Practical software engineering for software-writing scientists

Raul P. Pelaez

Departamento de Fisica de la Materia Condensada, Universidad Autonoma de Madrid, Campus de Cantoblanco, Madrid 28049

(*Electronic mail: raulppelaez@gmail.com)

(Dated: 29 September 2025)

Scientists have long developed software as an integral part of their research, creating tools to simulate models or analyze data. As these projects grow in scope and longevity, they often encounter challenges related to complexity, maintainability, and reproducibility. This article introduces a set of practical software engineering techniques (such as version control, testing, documentation, continuous integration, and packaging) that can help address these challenges. Drawing on well-established principles from the field of software evolution, the discussion is tailored for scientists who write software but may not have formal training in software engineering. Some example projects are drawn from the chemical physics community, showcasing how following these principles benefits them and the community. It also reflects on the evolving role of large language models (LLMs) in scientific software development, considering both their potential and their limitations. The goal is to present approachable, widely adopted practices that support sustainable, transparent, and collaborative scientific software.

I. INTRODUCTION

Humans cannot produce simple software. Lehman and Belady's made us aware of this reality almost 50 years ago[1–3], and even by then the term "software crisis" was already an established concept. They came up with the so-called laws of software evolution. The first two can be summarized as follows:

1. A program that is used in a real-world environment must change, or it will become progressively less useful in that environment.
2. The complexity of a program will increase as the program evolves unless work is done to maintain or reduce the level of complexity.

The second point is the key to understanding the state of so many software projects produced by scientists. In essence, complexity will inevitably emerge and grow unbounded unless one *actively does something about it.* Computer scientists discovered this fact of life in the 70s, but they failed to communicate it to the rest of us physicists, chemists, biologists, etc. Nowadays, programming is a skill accessible to everyone. A student of any scientific discipline can learn to express any computable problem in code in just a few days. Biologists will need to analyze some data and write a script in R[4]. Applied mathematicians will need to solve a differential equation and write a script in Python[5]. Physicists will need to simulate a physical system and write a code in C++[6] or Matlab[7]. When these small scientist-written codes and scripts prove to be useful, they tend to be shared and grow as per the first rule above. Unfortunately, most of these scientists were not exposed to the second rule when the power of coding was granted to them. Thus, these soon-to-be large projects are born with the cross of unconstrained complexity growth. We seldom anticipate the size and scope of a software project will attain. Linus Torvalds first introduced Linux in 1991 by stating that it was "just a hobby, won't be big and professional"[8]. Scientists are often interested in solving the problem at hand, and not the philosophical intricacies of software development. Moreover, the learning resources available to them (such as online courses or as part of their university studies) often focus on teaching a specific language or framework, disregarding the broader context of software engineering.

Complexity will manifest in several aspects of the software's (and user's) life. Among other symptoms, these projects might become hard to understand/use (little to no documentation), untrustworthy (no tests), inflexible (high internal coupling), and hard to distribute (no stan-

dard build system, no packaging). The software engineering community has been working on this problem for decades. There exists a set of standard practices, tools, and methodologies that can help us tame complexity. These practices are sane, practical, and easy to adopt. More importantly, they are *standard*, meaning that other members of the community will *expect* to find them in your project.

In this article, we will discuss some practical and essential software engineering techniques that can help scientists manage the complexity of their software projects. Countless books and articles have been written on this topic for all levels of expertise and all kinds of audiences, see[9–14] to name some. The following is not intended to be a comprehensive guide to software engineering. Instead, it aims to provide a concise and practical introduction to some of the most important concepts and techniques tailored for scientists who write software but have not been exposed to good software engineering practices during their training.

This work focuses on the aspects of developing software that are not related to writing code. As such, the lessons in it transcend the choice of programming language or framework, focusing instead on the commonalities any software project should present. There is no discussion about coding strategies, abstractions, design patterns, or programming paradigms. The application of the practices here described will foster, in time, sane and healthy programming habits. While it is true that embracing these practices will require some initial effort that will hurt your short-term productivity, the long-term benefits are well worth it. For instance, documenting your code and the interfaces in it will help you realize which kinds of interfaces work and which do not. Testing your code will lead you to write it in a way that is easy to test, which will lead to better abstractions and design. Semantic versioning will help you understand the difference between the public interface of your software/library and an implementation detail.

Most importantly, these practices are *standard*, meaning that adhering to them will make your software more expectable (a.i., understandable) to other developers. In return, learning this common software engineering "language" will foster your ability to interact with a mature project that you did not create, as it will become understandable to you.

Advice

Tools vs. principles
Specific tools vary with the language, problem domain, and time. However, the principles (the kind of problems that arise and the general strategy to tackle them) have remained the same for decades. I encourage the reader to challenge the relevance of the specific tools mentioned throughout this article, especially if a long time has passed since its publication. Some of them, like the `conda-forge`[15] repository, are community-maintained. The community will shift its collective efforts to new technologies as they come and go.

II. COMMON FALLACIES AND PITFALLS

There is really no good reason to neglect good software practices. Can you trust the code you wrote a year ago if you cannot test its correctness? Can you understand that code in order to fix a bug or add a feature if it is not documented? Can you share that code with a colleague if it is not properly packaged? Does it even run on your machine, which has been updated several times since you last ran it? Most people will agree on the importance of good software engineering practices, but many will not apply them. I would like to address the most common excuses for this behavior and provide some counterarguments.

My code is too small or inconsequential

When starting some project (be it an innocuous analysis script, a measuring tool, a solver...), there is always a chance for it to become useful, thus growing in scope and functionality as per the first rule. In that case, complexity will rot your codebase and burden you immensely if you do not deal with it. The time and effort it takes to apply good practices from the ground up is so minute that it is not worth the risk of not doing it. Furthermore, we humans seem to be really bad at judging the impact of a project at its inception. Slack started as an internal communication tool for a video game called Glitch, which never took off[16]. The developers of the Zoom video call app[17] were probably not ex-

pecting a global pandemic to hit and make their app a household name. And so on.

