Improving software quality in bioinformatics groups through teamwork

This manuscript (permalink) was automatically generated from ferenckata/SQSeminarPaper@7bb5c9f on November 27, 2023.

Authors

- **Katalin Ferenc** ✉
  0000-0002-3006-4297 · ferenckata
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway · Funded by Grant XXXXXXXX

- **Ieva Rauluseviciute**
  0000-0001-9253-8825 · ievarau
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway

- **Ladislav Hovan**
  0000-0001-8847-9295 · ladislav-hovan
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway

- **Vipin Kumar**
  · princeps091-binf
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway

- **Anthony Mathelier** ✉
  0000-0001-5127-5459
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway; Department of Medical Genetics, Institute of Clinical Medicine, University of Oslo and Oslo University Hospital, Oslo, Norway

✉ — Correspondence possible via GitHub Issues or email to Katalin Ferenc <k.t.ferenc@ncmm.uio.no>, Anthony Mathelier <anthony.mathelier@ncmm.uio.no>.

Abstract

Introduction

Bioinformatics and computational biology are continuously gaining importance in biological research. These fields are heavily relying on inventions of computer science and software engineering. About 90% of researchers rely on results produced by scientific software [1]. One implication of poor software engineering practices is that incorrect software results in invalid scientific findings [1,2]. Beyond that, even when the software performs as intended, researchers spend significant amount of time on software building using suboptimal practices which can further increase the necessary time investment in the future [3]. Saving time in early development at the expense of future maintenance is known (in the software engineering literature) as accumulation of technical debt [4,5].

Good software development practices mitigating the risk of incorrect software solutions has been established in other software-heavy endeavours. However, bioinformaticians or more generally scientists working with scientific software often lack formal education in computer science or software development [2,6,7]. Such lack of theoretical and practical foundations hinders the adoption of good coding practices (e.g. software architecture planning, continuous integration, unit tests, code reviews). Indeed, the software quality attributes defined by the International Organization for Standardization may be summarized into the “unknown unknowns” for bioinformaticians [8]. For example, reliability of the software should ensure that the expected output is always produced by the software, which maintainability should allow for easy inspection, update and testing of the software.

Beyond the limits of individual education, many of these practices rely on redundancy of knowledge within team members, supported by practices such as pair programming, daily stand-up meetings and code reviews. In contrast, historically academic research projects are mostly driven by a single person (a PhD student or a post-doc) as these projects are often part of academic degree evaluation. Sadly, academic world still often tends to disregard the team effort, priding itself on enabling individual achievements and career progression instead. Taking this into account, with one (or very few) person developing the software for a project, scientific software remains poorly maintained even if it is used by a significant number of researchers worldwide [7,9,10,11]. The concept of a team is therefore different in a research-oriented project compared to a software development project. While the group members help each other with scientific suggestions, most often there is a single responsible person for the design and implementation of the code base. As official guidelines on coding practices are rarely definitive, but rather suggestive, the actual craft of software engineering is treated as secondary task and is often up to individual judgment. These guidelines may naturally emerge in larger groups, if software is used or even developed by multiple group members. It is not always obvious that to follow them requires a form of team organization not intrinsic to academic groups. To highlight the team aspect, it was reported previously that researchers tended to rank software engineering concepts higher if they worked in a team [6]. High-profile code bases often feature larger development teams and their activities are marked with longer commit messaged indicating better communication and documentation of the software [10]. We hypothesize that the form of team structure organized around individual software products could improve the quality of our scientific code.

The next, almost natural, question related to better maintenance of scientific software in academia is funding. It is notoriously hard to attract funding for existing scientific software, which contributes to the poor maintenance status of much of scientific software [1]. Fortunately in recent years, the lack of funding is being recognized and addressed by a few agencies, such as the Chan Zuckerberg Initiative Essential Open Source Software for Science fund [12]. These initiatives are great examples of how our views towards scientific software should be updated. Scientific community and funding agencies should welcome the efforts of maintaining original software and encourage its updates instead of development of new software that risks remaining unmaintained. Development and maintenance of scientific software can be improved by taking into account good software practices. This can be implemented in research groups by creating a teamwork atmosphere where trainees and staff are directly or indirectly working on the same software.

In this work we first analyse how software engineers are evaluating the computational scientists (emphasizing, but not limiting to bioinformatics) and what suggestions they give to improve the overall quality of software. In addition, we review the literature on guidelines posed by bioinformaticians to the bioinformatics community to summarize the current software standards and priorities circulating in the community. To better understand the team work aspect, we present the main aspect of team management. Finally, we present and discuss how our research groups, motivated and inspired by current software quality state, created weekly sessions to practically discuss and learn to incorporate different aspects of software quality. Good practices require investment in time and effort that may not feasible to fulfill as an individual (e.g. having limited time to perform and to deliver doctoral thesis). However, we suggest that sharing the existing software knowledge base in the group, learning new tool development and implementation together, and, more importantly, normalizing showing and discussing code results in an overall better code quality of the members with a reduced effort on an individual level. We aim to provide a motivation and framework on how to get started with a setting, where code can be developed in a team setting directly or indirectly involving all group members with or without formal training in software engineering.

