, M., & Myers, B. A. (2017). Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 25–29). IEEE.

Knuth, D. E. (1992). *Literate programming*. Stanford University Center for the Study of Language and Information.

Laurent, A. M. (2004). *Understanding open source and free software licensing*. O'Reilly Media.

Li, Q., & Chen, Y. L. (2009). *Data flow diagram*. In *Modeling and analysis of enterprise and information systems* (pp. 85–97). Springer.

LINCC Frameworks. (n.d.) Home page. https://www.lsstcorporation.org/lincc/frameworks

Linux Man-Pages Project. (2009, March 16). *Home page*. Kernel. https://www.kernel.org/doc/man-pages.

Loeliger, J., & McCullough, M. (2012). *Version control with git: Powerful tools and techniques for collaborative software development* (2nd ed.). O'Reilly Media.

Martin, R. C. (2003). *Agile software development: Principles, patterns, and practices*. Prentice Hall.

Milewicz, R., Pinto, G., & Rodeghero, P. (2019). Characterizing the roles of contributors in open-source scientific software projects. In M.-A. Storey (Ed.), *MSR '19: Proceedings of the 16th International Conference on Mining Software Repositories* (pp. 421-432). IEEE Press. https://doi.org/10.1109/MSR.2019.00069

Nielsen, J. (2000, March 18). *Why you only need to test with 5 users*. NN/g Nielsen Norman Group. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

Nguyen, Q. (2019). *Hands-on application development with PyCharm: Accelerate your Python applications using practical coding techniques in PyCharm*. Packt.

NYU Center for Data Science. (2022). *Data Science and Software Services*. https://cds.nyu.edu/ds3/

Osimo, D., & Switters, J. (2019). *Recognising the importance of software in research Research Software Engineers (RSEs), a UK example*. Publications Office of the EU.

Patton, R. (2005). *Software testing*. SAMS Publishing.

Ponuthorai, P., & Loeliger, J. (2023). *Version control with git: Powerful tools and techniques for collaborative software development* (3rd ed.). O'Reilly Media.

Poppendieck, M., & Poppendieck, T. (2003). *Lean software development: An agile toolkit*. Addison-Wesley Longman.

Postan, L. (2021, January 11). *Best software engineer blogs for 2021*. DEV. https://dev.to/rliraz/best-software-engineer-blogs-for-2020-jhi

Price-Whelan, A. M., Sipőcz, B. M. Günther, H. M. Lim, P. L. Crawford, S. M. Conseil, S. Shupe, D. L. Craig, M. W. Dencheva, N. Ginsburg, A. VanderPlas, J. T. Bradley, L. D. Pérez-Suárez, D. de Val-Borro, M. Aldcroft, T. L. Cruz, K. L. Robitaille, T. P. Tollerud, E. J. Ardelean, C. ... Dietrich, J. P. (2018). The Astropy Project: Building an open-science project and status of the v2. 0 core package. *The Astronomical Journal*, *156*(3), Article 123. https://doi.org/10.3847/1538-3881/aabc4f

Project Management Institute. (2017). *A guide to the project management body of knowledge.*

Ragan-Kelley, M., Perez, F., Granger, B., Kluyver, T., Ivanov, P., Frederic, J., & Bussonnier, M. (2014). The Jupyter/IPython architecture: A unified view of computational research, from interactive exploration to

communication and publication. *AGU Fall Meeting Abstracts, 2014*, Article H44D-07.

Randles, B. M., Pasquetto, I. V., Golshan, M. S., & Borgman, C. L. (2017). Using the Jupyter Notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, 1–2. IEEE. https://doi.org/10.1109/JCDL.2017.7991618

Sauro, H. M. (2018, August 28). *Department web page*. University of Washington. https://bioe.uw.edu/portfolio-items/sauro/

Sauro, H. M., Hucka, M., Finney, A., Wellock, C., Bolouri, H., Doyle, J., & Kitano, H. (2003). Next generation simulation tools: The Systems Biology Workbench and BioSPICE integration. *OMICS*, *7*(4), 355–372.

Schmidt Futures. (n.d.). *Our mission*. Retrieved April 27, 2023, fromhttps://www.schmidtfutures.com/

Smith, L.P., Bergmann, F. T., Chandran, D., & Sauro, H. M. (2009). Antimony: A modular model definition language. *Bioinformatics*, *25*(18), 2452–2454.

Somogyi, E. T., Bouteiller, J. M., Glazier, J. A., König, M., Medley, J. K., Swat, M. H., & Sauro, H. M. (2015). libRoadRunner: A high performance SBML simulation and analysis library. *Bioinformatics*, *31*(20), 3315 – 3321. https://doi.org/10.1093/bioinformatics/btv363

Stack overflow. (2022). https://tinyurl.com/4zf737j5

Timperley, C. S., Herckis, L., Le Goues, C., & Hilton, M. (2020). Understanding and improving artifact sharing in software engineering research. *Empirical Software Engineering*, *26*, Article 67. https://doi.org/10.1007/s10664-021-09973-5

Tody, D. (1986, October 12). The IRAF data reduction and analysis system. In D. L. Crawford (Ed.), *Proceedings SPIE 0627, Instrumentation in Astronomy VI* (p. 733). SPIE.

University of Washington. (2022-a). *CHEME 545-546: Data science methods for clean energy research and software engineering for molecular data scientists*. Retrieved April 27, 2022, from https://chem.washington.edu/courses/2021/autumn/chem/541/a

University of Washington. (2022-b). *CSE 583: Software development for data scientists*. Retrieved April 27, 2022, from http://courses.cs.washington.edu/courses/cse583

University of Washington. (2022-c). *DATA 515A: Software design for data science*. Retrieved April 27, 2022, from https://courses.cs.washington.edu/courses/csep515/

Waldon, B. (2012). *A beginners guide to software deployment*. CreateSpace Independent Publishing Platform.