I just want to solve my problem, I do not care about programming

This is a common sentiment among scientists. The fact that learning to code is so easy and accessible makes it tempting to just write some code and move on. When learning to drive, one could technically get away with knowing just how to accelerate and brake, but lacking the ability to read traffic signs will eventually lead to a crash. The same is true for coding. You can get away with just knowing how to write code, but you will eventually crash into complexity if you do not learn how to manage it. You might have found learning the rules of the road boring, but be it by law or common sense, you endured learning them. Think of software engineering as the rules of the road for coding. Your users (which includes you in the future) will thank you for it.

I do not have time

Once good practices are internalized, the difference between writing a quick and dirty Python script and creating a full-fledged package that can be tested, distributed, and understood is 10-15 minutes. Often, the engineering requirements of a new project (dependencies, language, and/or framework, etc) will be very similar to those of a previous one, allowing for reuse of the infrastructure with minor modification.

I will do it later

The longer you wait to apply these practices, the harder it will be to do so. It will snowball into a mess that will be hard to untangle (this is known as a *big ball of mud* in technical terms[18]). Employing these practices makes your life easier in the long term, and their benefits are immediate. You will be able to understand your code better, you will write code that you can trust, and share it more easily. Once you have gone through the process of applying these practices to a project, applying them to a future project will take you a minuscule amount of time. When neglecting good practices, you are accruing technical debt, which you (or your peers) will have to pay back later, with interest. Real life is nuanced, and there are times when you will have to prioritize code production beyond all else. In these cases, it is paramount that you understand that there is a price to pay later.

I am pressured to deliver quick results

Senior researchers are often not encouraged, nor rewarded, for following good practices during project reviews. Grant applications and journal submissions have minimal software quality requirements, often limited to making the code available publicly. Researchers might then impose these low standards onto their students and postdocs, perpetuating a vicious cycle of low-quality software brought down by the institutional pressure to produce short-term results.

This systemic problem must be addressed at the community level, with journals and funding agencies setting higher standards for software quality. Some journals are taking steps in the right direction. For instance, the Journal of Open Source Software (JOSS)[19], states a series of requirements for software submissions in their review criteria, including the software to be easily testable and well documented. The Biophysical Journal is more succinct in their author guidelines, stating that "reviewers will be asked to test-drive the computational tool and judge its usability"[20]. As the complexity of the software ecosystems around us grows, it is my hope and expectation that more journals and funding agencies will adapt by being more stringent with their software quality requirements.

Individually, we can all contribute to raising the bar of scientific software. When reviewing a paper that includes software, we can advocate for the application of good practices. When contributing software ourselves, we must strive to publish code we are proud of. Code that we can assure is correct and reusable.

III. GOOD PRACTICES

It might seem that by following these practices, one is adding complexity to a project. After all, they require including more elements in our codebase. The opposite is indeed the case in the long term; the practices covered here are all intended to help manage complexity. The second law of software evolution urges us to proactively tame complexity, and these practices are the tools to do so.

In your software-writing journey, you will face certain "inconveniences". You must train yourself to recognize them and look for a standard

solution that the software community has already provided. Let me go over an example:

At some point, you might have a working piece of software. You came up with an improvement and decided to implement it. Naturally, you missed something, and now some tests are failing. If you had enough foresight, you might have stored a copy of your project in another place just in case. You are not sure which of the many changes you made introduced the bug and thus are forced to just revert to your backup entirely. If you do not have a backup, you are forced to fix or revert the changes manually.

This "inconvenience" is trivially solved with version control. Lacking the knowledge, one can easily fall into asking the wrong question. In the previous example, a novice might look for a backup solution, a la Google Drive, instead of stumbling upon version control. Try to identify thoughts in the form of: "It would be great if X existed". With experience, the intuition will come on how to formulate the right questions.

Advice

Do not get overwhelmed
Familiarity with these tools and practices will eventually streamline a project's workflow to the point where working without them will be unthinkable. However, achieving this familiarity takes time, effort, and a lot of trial and error. A novice will easily become overwhelmed by the depth and reach complexity of some tools (e.g. `git`). I encourage you to take a pragmatic approach and integrate them into your workflow little by little as the need arises (which will manifest in the form of "inconveniences"). When needed, the contents of this article will serve as a reference to know how to proceed.
As a personal note, it took me many months to go over[9], which is divided into small lessons. I would go over one of them and take it into account during the following programming sessions. Only when I decided I had internalized the lesson would I go on to the next one.

A. Keep track of your project's history

Definition

Version control (VCS)
The practice of tracking and managing changes to your code. VCS is not only useful for code; it can be used for any text-based workflow, collaborative or not. For instance, articles, class notes, to-do lists, or configuration files can benefit from version control.

VCS serves three main purposes:

- **Stores the history of your project**. This is useful to track changes, go back in time, and understand why a certain change was made.
- **Facilitates collaboration**. When combined with a repository hosting service (such as GitHub, Gitlab, Bitbucket, etc.), it allows multiple people to work on the same project without stepping on each other's toes. Each person can work on their copy and merge their changes into the main one when ready.
- **Serves as a form of backup** by a combination of the two previous points and a bit of discipline by synchronizing with the remote hosting service often. If access to the local copy of the project is lost, it is possible to clone it again from the repository hosting service.

The most popular VCS tool is Git[21], another gift to humanity from Linus Torvalds besides Linux.

Advice

Should you choose a single good practice to start with, it should be git (which is why this article goes a bit more in depth with it). The first resource for learning git should be[22].