Guidelines and practices of bioinformatics as seen by software engineers and bioinformaticians

The internet is full of learning and support material for developing or working with software products. Since bioinformaticians are often teaching themselves coding and only a small fraction have formal training in software quality practices, these recourses become vital. They include blog posts from peers, freely available lecture materials from universities, forums or articles that propose guidelines how to code or analyse data in a better way. The practices that are encouraged vary a lot, therefore the code produced by bioinformaticians may lack a certain standard. This has been analysed by the software engineer community, where certain limitations and caveats were identified and discussed. We provide an overview of the available literature on these discussion points.

First, let us summarize the main themes in papers with guidelines for bioinformaticians. We used several key phrases to search for papers: “guidelines for bioinformatics software”, “rules for biologists learning bioinformatics”. Selected papers focus on specific suggestions, often referred to as rules or “tips & tricks”, or they more broadly direct the readers towards good practices of coding, which are put together into guidelines. Specialized topics include particular data analysis in a single disease [13], while broad themes span from next-generation sequencing (NGS) data analysis to outlining tips on how to start on computational analysis of the experimental data [14,15,16]. Both types of papers emphasize the need to learn how to analyse data properly and provide good suggestions to do that, based on the chosen topic. The guideline papers tend to target early career researchers with minimal coding experience. Therefore, the guidelines might be slightly basic, especially from the perspective of software engineers. For example, documentation and version control are most commonly highlighted [17,18]. However, instructions or tips on how to make your documentation and code actually good and up to software quality standards are usually limited due to the short nature of guidelines.

In Table 1, we collected recommendations to improve the software quality of scientific, and whenever possible bioinformatics, software. The literature search was performed in multiple iterations using Google (to include grey literature) and Google Scholar based on phrases “scientific software development”, “software engineering bioinformatics” and “bioinformatics software recommendations” throughout 2023. Additionally, relevant articles were selected based on the snowball effect from the references of the initial publications. Software engineers notice the separation between software engineering and scientific computing community. Already almost two decades ago Diane F. Kelly wrote that scientific computations keep on being performed using error-prone development practices and reaching suboptimal solutions and poor software quality due to lack of appropriate software engineering practices [19]. Software engineering community also writes guidelines on how these practices should be followed after surveying the current state of software in scientific community (focusing on bioinformatics) [1,3,6,11,20]. In addition, an extensive literature review has been published recently in which known issues and suggested solutions are collected [2]. Among recommended practices there are agile coding, the DRY (don’t repeat yourself) principle, requirements gathering and unit testing, none of which concept is intuitive, well known in the bioinformatics community or even trivial to adapt to bioinformatics software [22]. On the other hand, concepts like object-oriented programming and extreme programming are known and used in the bioinformatics community, but they are not used to their full power [20]. For example, bioinformaticians might use classes and inheritance, but rarely encapsulation and polymorphism [20].

Table 1: Collection of recommendations for improving scientific software quality. Some guidelines are more vague than others, they also have varied scope, and they target different stakeholders. Therefore, it may be hard to find individual responsibility and actionable points from the literature.

Recommendation	**Source**
version control	[2,3,7,23]
user (and developer) documentation	[2,7,20,23]
standardised tests	[2,3,7,20,23]
independent review of source code	[20,23,24]
recognition and assignment of adequate time for quality-assured development	[20,23]
recognition of software development as academic achievement	[23,24]
standardized working environment and automation	[2,23]
financial support for software development and maintenance	[23,24]
support for developer community for long term maintenance (when applicable)	[23,24]
licensing	[7,23]
requirements gathering	[2,20]
containerization for portability	[2,7]
reuse existing (reliable) software	[3,7]
agile software development methodology	[2,3]
educated choice of software development methodology	[20]
adoption of international best practice standards of software quality	[23]
establish validation and acceptance procedures	[23]
cooperation between developers and users	[23]
description of the software version used, its configurations and parameters in publications	[23]
preferentially selecting freely available open-source software	[23]

Recommendation	Source
encourage user participation in the software development process	[23]
tagging of software version for reproducibility	[7]
sanity check on input parameters	[7]
do not hard-code changeable parameters and paths	[7]
rely on package managers	[7]
do not require superuser privileges	[7]
provide a small test set	[7]
ensure reproducibility of results	[7]
refactoring	[3]
usage of design patterns	[3]
quality monitoring (e.g. SonarQube)	[3]
continuous integration	[3]
contribute to open-source development	[24]