Wang, A. Y., Mittal, A., Brooks, C., & Oney, S. (2019). How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction*, *3*(CSCW), Article 39.

Wikipedia. (n.d.-a) *Project management software*. Retrieved April 27, 2022, from https://en.wikipedia.org/wiki/Project_management_software

Wikipedia. (n.d.-b) *Khan academia*. Retrieved April 27, 2022, from https://en.wikipedia.org/wiki/Khan_Academy

WIkipedia. (n.d.-c) *EdX*. Retrieved April 27, 2022, from https://en.wikipedia.org/wiki/EdX.

Wikipedia. (n.d.-d) *Coursera*. Retrieved April 27, 2022, from https://en.wikipedia.org/wiki/Coursera.

Wikipedia. (n.d.-e). *GitHub*. Retrieved April 27, 2022, from https://en.wikipedia.org/wiki/GitHub

United States Research Software Engineering Organization. (2023). *Home page*. Retrieved April 27, 2022, from https://us-rse.org/

Appendices

Appendix A. Glossary of Terms

- **artifact**: a work product of the software engineering process including code, documentation, data, design documents, and plan documents.
- **branch**: changes (commits) to code to implement or modify features within a software repository, where the changes are kept separate from other changes to the repository.
- **class diagram**: a visual description of the relationships between elements of a software design.
- **clone**: a copy of a repository.
- **code review**: a process by which software engineers share their work and receive feedback on coding style, design, and implementation of the software.
- **coding style**: the manner in which codes are written to improve readability by others, such as PEP 8 for Python. This includes naming conventions for methods and variables.
- **commit**: a change to a repository branch.
- **component design**: the specification of the software elements in a system by their inputs, outputs, and behaviors (e.g., how inputs are transformed into outputs).
- **Conda**: Package dependency and environment deployment. https://docs.conda.io/en/latest/
- **CRAN**: Comprehensive R archive network. https://cran.r-project.org/
- **dependency**: software that is required for installing a package. For example, it is common for Data Science packages to have dependencies on `numpy`, `pandas`, and `matplotlib`.

- **deployment**: the process of making a software package available to others. This can be done by download (e.g., PyPI), containers (e.g., Docker), or server-based hosting (e.g., on Amazon Web Services).
- **engineering practice**: a generally accepted method for some aspect of building software.
- **SourceForge**: Software repository and distribution. https://sourceforge.net/
- **fork**: creation of a new code repository based on an existing code repository.
- **GitHub**: an online system for version control and continuous integration. http://github.com
- **GitHub issues**: a feature of GitHub that provides for reporting, prioritizing, and tracking issues.
- **linter**: a tool that detects errors in code without executing the code (e.g., detecting that a variable may be referenced before it is assigned).
- **merge**: combining changes committed to two branches of a code repository.
- **module**: (1) reusable software with a well-defined set of features and ways to use it; (2) a Python file (i.e., as the file extension "py").
- **notebook (or software notebook)**: a programming environment in which code is mixed with a narrative text to create an easily updatable report.
- **package**: a unit of software distribution and reuse.
- **pull request**: a set of proposed changes to a branch of a software repository.
- **project management**: the process of setting goals, assigning tasks, and managing the progress of a project.
- **PyPI**: Python package installer. https://pypi.org/
- **regression test:** tests that detect that a previously implemented feature no longer works or works incorrectly.
- **repository**: a collection of files that is managed so as to capture incremental changes or 'commits.'
- **semantic versioning:** a way to specify software versions that indicate the magnitude of change from previous versions (GeeksforGeeks, 2022).
- **software design**: the process by which software is created including specification of the user interactions (user experience) and the specification of software components such as modules and functions.
- **squash**: combine multiple commits (changes) into a single commit.
- **technology review**: a process whereby a software development team selects a software package to use in their project.
- **test coverage**: the fraction of statements in a module or collection of modules that are executed when all of the unit tests are run.
- **unit test**: code written to detect errors in a software project.
- **use case**: a way in which the user employs the software system to perform a task of interest to the user.
- **user experience**: how the user interacts with the software system such as by a web browser or a command line.
- **version control**: tools and best practices for tracking changes to code, data, and documents.

Appendix B: Template Functional Specification

A functional specification describes how developers of the software system envision that the user will interact with their software. Over the past 10 years, we have taught several thousand students how to write these specifications. We provide our students with the following outline:

- Background. Describe the problem being addressed and why it is important.
- User profile. Detail who will use the system. What do they know about the problem domain? What is their computer literacy (e.g., can browse the web, can program in Python)?
- Data sources. What data you will use and how it is structured?
- Use cases. For each use case, describe: (a) the objective of the user interaction (e.g., withdraw money from an ATM); and (b) the expected interactions between the user and your system.

Appendix C: Template Component Specification

A component specification describes the components of the software and how they interact to accomplish use cases in the functional specification. By component, we generally mean higher level capabilities such as data retrieval, significant analysis capabilities, or any novel capability. Over the past 10 years, we have taught several thousand students how to write these specifications. We provide our students with the following outline:

- Software components. High level description of the software components such as: *data manager*, which provides a simplified interface to your data and provides application specific features (e.g., querying data subsets); and *visualization manager*, which displays data frames as a plot. Describe at least three components specifying: what it does, inputs it requires, and outputs it provides.
- Interactions to accomplish use cases. Describe how the above software components interact to accomplish at least one of your use cases.
- Preliminary plan. A list of tasks in priority order.