In git, the current state of a codebase is tracked as a branching sequence of incremental changes to groupings of files. Each collection of changes is called a commit, and each commit is assigned a unique ID called a hash. An example git hash could look like the hexadecimal string `2fd4e1c67a2d28fced849ee1bb76e7391b93eb12`, but note the simplified representation using

capital letters to denote commits in figure 1. Each commit represents a snapshot of the repository at a given point in time, along with some metadata (such as the author, the date, and a message describing the changes). The first commit (the initial state) is called the root commit. Commits are connected to each other in a parent-child relationship. This structure facilitates the tracking of changes and the ability to move back and forth in time. A folder controlled by git is called a repository. A repository can be local (on your computer) or remote (on a server). The most common way to host a remote repository is using a service like GitHub, Gitlab, or Bitbucket. These services provide a web interface to manage your repositories, as well as additional features like issue tracking, pull requests, and continuous integration.

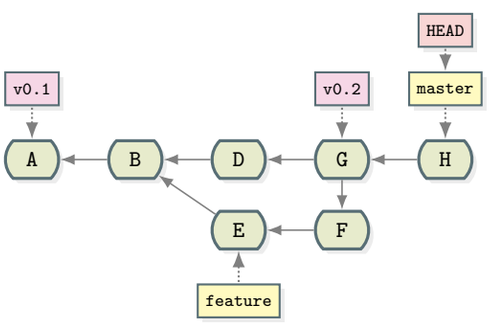

FIG. 1. A simple git history with two branches (master and feature) and two tags (v0.1 and v0.2). The current branch is master , as indicated by the HEAD pointer. Commits are represented by the green nodes and are labeled with letters. Each commit encodes the state of the whole project at some point in time, which git allows to visit (checkout) and name (tag). Continuous arrows represent parent-child relationships between commits. For example, the state of the repository at version 0.2 (commit G) comes from applying G to commit D , and then B , to the initial state of the repository, A . One of git 's killer features is that we can "branch off" from a commit to create a new line of development. This is useful when you want to work on a new feature without affecting the main branch. When you are sure the feature is ready, you can merge it back into the main branch. Above, the feature branch is created from commit E and merged back into the master branch at commit G .

All that is known to git will be stored in a hidden folder called `.git`, and we can interact with it using the `git` command line tool. Some common commands are:

- `git init`: Initializes a new git repository in the current directory.
- `git clone <repo_url>`: Clones a remote repository to the current directory.
- `git add <file>`: Stages a file for commit, meaning that the file will be included in the next commit.
- `git commit -m "<message>"`: Commits the staged files with a message describing the changes.

These commands can be issued from the command line, but most code editors have integrated support for git, allowing users to apply VCS via a graphical interface. For instance, in VS Code[23], you can see the current state of the repository in the source control tab. You can stage and commit files from there, as well as view the history of the repository.

We can tag individual commits to mark them as special. Typically, tags are used to mark release versions. This is a key feature, since it allows us to "freeze" the state of the repository at a given point in time, marking it as special. So if a user experiences a bug (or you uncover a problem by processing the results of the code), you can go back to the exact state of the code that produced the bug to investigate it. All users of the code will also be able to know if they are using the bogus version of the code, and update it to the fixed version.

1. *Semantic versioning*

It is useful to fix certain commits as releases. This is done by tagging them with a version number (see figure 1). Semantic versioning is a convention that serves to name these versions in a way that is predictable to users. A version number is composed of three numbers separated by dots: MAJOR.MINOR.PATCH. The MAJOR number is incremented when you modify the Application Public Interface (API) of your software. The API is a contract that a program makes with its consumers, setting rules, protocols and definitions for interacting with it. When the API changes, users of your software will have to change their code or workflow to adapt. The MINOR number is incremented when you add functionality in a backwards-compatible manner (when you add new features to your code, but the public interface re-

mains the same). The PATCH number is incremented when you make backwards-compatible bug fixes. So, if my project depends on a library that is version `1.2.3`, I can expect that upgrading to version `1.3.7` will not break my code, but upgrading to version `2.0.0` might.

Most dependency managers (such as `conda` or `pip`[24]) allow you to specify dependencies using semantic versioning. For instance, the following is an example of an `environment.yml` file that specifies the dependencies of a Python project using `conda`:

```
name: my_project
dependencies:
  - python==3.12
  - cuda-version==12.*
  - numpy>=1.20.0<2.0.0
  - matplotlib 3.*
```

By marking the package `matplotlib` version as `3.*`, this file states that the project will be compatible with any version from `3.0.0` to `4.0.0` (non-inclusive). It can make this promise because `matplotlib` follows semantic versioning, so that a code that works for version `3.0.0` should work without changes in functionality up until the next major version, `4`.

In accordance with the tradition in the software engineer community, the conveniently named webpage `https://semver.org/` lays down everything there is to know about semantic versioning.

I invite the reader to visit and explore two projects of mine that are git enabled and hosted on GitHub: Superpunto[25], a purely C++ project, and spreadinterp[26], a mixture of Python and C++/CUDA. GitHub serves as a visual frontend for git, allowing users to visit any commit, tag, or release. For instance, visiting `https://github.com/RaulPPelaez/superpunto/commits/master/` will produce a list of commits that one can visit (going back to 2015). Visiting `https://github.com/RaulPPelaez/superpunto/releases` will list all releases (tags) with a showcase of the differences to the previous version and will also allow downloading the project at that version. All this structured information can be produced by GitHub automatically and seamlessly because those repositories integrate git and versioning via tags.

B. Test your code

Definition

Unit testing
The practice of writing small, contained, and reproducible tests for every functionality added to your software. These tests should be run constantly (and automatically) when introducing changes to your code.

Unit tests are the most basic form of testing. They test the smallest possible unit of code (a function, a class, a module) in isolation. Unit tests have several benefits:

- Ensuring that the code behaves as expected.
- Documenting the expected behavior of the code.
- Catching bugs early in the development process.

A healthy test suite will grant you the confidence that the introduction of functionality `B` does not break functionality `A`. This not only includes the possible introduction of bugs, but also warns you of unintentional changes to the public interface of your code.