The guidelines from engineers are struggling to penetrate the bioinformatics community and the challenges in software development cannot be an excuse for skipping good practices [2]. As reported, 47 percent of scientists had a good understanding of testing, while just 34 percent thought any formal training was important [1,6]. To put it into perspective, a scientist would likely not trust results of the microscope or an experimental assay, if they would know that the chances it was tested and passed quality control are that low [1]. In bioinformatics analysis it is common to combine the functionalities that are coming from various packages. This should be encouraged further, but it also highlights the need of quality software, because with good software researchers spend less time on adopting it or maintenance [3,7,24]. In contrasts, bioinformatics software developers view their code as “means to an end” and care less about the future of the software they are writing. Bioinformaticians are not well aware of the relationship between complexity, size, age, and the change-proneness of a code, which heavily affect maintainability [20]. Another obstacle is the non-trivial nature of testing of scientific software [21,22]. In a recent review paper [22] two key aspects of scientific software testing has been highlighted: the oracle problem and the cultural differences between scientists and software engineers. Software behaviour can be tested against an expected output, but often in science we use software to find new knowledge. This results in an oracle problem, when scientists actually do not know *a priori* how the software should behave, thus straight forward verification is impossible. According to the authors, scientists also view their scientific model and the implementation as a single entity. Therefore, scientists tend to test the validity of the model but not verify the code which produces it. Uncovered faults can and do lead to incorrect scientific insights as shown in multiple examples [25].

The crisis of scientific software is fairly well known [26,27]. Without a shift in coding culture within bioinformatics, these concepts might remain unavailable to help bioinformatics professionals. Therefore, we need to increase the awareness and applicability of the software quality standards. Depending on the application of the scientific software, whether it is a tool or a data analysis pipeline, the authors may prioritize different quality attributes [3]. For example, in the world of big data, performance and efficiency gain importance. Shown in a previous study reviewing mappers, individual tools have varying level of compatibility, usability, and portability [9]; quality attributes which directly impact user experience. Frameworks, such as Snakemake [28] or Nextflow [29] support usability, reliability, and maintainability. Anaconda [30] and container solutions [31,32] help achieve portability. These are also compatible with Snakemake and Nextflow, making these frameworks staple for reproducible data analysis. Scientific software developer needs to prioritize the software quality attributes to make choices among the good practices [3]. Understandably, trying to implement all software quality practices can be daunting, especially while the main priority is to produce scientifically relevant results, and may actually discourage scientists to change their habits. We propose that coding in a team environment could help to ease into good software quality practices.

To our knowledge bioinformatics literature almost never presents suggestions how to code in a team setting and utilize multiple people's expertise on software development. Often guidelines for starting bioinformaticians encourage reaching out to others, but mostly to seek help when encountering a problem with the code. This could include consulting with colleagues, finding a mentor or participating in online communities (for example, Stack Overflow or Biostars) [18]. However, this does not represent code production and maintenance in the team and is still mainly focused on individual practices. In contrary to software engineering-oriented literature, where there is a lot of focus on practices when coding in a team [33,34]. The change in bioinformatics culture can be spotted though even if rare or unreported. Hagan et al. described Code Clubs - the practice in their research lab, where group members are collectively engaged in software development through code reviews and pair coding and software engineering education through workshops or seminars [35]. The authors give tips on how to organize such meetings and what should be the ground rules. Sharing your coding experience with others helps minimize the isolation, allows individuals to learn from their peers, and helps to write a better quality software.

Coding in team from software engineering perspective

Beyond the brief mentioning of getting support or coding in a team in guidelines for bioinformaticians, there is specialized literature that examines how to effectively organize coding activities in a team. Programming as a collective practice is a key notion in software engineering. A central theme in this literature is maximizing team cohesion while minimizing code coupling [36]. Authors (citation needed) argue that the viability of a software project along it successive development phases is largely determined by the adoption of sound software design enforcing modularity and extensibility coupled with team management practices centred around communication and collective governance.

