Increasing the quality of (academic) scientific software

Jan-Patrick Lehr[a,*], Tomislav Maric[b], Dieter Bothe[b], Christian Bischof[a]

[a]*Scientific Computing, Computer Science Department, Technische Universität Darmstadt, Darmstadt, Germany*
[b]*Mathematical Modeling and Analysis, Mathematics Department, Technische Universität Darmstadt, Darmstadt, Germany*

1. Introduction

In computer science and other computational sciences, reliable and maintainable software is key. The urgent need for reproducibility of results puts another strong emphasis on maintainable and traceable software versions, and their development processes. Methodologies supporting developers to ensure these properties are available and benefit industrial software development, e.g., [1]. However, in our experience such technology is only adopted slowly in academia – often giving the reason that it is not serving the purpose of science, i.e., writing papers, or they are considered as given, underestimating the time actually required to implement the required mechanisms.

While we understand both positions, we highly encourage using tools, such as continuous integration (CI), to develop better software (in academia). Given well-tested software, it can, e.g., be extended without the fear to break functionality. In our opinion this ultimately leads to more productivity of the people involved, i.e., the developers and scientists, while ensuring reproducibility and traceability of results. This perception is, e.g., supported by[2].

In this paper, we highlight three key approaches that, for us, increased the software quality, usability and ultimately, developer and user productivity. (1) **build from a user perspective**: design and implement the software in a way that you would be happy with it as a **user**, e.g., using so-called Test Driven Development (TDD). (2) **use continuous integration and testing**: setup (at least basic) CI capabilities right from the start of a new project and expand with increasing project time. Integrate CI into existing projects gradually but steadily. (3) **containerize early and everything**: setup containerization early in the project to help both users and CI with reproducible environments, and at least create images of the software environment associated to accepted publications, to simplify the reproduction of results.

2. Build from a user perspective

In academia many software projects are built from a singular research point of view, as compared to, e.g., an overarching mission. In our experience this means: many (sometimes very) specific (library) dependencies; non-intuitive or broken build processes; clumsy to use non-intuitive Application Programming Interfaces (API); difficult to extend tools and libraries because of a non-modular code design. Very often, new versions of the code are created to investigate alternative problems, or introduce new methods, that are never merged with the original version. This makes sharing of knowledge between scientific projects practically impossible, as the codes continue to diverge over time. Because of these issues, researchers often rather start from scratch, than face the nightmare of scientific legacy code. This is sub-optimal, as it limits the usefulness of somebody's work, as (1) only few other researchers benefit from the achieved software, (2) the code is rendered unusable as soon as the project ends or the researcher leaves As a consequence, one cannot build on the existing research work, which increases the amount of work per individual, and severely limits progress in research, as people reimplement existing approaches and technology.

If software is built from the user perspective, artifacts, such as correct documentation or usable containers, are essential outcomes. Thus, extending existing work, or applying it to new fields is much eas-

*Corresponding author
Email addresses: jan-patrick.lehr@tu-darmstadt.de (Jan-Patrick Lehr), maric@mma.tu-darmstadt.de (Tomislav Maric), bothe@mma-tu-darmstadt.de (Dieter Bothe), christian.bischof@tu-darmstadt.de (Christian Bischof)

ier and brings benefit to the individual researchers who spend less time dealing with library incompatibilities and half-broken APIs. The user perspective can be demanded or enforced at various levels, e.g., at the institution, or by an individual researcher. Nevertheless, we prefer to demand artifacts for publications, whenever possible, and enjoy seeing more and more artifact initiatives at conferences.

In our view, tests are the cornerstones of scientific codes and a requirement for any scientific publication. An interesting approach to building software from a user perspective is the so-called **Test-driven Development** (TDD). Teaching researchers this simple practice might improve the quality of scientific codes from the user perspective.

Developing from a user perspective from the start requires some time and effort initially. However, we are convinced that the return on investment is positive for any software project, as many of them tend to extend to use cases not anticipated in the first place, and the time required to remove technical debt typically is significant.