Advice

Test-driven development (TDD)
TDD is a software development strategy based on unit tests, which consists, essentially, of repeating the following cycle:

1. Write a test that fails.
2. Write the minimum amount of code to make the test pass.

In essence, TDD proposes to design and write the tests before implementing the functionality to be tested. According to Kent Beck in[27], TDD is "Quite simply [...] meant to eliminate fear in application development". By employing TDD, you will write code that is correct by construction. As your codebase grows, TDD will give you the assurance that no new functionality breaks the existing one.
I encourage you to try TDD in your next project. While its dogmatic adherence

might be counterproductive in practice, its basic philosophy is sound and should be followed. Writing a test before the implementation that satisfies it will force you to think about the problem at hand and design a good interface for it. This will lead to better abstractions and a more maintainable codebase.

The community has developed many testing frameworks that help focusing on writing and running tests in a standardized manner. Most languages have a built-in or standard testing framework. For instance, `Python` has `unittest` and `pytest`, `C++` has `Google Test` and `cppUnit`, `Matlab` has the `Unit Testing Framework`, `R` has `testthat`, etc. They are all similar in spirit, but adapted to the language they are written in/for.

In `Python` you could write a test for a function `my_function` in a file `test_my_module.py` as follows:

```python
# project/tests/test_my_module.py
from my_module import foo
def test_foo_returns_2():
    assert foo() == 2
```

And you would run the test using the `pytest` command in the terminal, which would output something like:

```
$ pytest
test_my_module.py .     [100%]

===== 1 passed in 0.01s =====
```

`pytest` will automatically discover and run all functions that start with `test_` in all files that start with `test_` in the project directory. Frameworks for other languages work in a similar way.

Anecdote

A researcher who had recently run into the consequences of not following good practices recently approached me. This researcher works in a laboratory and writes code that interfaces with actual lab hardware (like oscilloscopes or power supplies) in MATLAB. When introducing them to unit testing, they expressed their frustration with the fact that they could not test their code because it was dependent on the hardware, which they could not have available at all times when developing. This is a common misconception. Unit tests should not depend on external resources. Instead, you should *mock* these resources. Mocking is the practice of creating a fake version of a resource that behaves like the real one. This way you can test your code in isolation. For instance, if you are writing a function that deals with data from an oscilloscope, you could mock the oscilloscope by providing this function with a fake data source that behaves like the oscilloscope, say, a pure sine wave. If you design your code carefully, mocking will not have an impact on the rest of your interfaces.
There are several tools that can help you with mocking. For instance, Python has `unittest.mock`, C++ has `Google Mock`, and Matlab has the `Mocking Framework`.

Advice

Beware of the fallacy of thinking that one develops faster if healthy practices such as testing, design, or documentation are neglected. The time saved by not writing tests is time that will be spent later debugging, which is time that is not spent doing science. In the age of Large Language Models (LLMs)[28], we oftentimes do not even write most of the code ourselves. It is now more important than ever to go out of your way to write a comprehensive test suite for your project.

1. TDD: A practical example

The notion of writing tests for a code that does not exist yet might sound counterintuitive at first. Doing so, however, will force you to think about the problem at hand and design a good interface for it while being released from the burden of having to implement it. This is the essence of test-driven development (TDD). Let us go over an example.

Suppose that as part of a project, you need to implement a function that finds peaks in a signal. This could be used for spike detection in neural signals, heartbeats, event detection via sound... There are countless ways to define a "peak", and countless ways to implement a function that finds them. We could use a simple threshold, or differentiation, spectral analysis, etc. At this stage, TDD tells us not to worry about the implementation, but rather to focus on the interface and the expected behavior of the function. We should start be writing a test for some trivial case, such as when the signal has a single peak (or even simpler, when the signal is flat, or empty), e.g. $f(t) = \delta(t - t0)$, or as a list of numbers, e.g. `[0, 0, 1, 0, 0]`.

At this point, we realize that in order to write this test we need to define the interface of the function (note that we have already determined that this should be a function, as opposed to a class or command line utility), let us call it `find_peaks` (the Python function `scipy.signal.find_peaks` implements similar functionality). What arguments should it take? It could be a list of numbers, or a path to a file, or some descriptor for a device that provides the signal. On the other hand, what should this function return? Perhaps the number of peaks would suffice, or the indices where the peaks take place, or the values of the peaks themselves. The space of possible designs is vast, and just thinking about one simple test forces us to narrow it down to a single interface. Let us use our knowledge about the problem domain to come up with a sensible set of constraints. Let us say that the kind of signals we need to process are *not* equidistant in time, and that we need to find the times at which the peaks occur. Let us write the related code in Python, with the intention of the code being close to pseudocode, so that it can be easily translated to any language. The signature of the function could look like this:

```
def find_peaks(time: list[float],
↪  signal: list[float]) ->
↪  list[float]:
```

The first test could then be written as follows:

```
def test_find_peaks_single_peak():
    time = [0, 1.5, 2, 3, 7]
    signal = [0, 0, 1, 0, 0]
    # There should be one peak that
    ↪  occurs at time=2
    assert find_peaks(time, signal) ==
    ↪  [2]
```

TDD tells us to write just enough code to make this test pass. Technically speaking, we could then define the function as just returning `[2]`. Then, we would write another test for a more complex case, such as when the signal has multiple peaks. That test would then fail, thus forcing us to write more code to make it pass. And so on and so forth. In my opinion, it is not necessary to be so dogmatic about it; perhaps we can write the second test to have an arbitrary number of peaks, with arbitrary locations. Regardless of how we grow towards the final implementation, the interesting thing about this procedure is that, as we increase the complexity of the implementation (and/or the interface), we will be sure that the function behaves as expected. In this particular case, our implementation will always have to satisfy the `test_find_peaks_single_peak` test.

Note that, thus far, we have not had to worry about how to implement a generic peak finder, or even what exactly a peak is. These aspects will naturally arise as we design and write more tests.