In general, we understand management as the set of tasks ensuring the viability of a software project. These tasks revolve around planning, monitoring resources, and tracking progression [37]. Typically, the overseeing of these functions would be taken up by a single individual referred to as the “manager” of a project. In the particular context of computational projects in academia, a strict division of labour is rarely found in regard to the management of software projects. This means that the management and execution of a software project falls into the hands of the same person. Furthermore, some tasks, such as risk, budget and time management and maintenance are decoupled from the actual software development phase. The remaining management tasks would often be deliberated by the developer(s), eventually reaching a consensus on the desired way forward and acted upon. This sort of self-management, at times collective, echoes some prescriptions of the SCRUM method [38]. SCRUM is a framework to perform these management tasks through team self-management. This framework was introduced to respond to the aspiration for more autonomy and responsivity from software developers, best illustrated by the agile manifesto [39]. This proposition was a reaction to the typical blueprint-like management for engineering projects which proved ineffective in addressing the emerging challenges of large software projects [38]. The similarity is probably the reason why agile practices are part of guidelines for scientific software developers (Table 1).

One outstanding aim of agile is the aspiration for more autonomy for organizing the work of software developers. In the particular context of large computational project, agile opened the opportunity for collective governance and a move away from a project structure producing a division of labour coupling one developer to a particular task or aspect indefinitely. Incentivizing a collective ownership and governance of the codebase as a whole, promotes the adoption of software engineering best practices among developers contributing to a software project [40]. Indeed, by aspiring to make any developer within the team interchangeable across the various ongoing tasks, we create the need for robust testing, comprehensive documentation and coherence across the difference parts of the project [36]. Furthermore, by exposing every developer to a variety of tasks over the course of the project development, we strengthen the knowledge and skill base of the team as a whole, as well as create a better mutual awareness of team member expertise (transactive memory system [41]). Taken together these merits further improve the team's capacity to overcome technical challenges that will arise over the course of the development process.

The previous paragraph outlined some desirable outcomes of agile-like practices. Such benefits require the implementation and effective adoption of this mode of project management. In turn, it relies on the execution of a variety of methods whose success in realizing the merits of agile depends heavily on setting the adequate circumstances for the team to need to incorporate elements of agile in their regular work practice. Practices and methods aligned with agile prescriptions include stand-up meetings, task allocations, pair-programming, or code reviews. Note, that many of these practices do not require the presence of the manager, but assumes a work culture and standardized procedures. At their core, these practices incentivize continuous communication and collective decision-making among developers. This constitutes an additional overhead in terms of time and resources needed when developing, but this is offset by the aforementioned benefits in terms of coding practice, software resilience and improved team capabilities.

We do not believe that all the software engineering guidelines employed in the industry are necessarily relevant to the production of scientific software. The circumstances differ significantly, mainly due to the more personal nature of projects. Whether they are optimal or not, personal projects remain the norm, and it would be futile to expect another group member to achieve an equal level of familiarity with one's project. However, this should not prevent interactions between the people in the group, as it is through these interactions that rules are enforced and quality increased.

In our research groups, we have practically implemented the environment in which we, as a group, learn about and implement software quality practices that have been discussed in literature. We want to share this experience and propose how simple additions, such as weekly code review sessions or seminars, can lead to improved quality collective or personal software.

Our experiences for development processes involving teams

The preceding sections mention a lot of possible approaches to improving software quality. Given the abundance of opinions on this topic, and the variety of challenges bioinformaticians face, we believe that everyone should find out what works best for them. Here, we describe the practices that we have settled on.

The software development practices that we have adopted can be broadly separated into three categories: code reviews, what we have called software quality meetings, and resource sharing.

Code reviews

Code reviews are not a new invention and many people have discussed their benefits [38]. Here we would just like to briefly summarize how being made to present your code and receiving feedback leads to improvement in the process of creating software.

Prior to a scheduled code review, the author is forced to write their code in a way that it will be explainable and understood by others, which is always desirable. In a large distributed project this may be trivial, but because the bioinformatic projects are often handled by a single person, it is very possible to make the code needlessly complex and obfuscated. We also observed that during data analysis parts of the code are re-run in an ad-hoc manner (e.g. by commenting out parts), making it increasingly difficult to reproduce the same analysis.

During the code review, the author has to explain some aspect of their code clearly (e.g. structure, algorithm implementation, performance related decisions), which depends on them understanding it. Trying to explain your code to someone is shown to help with understanding, as with the rubber duck method [43]. The feedback obtained can help fix existing or potential future issues, improve the implementation, and produce cleaner, more concise code. The other participants may not be deeply familiar with the particular project, but they have their unique knowledge and point of view. We agree with the ten simple rules described by Hagan et al. [35], and note that many of those naturally emerged as a code of conduct after a few rounds of trial and error.

After the review, the received suggestions should be implemented swiftly to improve the code before advancing the project. The success of code review is highly dependent on its frequency. A long time between reviews means a lot of new code, difficulty to cover all changes in a single session, and potentially a lot of rewrite post review. This means that code reviews should be as regular and frequent as reasonably possible.