3. Use continuous integration and testing

A technology that is of increasing interest and adopted more often in our institution is continuous integration as it performs remote automated testing of the code in a central repository. This ensures that the (automatic) tests not only "run on my machine", but also that the integrated modifications proposed by others in the team either improve the existing code, or at least not break anything. Moreover, the infrastructure ensures that all tests are run, reducing the risk of fully broken builds. In our software projects, we started employing CI over the past couple of years to increase software reliability.

It becomes especially relevant to introduce CI in development of scientific codes that implement methods that cannot be made entirely mathematically rigorous. Such methods are usually based on numerical approximations whose stability cannot be analyzed and therefore require very rigorous testing. Verification tests serve the purpose of ensuring that modifications in the numerical approximation do not break the convergence and stability elsewhere. Even methods that allow more mathematical rigorous treatment may contain serious bugs in their implementation and require automatic verification and validation.

3.1. Continuous Integration with GitLab

We use GitLab CI and separate the tests into two categories: (1) tests (potentially) executed on a test server, and, (2) tests executed on a high-performance computing (HPC) system. The reason behind this separation is the disparity in execution time, i.e., ranging from a couple of hours to up to 24 hours. We separate the CI pipeline further in build, unit, smoke and production tests.

The build stage builds the software and potential dependencies in a well-defined way. Unit tests target a specific single interface, e.g., a class. Smoke tests integrate multiple components and test larger parts of functionality, but with heavily reduced inputs to limit execution time. Finally, production tests run actual experiments to validate the correctness of the methods in real scenarios. Depending on their computational demand, these stages and tests are performed on different hardware resources.

3.2. Use CI with Numerical Software

In numerical software, obviously, the major attention is on the numerical properties. Thus, numerical unit tests can be fully verified with small input data, and they require very short computation times, which leads to an immediate feedback to the developer. Smoke tests generate exactly the same numerical output as performance tests, but with a strongly reduced input size. They are good indicators if something has gone seriously wrong when introducing a modification, however, the lack of resolution in those tests makes them inapplicable for errors that can only appear in production. Production tests are used for the overall numerical method with input data that have realistic sizes. They are basically numerical experiments, that must be reproduced, up to some accuracy, in order to ensure that modifications are either improving the overall method, or at least not making it worse.

3.3. Add CI in Existing Project

The PIRA project started as a small research tool without any of, e.g., contribution guidelines, or automated testing, and was split into five different repositories for historic reasons. The increasing complexity and people using it, highlighted the need for more rigorous testing and a contribution workflow. The introduction of a defined workflow, supported by CI, had an immediate impact on quality and reliability as it emphasized the need for (more) (1) setup automation, (2) robust testing, and, (3) easier configuration. We followed the PSIP Progress Tracking Cards[1] to investigate the status of the project and identify first steps. The introduction of CI in all PIRA projects revealed some minor bugs in the different tools, and in the actual PIRA integration. The Gitlab issue tracker helped with reporting and addressing them.

The initial CI integration was done over the course of a few months, counting from the initial start with CI at all, up to having the procedures in place and the CI pipeline fully running.

3.4. Add CI from the Start

Codefilter is a project that started recently and in which we, opposed to PIRA, set up both contribution guidelines and CI from the very start. This, so far, kept the time investment required into CI reasonable, as we profit from (1) our previous experiences from the PIRA project, and, (2) making incremental changes rather than huge one-time efforts. Specifically students who contribute to this project can follow the clear workflow. In our opinion, this lead to a much cleaner repository, history, and working software, hence, people had to deal less with clumsy setups and broken software states.

3.5. Challenges experienced

One obstacle was the lack of experience or training in CI and some required technologies. This is especially true for domain scientists who do not receive training in, e.g., software testing or engineering. Another challenge was the lack of certain technical infrastructure, e.g., Docker executors, which resulted in more complex CI solutions. A challenge in the PIRA project was the unnecessarily complex project setup with numerous repositories, which contributed to reliability issues.