2. Continuous integration (CI)

Definition

Continuous integration
The practice of automatically running tests and other checks on your code whenever you push changes to a remote repository. This ensures that your code is always in a working state and that no changes break existing functionality.

Unit tests are only useful when executed continuously, at every change (or commit) to the codebase. Alas, humans cannot be trusted to do

so manually. Continuous integration ensures no change breaks existing functionality. Most CI services are free for open-source projects. For instance, GitHub Actions, Gitlab CI, Travis CI, Circle CI, etc. Another benefit of CI is that it forces us to execute our software on a different machine than the one where development takes place. This oftentimes helps us detect and fix issues (such as in dependency installation) that were overlooked due to the particularities of the development system.

Advice

Continuous integration scripts are great for automating chores, but setting them up can be tricky and error-prone. The documentation for the CI services is often unintelligible for the average scientist (when not outright wrong or outdated). I encourage you to keep this technique in your radar, but leave its implementation for a late stage in your software-developing journey.

Applying CI amounts to a special script hosted in your repository that is executed whenever you push changes to your repository. This script will install the dependencies of a project, build and install the project, and finally run the tests. If any of these steps fail, the CI service will notify the developers. Below is an example of a GitHub Action that will run on every commit pushed to the master branch, on every commit pushed to a pull request, and also every Sunday at midnight. The script, which is written in a YAML format, will install and test the software on a machine with Ubuntu (runners are also available with Windows and OSX), which is made available to the developers by GitHub. The rules in this example could serve for a typical Python project:

```yaml
# project/.github/workflows/ci.yaml
name: CI
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
  schedule:
    - cron:  '0 0 * * 0'
jobs:
  runs-on: ubuntu-latest
  steps:
    - name: Get the source
      uses: actions/checkout@v4
    - name: Install dependencies
      run: pip install -r
       ↪  requirements.txt
    - name: Install the library
      run: pip install .
    - name: Run tests using pytest
      run: pytest
```

GitHub will automatically pick this script up when placed inside a directory called `.github/workflows/` as a file with a `yaml` extension. The GitHub web interface will provide a plethora of information and logs about the execution of this script.

C. Document your project

Good documentation reduces cognitive load and facilitates the use and extension of a project by both its original author and others. Documentation must not be an afterthought; it must grow alongside the code. It should live close to the code, and ideally be generated directly from it. Most languages provide standard mechanisms for this. In Python, docstrings and the Sphinx system (with extensions like `autodoc` and `napoleon`) allow for the extraction of documentation from the source itself. C++ has Doxygen, Java has Javadoc, and so on. For instance, in Python, you can write a docstring for a function as follows:

```python
def register_user(email: str, password:
↪  str) -> bool:
    """Adds a new user to the system.

      :param email: The email of the
      ↪  user. Must be a valid email
      ↪  address.
      :param password: The password of
      ↪  the user. Must be at least 8
      ↪  characters long.
      :return: True if the user was
      ↪  registered successfully,
      ↪  False otherwise.
    """
    # ...
    # Code to register the user
    # ...
```

Tools like Sphinx can then extract this information and generate a nicely formatted documentation page. By having the documentation in

the code itself, we reduce the risk of it becoming outdated or inaccurate while reducing the development time needed to write it. A multitude of programs have chosen Sphinx as their documentation generator, such as Calibre, OpenCV, and even Python itself. The interesting thing about these examples is that most of what you can find on those webpages is generated from the code itself. A smaller example is the documentation of the Python package `spreadinterp` `https://spreadinterp.readthedocs.io/`, which is generated from the docstrings in the code using Sphinx and hosted on Read the Docs `https://readthedocs.org/`.

Advice

Automatic documentation prevents documentation rot
Invalid code will produce bugs that trigger tests or cause the code to not even compile. On the other hand, documentation, as well as comments in the code, do not need to be correct in order for the code to execute or compile. This makes it possible for documentation/comments to *rot*, becoming false information as the code evolves, but we forget to update the documentation. Keeping the documentation integrated (by using docstrings, for instance), as opposed to having a separate document, reduces the chance of it becoming outdated. A plethora of tools exist to then extract and render this documentation in a browsable format.

Picture yourself as a new user of your software and go to your project's repository. Would you be able to understand how to install and use it? Would you be able to understand how to extend it? Would you know how to test its correctness? If the answer to any of these questions is no, then you need to work on your documentation. Users will look for a `README` file when first encountering a new project. Most repository hosting sites have adapted to this by automatically rendering any `README`-like file if found when accessing a repository page. The `README` file is thus the first thing a user will see when they visit your project. It should be clear, concise, and informative. It should include installation instructions, usage examples, and a way to reach the documentation, either as a link to the documentation site or as instructions to generate it from the repository.

Users want to know how to install and run the software, and what functions or classes are available for them to call. Developers, on the other hand, need to understand the internal structure of the project and the rationale behind design decisions. These two audiences require different kinds of documentation. The public-facing documentation, intended for users, should include tutorials, usage examples, installation instructions, and a complete reference for the public API. This content should be rendered and hosted in a browsable format, and linked prominently from the `README`. Developers benefit from inline comments, architectural notes, and contributing guides that explain how to extend the code and follow its conventions.

Documentation must be easily accessible whenever the code is. That includes being part of the repository itself in source form, and also being available in rendered form online. Services like `Read the Docs` allow for generating and hosting rendered documentation automatically from a GitHub repository, triggered every time the main branch or a release tag is updated. Placing the necessary configuration for `Read the Docs` will allow the documentation to be rebuilt automatically upon changes. This setup can also be integrated into your CI system, so that documentation builds fail if there are errors or broken links. Alternatively, GitHub offers so-called GitHub pages, which allow to host a website as part of the repository itself. We can leverage this service via a GitHub action, making it so that the documentation is built and published at every release or commit.