Our experience indicate a broader adoption of notions and practices of good software engineering standards highlighted during these code review sessions. Here we will focus on couple examples to illustrate how code reviews incentivised coding practices and team self-managements aligned with agile prescriptions. Code review involves some elements of problem solving, often revisiting fundamental notions of design patterns, algorithms or data structures. Recurringly we would examine best strategies to modularise the presented code and discuss what would consitute effective and self-contained computational task and elaborate collectively possible design patterns. This strengthens the team's overall competency as well as promoting some form of standardization regarding the mental models to use for common tasks and objects solicited in many computational projects. An important part of the code review process focuses on the compliance with good code practices, and constitutes an explicit attempt at standardization. This is particularly well illustrated with the review of documentation which goes beyond simple linting. Effectively this process promotes the adoption of a shared and systematic manner to describe and document the behavior of the considered tool, which facilitates its intelligibility for a wider audience.

As a positive additional outcome, we noticed an increasing understanding in each other's projects that naturally emerged through talking about the analysis code. This enabled us to give more involved comments during subsequent group meetings too. We noted however, that the focus can easily shift from the code to the biological question at hand. This we believe is more of a feature than a bug, as each code review session is led by the person bringing the code and the rest of us are there to support to the best of our abilities. Especially after the general level of coding style and quality increased to a good baseline. E.g. after about half a year, it was trivial for everyone involved that code organized into functions is preferred over spaghetti. The shared knowledge base and standards also allow us to make new group members adopt good coding practices more quickly.

Software quality meetings

Within the framework of software quality meetings, we have established larger-scale knowledge transfer between the participants. Presentations and demonstrations of new techniques and tools that are not necessarily tied to a specific project help broaden our knowledge base and awareness. In this sense, they form almost a substitute for a more formal computer science education, which most bioinformaticians lack [2]. Topics can arise from code reviews, own projects, or effectively be a reproduction of a useful talk or seminar given elsewhere.

The presenters benefit as well by having to research the topic further and present it coherently. It is not necessary to have these meetings be as regular as code reviews. The time investment is higher, given that a preparation is needed unlike just writing code as for code reviews.

During the software quality meetings, we have also explored the possibility of collaborative projects and pair programming, but have not managed to implement it successfully yet outside the scope of preparation for the JASPAR 2024 release [44]. The main reason for this is that we experimented with collaboration on a software tool not directly used by any of the members. As researchers, we could not afford to invest time in a hobby project.

The outcome of these sessions are manifold. A few examples: 1) a shared vocabulary that enables quick discussion about implementation details and code structures (e.g. design patterns, software architecture, data structures and algorithms), 2) a kind of toolkit and set of recordings we can sample from and build on in our own research projects (e.g. planning with UML diagrams, git features to ease and quicken software development), 3) awareness of previously unknown algorithms and packages, improving software performance and quality (e.g. dynamic programming, heap, Python packages such as bioframe).

Resource sharing

Resource sharing is a basic thing, but it boils down to making sure that useful online resources are brought to the attention of all participants easily.

Resource sharing could be discussed from two perspectives: external open-access resources (forums, repositories, packages and libraries) and internal (within-group resources with tools). The latter is very important as it allows for team contribution that can benefit the individual project development. A simple example of this could be a shared repository of various computational tools that were developed by members of the group. Such tools are universal enough and fit the group's research questions, so all people in the group can re-use them. In addition, each tool can be potentially developed and reviewed by multiple group members.

During software meetings, we aimed to set aside time to improve these tools from perspectives identified by the members. We observed that many of these tools do not have a clear scope and are rather a small script for a sub-task from a previous project. Based on this observation, we noted that there is a difference between a script and a standalone tool that can be inserted into various projects. The latter requires exploration of use cases related to the tool, handling of unexpected input, and extensive documentation, to name a few tasks. This understanding was actually quite relevant in a code review discussion when the expected usage modes of a new tool was the main focus.

We have chosen to illustrate the process of improvement in software writing by comparing it to rock climbing. At the top of the rock is our goal of reliable, performant and extensible software. In order to reach it, we need to become proficient in the various concepts depicted by the holds. The higher they are on the wall, the more advanced we consider the concepts to be. As the progress is gradual, we have chosen to show the holds in the same colour if they represent related concepts that build upon each other. The most important point, however, is the fact that rock climbing requires a partner to belay you, just as we believe the input of other people helps us become better programmers.

(Figure 1)

Figure 1: An illustration comparing the process of improvement in software writing to rock climbing.

Figure 1: An illustration comparing the process of improvement in software writing to rock climbing.