Strict continuous integration, i.e., allow a contribution only if all tests have passed, is possible with the short running build, unit and smoke tests. It is, however, still not clear to us how to integrate production tests most beneficial into the CI workflow. Continuing to work on the code while waiting for production tests could mean a loss of a workday for the researcher, if the production tests fail.

[1]`https://github.com/bssw-psip/ptc-catalog`, last accessed 2020/06/29

3.6. Opportunities identified

For a developer/scientists the introduction of CI means higher productivity as people can focus on the research task more quickly. Using CI implements an environment in which the software is known to work. Consequently, the CI configuration (1) ensures the software builds in a defined environment, and, (2) can serve as a reference for getting started with the project. The latter being valuable for, e.g., students, joining the project.

At an institutional level we see the consolidation of efforts by individual research groups into competence that is accessible across groups as a great opportunity. From our experience, this can greatly reduce the initial invest necessary and open the technology to rather domain focused groups as well. Thus, providing consulting and support for CI solutions and teaching can help adopt the technology, as we found the incremental changes throughout the project much easier than the initial setup.

4. Containerization

Another technology that has gained traction – in our case mainly for reproducibility – is containerization. In addition, the technology can help with automated testing in CI and to make prototype software more widely accessible, as a potential user does not need to set up the environment. Thus, using published research for a new use-case and in other scenarios is easier to achieve.

In case of numerical software, we started using containers by creating a Singularity [3] image for each publication of the numerical software that is developed at our institute. The image contains both the software environment and the input data to generate the results.

In case of other software projects, e.g. PIRA, we are now slowly adopting containers. We currently evaluate Docker as the technology to create portable development environments and both Docker and Singularity as a means of easing the access to the software.

4.1. Challenges and opportunities

The challenges are similar to the ones outlined for CI: (1) time for initial adoption, i.e., to familiarize one self with the technology, and, (2) creating reasonable workflows that reduce the time to build and prepare the images. The second part is a crucial step, as it requires a decent understanding of the technology to create achievable and lightweight workflows that both research staff and students can follow. In addition, the portability of such containers may be affected by the targeted HPC system, as different systems use different schedulers, thus, require different submission scripts, etc.

Containerization offers many opportunities for researchers that develop software, as it can heavily ease the process to set up an environment, and ship prototype software. In addition, setting up well-defined software environments increases productivity as it removes many pain points in the development processes of research software. This can be used, not only for research, but also in practical teaching labs, in which students are asked to develop software that should be submitted for grading at the end of the semester.

5. Conclusion

From our experiences we draw several conclusions: First, software should always be thought from a user perspective, as also a developer is a user (at least of the internal APIs). This helps to increase the quality and the user experience, of which both help to enjoy, and interact with, the software. Second, continuous integration helps significantly to ensure software quality and helps to catch several errors early. As a result, users and developers find themselves less frequently in a "That worked for me" situation, which helps with introducing new people to the project and to enjoy, and interact with, the software. Third, containers help with productivity through a software environment that is easier to distribute, reducing the time required to install and configure dependencies. The aspect of less frustrating interaction with a software is independent of whether it is used by a developer who relies on a library API or a front-end user doing domain science with, e.g., a simulation software. This is important. People have to deal less with obscure or unexpected behavior of a software package, hence, they can work productively on the actual task, i.e., their science.

Implementing the aforementioned techniques requires time and commitment. In particular, the initial time invest may be significant and adds to the *assumption* that it slows down (research) progress. In our observation, this is not true, because of the mid- and long-time benefits achievable. We see particular opportunities for both centralized and coordinated and collaborative efforts at universities to provide experience and knowledge as a consulting service to establish such methodologies across research groups. Moreover, we think that best practices in (scientific) software development and research data management should also be taught to students during their curriculum.

References

[1] A. Miller. A hundred days of continuous integration. In *Agile 2008 Conference*, pages 289–293, 2008.

[2] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 805–816, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.