Sometimes a well-written `README` is enough. A self-contained script or small tool does not need a full documentation site. If all usage and installation instructions can be captured clearly in a few paragraphs, a structured `README` is an appropriate and lightweight choice. Still, that `README` should be version controlled and maintained just like the code. The same discipline applies; it is worth investing some time in writing it well. For instance, the repository `MeanSquareDisplacement`[29] (hosted here `https://github.com/RaulPPelaez/MeanSquareDisplacement`) has a `README` that explains how to install and use the code, as well as how to contribute to it. It relies on automatic documentation generation just for the public API (a couple of functions).

Remember that documentation is not a favor

to others. It is a tool for your own future self. Writing it now saves you time later. You will forget why a particular decision was made or how a complex function works. Documenting those details today is the only reliable way to recover them when needed.

D. Make your software easy to install and run

Definition

Packaging
The practice of preparing your software so that it can be easily installed and run by others. This includes automating the installation of dependencies, building the code, and providing a way to install it in a standardized manner.

Software must be shared to be useful. However, unless you take care to make your software easy to install and run on other machines, it will only ever work on your own setup. Sharing a `.zip` file or dumping a bunch of files on a colleague will not do. Packaging must be automated, standardized, and portable. This means using the tools that the community expects.

When trying to use a software project, users will have to follow these steps:

1. Get the source code.
2. Install the dependencies.
3. Build and install the code.

If the project is already packaged and hosted in some package manager, these steps are combined into a single command, such as `pip install my_project` or `conda install my_project`. However, if the project is not packaged, or we are the developers of it, we will have to follow these steps manually. Let us go over them in detail.

The first step is to get the source code, which is normally accomplished by cloning a git repository.

For dependencies, you should rely on a cross-platform, automatic dependency manager. This is a tool that, given a list of dependencies, will install them in the right versions for the target platform. These tools allow others to reproduce your development environment regardless of their platform (like operating system or hardware architecture). A particularly versatile option is `conda`[30], which works across multiple languages and allows you to specify an `environment.yml` file that lists all dependencies. If such a file is present, running the command `conda env create` will create a new conda environment with all the dependencies specified in the file. An environment is a self-contained directory that contains all the dependencies needed to run a project, including the language interpreter itself. This allows you to have multiple environments with different dependencies on the same machine, and switch between them easily. Below is an example of an `environment.yml` file for a project that mixes Python and C++ code:

```yaml
name: my_project
channels:
  - conda-forge
dependencies:
  - python
  - pip
  - cmake
  - gxx
  - make
```

This file specifies that the project requires packages called Python, pip, CMake, g++, and make to build and run. The `conda-forge`[15] channel is a community-maintained repository of conda packages that includes many scientific libraries and tools. Instead of listing dependencies in a README and expecting users to install them manually, you can provide this file and let conda handle the installation. Besides the convenience, working with environments also allows you to avoid dependency conflicts, as each environment is isolated from the others. On the other hand, other tools will integrate nicely with virtual environments, such as continuous integration services.

Other options include `pip`[24] for Python, `vcpkg`[31] or `conan`[32] for C++, the built-in tools for Matlab or R, and so on. Each language has its own set of tools, but the idea is the same: provide a way to specify and install dependencies in a reproducible manner. The community has converged on standard tools for each language, so you should use those instead of inventing your own strategy.

Once dependencies are taken care of, your code must be built and packaged using a standard build system. Each language community has

one or two accepted tools for this task. In Python, the standard is to write a file called `pyproject.toml` that lists dependencies, metadata, and build instructions. This file is used by tools like `pip`, `setuptools` and `poetry` to build and package the code. In C++ and other compiled languages, there is `CMake`[33], which generates platform-specific build instructions from a common `CMakeLists.txt` file. Matlab has the `Matlab Compiler`, and most modern languages include a build tool as part of their standard tooling. These tools know how to build and install your software in a reproducible, and often cross-platform, way.

1. Distribution

Most users of your software will not be interested in modifying it, and therefore are not interested in getting the source code and building your package. You should distribute your software so that others can install it using a standard package manager. This means turning your code into a package with metadata, dependencies, and version information, and making it available through a registry. In Python, this can be done by uploading a `pip wheel` to `PyPI` so others can install it using `pip`. In C++, `vcpkg` or `conan` will do the same. Anyone can upload a `conda` package (which is created via a tool called `conda-build`) to `Anaconda`[30] or `conda-forge`, which accepts packages for any language. This allows users to install your software with a single command, such as `conda install my_project` or `pip install my_project`. If we have adhered to standard practices to set up our project (in terms of dependencies, build system, tests, etc), packaging it will be a matter of running a single command. For instance, in Python, we can create and upload a `pip` package by running the following from the root of our project:

```
$ pip install build twine
$ python -m build --wheel
$ twine upload -r testpypi dist/*
$ twine upload dist/*
```

Here, we install some necessary dependencies for building and uploading `pip wheels` from a Python project, and then we issue the `python` command to build the project. The last two commands instruct the `twine` tool to first upload the resulting package to `testpypi` (a staging area for verifying package correctness), and then, after checking that everything is correct, uploading to the `PyPI` package index. These steps are laid down in detail at the twine documentation. Users can then install the package using `pip install my_project`.

An easy way to distribute your software is to use GitHub releases. This allows you to create a release version of your software that can be downloaded and installed by others. Tools like `pip` are prepared to directly understand URLs to git repositories as sources for packages.

E. Standards are important

Making your software project *expectable* helps the users in understanding it. In addition, following the community standards will make integration and interoperability with tools seamless and mostly automatic. For instance, if the root of your project contains a file called `README.md`, GitHub will render it as the main page of your project. If you have a file called `environment.yml`, `conda` will recognize it, and the VSCode editor will try to automatically enable it when editing something in that project. Naming your test files with a `test_` prefix will allow `pytest` to automatically discover them, and so on and so forth.