Conclusions and future perspectives

(From Ieva: I would call this conclusions and future perspectives and would keep the section sharp and short)

We have implemented a form of code review that fits our specific needs and context. We used our experience from these sessions, with special focus on team-based software development practices ensuring good code quality during in the update of the curation analysis software of the 2024 release of the JASPAR database [44]. In this project, as well as in some of our own individual research projects, we introduced user stories when documenting the assumptions, current features and new ideas, and we relied on development tools such as Jira and git. We note that the usage of these tools is not necessarily aligned with industry practices, due to the experimental nature of scientific software. Nevertheless, as bioinformatics becomes a more and more software-heavy field, we believe a good direction is to collectively lower the barrier to adapting to new technologies.

How to decide when it is time to invest in a script? Taschuk and Wilson [7] suggest a cut-off where a script is being reused, shared with others or used to produce findings in a publication. This definition would potentially include the majority of code written by bioinformaticians, but the time spent on improving the scripts should be weighed against the time required to deal with suboptimal code. Overall, as good practices become routine, the required time investment will be reduced and the benefits will become more apparent.

There is a method called ad-hoc testing, which could be beneficial for scientists without test plans or documentation. It is flexible, allows creativity and discovery of new requirements [45]. However, it is not advised to be used on its own, as it is not robust and requires extensive know-how.

The lack of funding for the maintenance of software, particularly scientific software, prevents achieving a level of software quality that would inspire confidence in the results. Funding is typically provided for the development of novel software, and it can be hard to justify spending time on maintenance which provides no output in terms of articles. We believe the infrastructure would benefit from funding earmarked for maintenance, and from dedicating time to it in project proposals.

Optional section that might be put in discussion instead.

- Comparison to current academic coding
  - agile would be more stringent
  - agile would challenge the one project : one developer configuration
  - agile would highlight the impossibility to capitalise dividends of long-term building of knowlegebase and team chemistry
  - agile would point at the training gap
- Approaches and circumstances promoting the benefits of agile

Let us not forget that academia comes from a different place than where agile was developed. The Agile Manifesto was written against “bureaucracy, infantilization, and sense of futility” [38]. The academic software development agile practices would in fact be a more stringent approach to software development than current practices. For example it would include writing down requirements in form of user stories, plan a minimal viable product, plan an initial architecture, and dividing the project into tasks. This is important, because most literature pictures agile as management style free from traditional management. We should not forget that ad-hoc coding does not comply with agile, and we cannot use it as an excuse to continue our current practices. The agile system assumes that software engineering professionals seek to find the best approaches, and are well equiped to make good decisions on their own - when faced with shifting requirements and complex code base [38]. Additionally, the lack of top-to-bottom management, architects and system analysts in academic software development put even more responsibility on the individual developer. However, as noted in previous sections, current scientific software developer education does not necessarily cover these elements [20].

- Cost-Benefit of adoption

Working in teams is not an option, but a must for large projects. Although it is not necessary in smaller projects, the benefits are significant. - over time a small project might be taken over by another person, thus accidentally becoming a sort of team project with (by definition) insufficient communication - lack of standards and good practices undermine quality and maintainability - ease of technical knowledge transfer (which requires time and social factors) [36] - knowledge on who knows what (transactive memory system) speeds up problem solving [41]

However, it is not trivial to assemble a well functioning team. Authors found that the overall performance increases when team members are familiar with each other and build problem-solving routines together through cumulative experience [36]. Miriam Posner also points out that team management practices do not protect from a toxic environment [38].

References

1. **Better Software, Better Research**
Carole Goble
IEEE Internet Computing (2014-09) https://doi.org/vjz
DOI: 10.1109/mic.2014.88

2. **Improving bioinformatics software quality through incorporation of software engineering practices**
Adeeb Noor
PeerJ Computer Science (2022-01-05) https://doi.org/gsm3hg
DOI: 10.7717/peerj-cs.839 · PMID: 35111923 · PMCID: PMC8771759

3. **Software engineering practices for scientific software development: A systematic mapping study**
Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Jeffrey C Carver
Journal of Systems and Software (2021-02) https://doi.org/gq3jtm
DOI: 10.1016/j.jss.2020.110848

4. https://c2.com/doc/oopsla92.html

5. **Managing technical debt**
Eric Allman
Communications of the ACM (2012-05) https://doi.org/grx4cv
DOI: 10.1145/2160718.2160733

6. **How do scientists develop and use scientific software?**
Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, Greg Wilson
2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (2009-05) https://doi.org/bw966x
DOI: 10.1109/secse.2009.5069155

7. **Ten simple rules for making research software more robust**
Morgan Taschuk, Greg Wilson
PLOS Computational Biology (2017-04-13) https://doi.org/gfvpqw
DOI: 10.1371/journal.pcbi.1005412 · PMID: 28407023 · PMCID: PMC5390961