The cognitive load of a new user will be much lower if they can expect to find a `README.md` file in the root of your project, or a `tests/` folder with the tests. If they see a file called `setup.py`, they can instantly guess that your project is installed with `pip install`. Below is an example of a typical Python project structure skeleton that follows these conventions:

```
my_project/
|-- src/
|   \-- my_project/
|       |-- __init__.py
|       \-- ... other source codes ...
|-- tests/
|   \-- test_something.py
|-- docs/
|   \-- ... documentation files for
↪  Sphinx...
|-- .gitignore
|-- pyproject.toml
|-- README.md
|-- .github/workflows/
    \-- ci.yml
```

The `src/` folder contains the source code of the project, which is organized in a package called `my_project`. The `tests/` folder contains the tests for the project, and the naming suggests that these are compatible with `pytest`. The `docs/` folder contains the documentation files for Sphinx, which will be used to generate the documentation site. The `.gitignore` file tells git which files and folders to ignore when committing changes. The `pyproject.toml` file contains the metadata and dependencies of the project, and the `README.md` file contains the installation and usage instructions. Finally, the `.github/workflows/ci.yml` file contains the configuration for the continuous integration service, which might, for instance, try to build the project, run the tests, and generate the documentation site automatically, ensuring that everything works as expected.

When a user encounters this project, they can infer how to get its dependencies and install it (using `pip`), test it (using `pytest`), and build the documentation (using `sphinx-build`). They can also expect to find the source code in the `src/my_project/` folder. By looking at the CI file, the user can have an idea of how the project is built and tested, what dependencies it has, and how they are obtained. The user can gather all this information without even having to look at the `README` or the documentation.

This is a simple example, but it illustrates how following conventions and standards can make your project more approachable and easier to use. Projects of all sizes make use of this basic structure, such as SciPy, TorchMD-Net[34] or Home Assistant, just to name some.

The specific tools and conventions will vary depending on the language and software stack of the project, but the general principles remain the same. There will always be a file (or files) that deal with dependencies, a file that deals with the build system, a folder for tests, a folder for documentation, and so on and so forth. Other times, such as the example above, a single file takes care of several of these tasks, such as the `pyproject.toml` file in Python, which takes care of dependencies, installation and packaging. Most importantly, these files will be named and located following the community conventions.

Advice

dotfiles
Dotfiles are configuration files that are used to customize the behavior of various tools and applications. They are usually hidden by default (starting with a dot). Examples you might find in a repository include `.gitignore` (which tells git which files to ignore), `.clang-format` (which specifies the formatting rules for C++ code), `.travis.yml` (configuration for the CI service Travis CI), and so on. These files are usually placed in the root of the project and are used to configure the behavior of the tools used in the project.

Adhering to standards goes beyond naming conventions, project structure and choice of tools. It is also about using the standard solutions (and even languages) for common problems. For instance, a Python project that requires to multiply two matrices together should explore using `numpy.matmul` instead of writing its own matrix multiplication function. A Fortran project looking to diagonalize a matrix should try using `LAPACK` instead of writing its own routine.

Clear and consistent logs are another useful software engineer tool that aids in documenting a project, and where standardization is important. Logs are messages, with different levels of importance, that a program communicates informing of what the program is doing or failing to do). Error messages or warnings fall under this category, but also debugging-specific messages with detailed information about the current pathway the code is taking. Organically, some "inconveniences" will arise when dealing with logs, such as where to put them, or the ability to turn on and off certain types of messages. It is important for a project to choose a standard solution for logging. For instance, if such a project is written in Python, its readers will appreciate seeing calls to the familiar `logging` library instead of having to understand the rules of some hand made solution. Say that this hypothetical reader is attempting to fix a bug in your software. It is in your best interest to make life as easy as possible to this person by reducing their cognitive load to the minimum, lest they give up in their endeavor.

It would be strange decision to ignore the mathematical notation of your community. Your papers will require a section introducing your notation, which readers would need to process. It would be a bad idea to give a talk in a made-up language in an international conference. The effort required to understand your work would be so high that most of your peers would decide to move on. Instead, you and your peers decided to learn English. This common knowledge makes it easier to communicate.

Software is the same in all aspects. Writing a Python interface makes your project usable by other Python developers. Uploading your package to `PyPI` or `conda-forge` makes it easy to install for anyone familiar with these tools. Using GTest in C++ lowers the entry barrier for anyone wanting to write a new test. And so on and so forth.

1. *Following standards boosts interoperability*

The previous section presented, among others, the files `environment.yml` (which `conda` uses), `ci.yml` (processed by Github Actions). When presented with the decision to choose a format for their input files, the developers of these projects could had developed a custom format for it. For users, that would require to first learn the rules of this format (which also imposes an additional documentation pressure for the developers) and then learn about the available options that can be encoded in it. Moreover, interacting programatically with this custom format would entail developing extra (error prone) code specifically for encoding and decoding it.

Instead, they chose YAML, a popular human-readable plain text file format. Many users will have previous experience with the syntax of YAML, and the burden of documenting it does not fall on the developers of the project. Additionally, most programming languages have support (or de-facto standard libraries) for interacting with YAML files. At the end of the day, the choice of a standard solution (YAML) reduces the cognitive load for users and the software engineering effort for developers while making the project easier to integrate with other tools.

Advice

Use domain-specific formats
A molecular dynamics code that outputs trajectories in XTC, DCD, TRR or XYZ (among others) can automatically benefit from the postprocessing capabilities of the MDtraj project. When dealing with files in any capacity, choose a standard format instead of inventing a new one. Configuration files can use YAML, JSON or TOML, whereas HDF5 (or similar standards) is a good fit for binary data storage.

IV. CASE EXAMPLES OF GOOD SOFTWARE IN CHEMICAL PHYSICS

The practices described in this chapter are widely adopted in the scientific software community. Many projects, both large and small, follow these principles to ensure their software is reliable, maintainable, and user-friendly. Let us discuss a few examples of well-known scientific software projects that exemplify these practices, particularly in the field of chemical physics and molecular simulations.