8. **ISO 25010** https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3%20

9. **Empirical study on software and process quality in bioinformatics tools**
Katalin Ferenc, Konrad Otto, Francisco Gomes de Oliveira Neto, Marcela Dávila López, Jennifer Horkoff, Alexander Schliep
Cold Spring Harbor Laboratory (2022-03-13) https://doi.org/grx4jr
DOI: 10.1101/2022.03.10.483804

10. **A large-scale analysis of bioinformatics code on GitHub**
Pamela H Russell, Rachel L Johnson, Shreyas Ananthan, Benjamin Harnke, Nichole E Carlson
PLOS ONE (2018-10-31) https://doi.org/gskr8b
DOI: 10.1371/journal.pone.0205898 · PMID: 30379882 · PMCID: PMC6209220

11. **A survey of scientific software development**
Luke Nguyen-Hoan, Shayne Flint, Ramesh Sankaranarayana

Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (2010-09-16) https://doi.org/brfqvq
DOI: 10.1145/1852786.1852802

12. **CZI – Essential Open Source Software for Science**
Chan Zuckerberg Initiative
https://chanzuckerberg.com/eoss/

13. **Guidelines for bioinformatics of single-cell sequencing data analysis in Alzheimer's disease: review, recommendation, implementation and application**
Minghui Wang, Won-min Song, Chen Ming, Qian Wang, Xianxiao Zhou, Peng Xu, Azra Krek, Yonejung Yoon, Lap Ho, Miranda E Orr, … Bin Zhang
Molecular Neurodegeneration (2022-03-02) https://doi.org/gptgqt
DOI: 10.1186/s13024-022-00517-z · PMID: 35236372 · PMCID: PMC8889402

14. **A Clinician's Guide to Bioinformatics for Next-Generation Sequencing**
Nicholas Bradley Larson, Ann L Oberg, Alex A Adjei, Liguo Wang
Journal of Thoracic Oncology (2023-02) https://doi.org/gsm3mb
DOI: 10.1016/j.jtho.2022.11.006 · PMID: 36379355 · PMCID: PMC9870988

15. **Practice guidelines for development and validation of software, with particular focus on bioinformatics pipelines for processing NGS data in clinical diagnostic laboratories**
Nicola Whiffin, Kim Brugger, Joo Wook Ahn
PeerJ (2017-05-29) https://doi.org/gsm3mc
DOI: 10.7287/peerj.preprints.2996

16. **Standards and Guidelines for Validating Next-Generation Sequencing Bioinformatics Pipelines**
Somak Roy, Christopher Coldren, Arivarasan Karunamurthy, Nefize S Kip, Eric W Klee, Stephen E Lincoln, Annette Leon, Mrudula Pullambhatla, Robyn L Temple-Smolkin, Karl V Voelkerding, … Alexis B Carter
The Journal of Molecular Diagnostics (2018-01) https://doi.org/gcsstd
DOI: 10.1016/j.jmoldx.2017.11.003

17. **Top considerations for creating bioinformatics software documentation**
Mehran Karimzadeh, Michael M Hoffman
Briefings in Bioinformatics (2017-01-14) https://doi.org/bzmp
DOI: 10.1093/bib/bbw134 · PMID: 28088754 · PMCID: PMC6054259

18. **Ten simple rules for getting started with command-line bioinformatics**
Parice A Brandies, Carolyn J Hogg
PLOS Computational Biology (2021-02-18) https://doi.org/gh32h2
DOI: 10.1371/journal.pcbi.1008645 · PMID: 33600404 · PMCID: PMC7891784

19. **A Software Chasm: Software Engineering and Scientific Computing**
Diane F Kelly
IEEE Software (2007-11) https://doi.org/cbrmv5
DOI: 10.1109/ms.2007.155

20. **Software Engineering Education for Bioinformatics**
Medha Umarji, Carolyn Seaman, AGunes Koru, Hongfang Liu
2009 22nd Conference on Software Engineering Education and Training (2009) https://doi.org/b57ccg
DOI: 10.1109/cseet.2009.44

21. **Developing Scientific Software**

Judith Segal, Chris Morris
IEEE Software (2008-07) https://doi.org/bnm3xp
DOI: 10.1109/ms.2008.85

22. **Testing Scientific Software: A Systematic Literature Review**
Upulee Kanewala, James M Bieman
arXiv (2018) https://doi.org/gsrxg5
DOI: 10.48550/arxiv.1804.01954

23. **Handreichung Zum Umgang Mit Forschungssoftware**
Matthias Katerbow, Georg Feulner
Zenodo (2018-02-27) https://doi.org/ghk5fk
DOI: 10.5281/zenodo.1172970

24. **High-Performance Mobile Internet**
Anirban Mahanti, Subhabrata Sen
IEEE Internet Computing (2014-01) https://doi.org/gsszgq
DOI: 10.1109/mic.2014.8

25. **A Scientist's Nightmare: Software Problem Leads to Five Retractions**
Greg Miller
Science (2006-12-22) https://doi.org/fbvb8b
DOI: 10.1126/science.314.5807.1856

26. **Hunting for the best bioscience software tool? Check this database**
Matthew Hutson
Nature (2023-01-12) https://doi.org/gsnnww
DOI: 10.1038/d41586-023-00053-w

27. **Why science needs more research software engineers**
Chris Woolston
Nature (2022-05-31) https://doi.org/gsnnwt
DOI: 10.1038/d41586-022-01516-2

28. **Sustainable data analysis with Snakemake**
Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, … Johannes Köster
F1000Research (2021-01-18) https://doi.org/gjjkwv
DOI: 10.12688/f1000research.29032.1 · PMID: 34035898 · PMCID: PMC8114187

29. **Nextflow enables reproducible computational workflows**
Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, Cedric Notredame
Nature Biotechnology (2017-04) https://doi.org/gfj52z
DOI: 10.1038/nbt.3820

30. **Anaconda | The World's Most Popular Data Science Platform**
Anaconda
https://www.anaconda.com/

31. **Home**
Docker Documentation
(2023-08-22) https://docs.docker.com/

32. **Apptainer - Portable, Reproducible Containers** https://apptainer.org/

33. **Cooperative Software Development** https://faculty.washington.edu/ajko/books/cooperative-software-development

34. **Studying the impact of social interactions on software quality**
Nicolas Bettenburg, Ahmed E Hassan
Empirical Software Engineering (2012-04-28) https://doi.org/f4mhdp
DOI: 10.1007/s10664-012-9205-0

35. **Ten simple rules to increase computational skills among biologists with Code Clubs**
Ada K Hagan, Nicholas A Lesniak, Marcy J Balunas, Lucas Bishop, William L Close, Matthew D Doherty, Amanda G Elmore, Kaitlin J Flynn, Geoffrey D Hannigan, Charlie C Koumpouras, … Patrick D Schloss
PLOS Computational Biology (2020-08-27) https://doi.org/gg92xw
DOI: 10.1371/journal.pcbi.1008119 · PMID: 32853198 · PMCID: PMC7451508

36. **Team Familiarity, Role Experience, and Performance: Evidence from Indian Software Services**
Robert S Huckman, Bradley R Staats, David M Upton
Management Science (2009) https://www.jstor.org/stable/40539129

37. **What Is Software Project Management?** https://www.wrike.com/project-management-guide/faq/what-is-software-project-management/

38. **Agile and the Long Crisis of Software**
https://www.facebook.com/logicisamagazine
Logic(s) Magazine https://logicmag.io/clouds/agile-and-the-long-crisis-of-software/

39. **Manifesto for Agile Software Development** https://agilemanifesto.org/

40. **A teamwork model for understanding an agile team: A case study of a Scrum project**
Nils Brede Moe, Torgeir Dingsøyr, Tore Dybå
Information and Software Technology (2010-05) https://doi.org/cvr88b
DOI: 10.1016/j.infsof.2009.11.004

41. **Communication in Transactive Memory Systems: A Review and Multidimensional Network Perspective**
Bei Yan, Andrea B Hollingshead, Kristen S Alexander, Ignacio Cruz, Sonia Jawaid Shaikh
Small Group Research (2020-12-11) https://doi.org/ghpwvf
DOI: 10.1177/1046496420967764

42. **Walking the Talk: Adopting and Adapting Sustainable Scientific Software Development processes in a Small Biology Lab**
Michael R Crusoe, CTitus Brown
Journal of Open Research Software (2016-11-29) https://doi.org/gsmb78
DOI: 10.5334/jors.35 · PMID: 27942385 · PMCID: PMC5142744

43. **The pragmatic programmer: from journeyman to master**
Andrew Hunt, David Thomas
Addison-Wesley (2000)
ISBN: 9780201616224

44. **JASPAR 2024: 20th anniversary of the open-access database of transcription factor binding profiles**
Ieva Rauluseviciute, Rafael Riudavets-Puig, Romain Blanc-Mathieu, Jaime A Castro-Mondragon, Katalin Ferenc, Vipin Kumar, Roza Berhanu Lemma, Jérémy Lucas, Jeanne Chèneby, Damir Baranasic, … Anthony Mathelier

Nucleic Acids Research (2023-11-14) https://doi.org/gs4n8b
DOI: 10.1093/nar/gkad1059

45. **The Complete Guide to Ad hoc Testing**
BrowserStack
https://browserstack.wpengine.com/guide/adhoc-testing/