Virtually all major molecular dynamics packages follow these practices to some extent. Examples include OpenMM[35] `https://github.com/openmm/openmm`, LAMMPS[36] `https://github.com/lammps/lammps`, UAMMD[37] `https://github.com/RaulPPelaez/uammd`, and GROMACS[38] `https://github.com/gromacs/gromacs`. These projects have all made an effort to adhere to standards and best practices in such a way that we can discuss their properties in common. When exploring any of these repositories (which we can do because they all implement version control with git), one can immediately see that they follow the conventions and standards discussed in this article. For starters, they all have a `README.md` file in the root of the repository that explains how to obtain the software and access its documentation. Moreover, all these examples have a top-level folder called `docs/`, where we can expect to find the source files for the documentation site. Users will typically be quickly redirected to an installation command from some package manager (such as pip or

conda), a link to a binary download, etc. Let us look further into these repositories from a developer's perspective (e.g. someone trying to contribute to the software).

Owning to the domain of molecular simulations, which require the most performance possible, these particular projects are written in C or C++, sometimes with bindings to higher-level languages like Python or Julia. This is made evident by the presence of a file called `CMakeLists.txt` in the root of the repository, which indicates that CMake is used as the build system and signals that one can, in principle, build the software with the standard CMake commands (`cmake -B build && cmake -build build`). The user can make this guess before even looking at the `README` file, which, on the other hand, include a link to a documentation site that explains in detail how to build the software.

All of them have a top-level folder called `tests/` (except in LAMMPS, in which it is called `unittest/`). This signals to the user that, a priori, the project has a test suite that can be run to check its correctness. The presence of a `CMakeLists.txt` file inside the test folder suggests that the test suite is integrated into the CMake build system, so that it can be run with a command like `cmake -build build -target test` or `ctest`.

Similarly, all of these projects have a folder called `.github/workflows`, which tells us that they have some kind of automatic continuous integration scripts set up. In all these cases, the CI scripts will try to build the software and run the tests on every git commit pushed to the repository, ensuring that the code is always in a working state. See, for instance, this snippet from the file `.github/workflows/build_cmake.yml` in the GROMACS repository, which is set up to run on every new commit or pull request:

```
- name: Configure
  run: cmake -P
   ↪  .github/scripts/configure.cmake

- name: Build
  run: cmake -P
   ↪  .github/scripts/build.cmake

- name: Run tests
  run: cmake -P
   ↪  .github/scripts/test.cmake
```

By inspecting the scripts in `.github/scripts/`, the user can see an example on how to build and test the software, which (besides the benefits in ensuring correctness) is useful as a tool for discoverability.

Being aware of the standard tools and practices in scientific software development allow us to gather an impressive amount of information about a project with just a quick glance at its project structure, even before reading any documentation. Furthermore, adhering to these standards makes it easier for others to contribute to the project, as they can quickly understand how to build, test, and learn how to extend the software.

Let us now look at the domain of AI-enhanced quantum chemistry. In this field it is usual to write software in Python, making use of popular machine learning frameworks such as Pytorch[39] or JAX[40]. Examples include TorchMD-Net[34], SchNetPack[41], and NequIP[42]. Let us focus on the recent FeNNol[43] library, hosted at `https://github.com/thomasple/FeNNol`, which deals with the development of neural network potentials for molecular simulations[44]. For users, the first lines of the `README.md` file explain how to install the software using `pip`, and provide a link to the documentation site. From a developer's point of view, the presence of a file called `pyproject.toml` in the root of the repository signals that this is a Python that can be installed using `pip install .`. Furthermore, we can guess that any external dependency will also be handled by pip automatically. There is also a folder called `tests/`, containing files with a `test_` prefix, which suggests that the tests can be run with `pytest`. Finally, inspecting the filenames under `.github/workflows` reveals that the project automatically builds and publishes the documentation site on every new release, as well as uploading a new version of the package to the Python Package Index (PyPI) for distribution.

The software produced in each scientific community will have its own particularities, such as the choice of programming language, build system, distribution method, and so on. Additionally, each repository will have its own unique deviations from the standard way of presenting itself. However, if the authors took care to follow good practices, we will be able to quickly identify the main components of the project, such as how to install it, how to run the tests, how to

build the documentation, and so on. This will make it easier for users to get started with the software, and for developers to contribute to it.

V. LARGE LANGUAGE MODELS (LLMS) IN SCIENTIFIC SOFTWARE

Large Language Models are a type of artificial intelligence that has shown remarkable capabilities in understanding and generating human-like text. They are trained on vast amounts of text data and can perform a wide range of tasks, including code generation, natural language understanding, and more. With the advent of LLMs, which have proven to be powerful code generators[28], the community is currently undergoing a boom of research into artificial-intelligence-assisted coding. LLMs are a game-changer in the field of scientific software, elevating the abilities of developers and, if used correctly, dramatically increasing their productivity. They are, though, double-edged swords for a number of reasons that will be covered in this section. Given the novel and fast-changing nature of the field of large language models in software development, this section is bound to be highly speculative, opinionated, and perhaps quickly made obsolete. It is, however, indispensable to address here the current role of LLMs in scientific software. There are plenty of works exploring the role of LLMs in different aspects of the software development cycle. The authors of[45,46] discuss the role of LLMs in software engineering, while[47,48] focus specifically on scientific software development.

LLMs are good at answering questions, but they will not protect you from asking the wrong questions. As a simplified example, imagine the following prompt a novice might send to an LLM:

> I have been working on a code for the past months on my laptop, yesterday my laptop was stolen and I have lost it all. What can I do to ensure that this does not happen again?

The right answer to this question is guiding the user through version control and repository hosting providers. With this prompt, though, the LLM has several other avenues to answer satisfactorily. For instance, it can suggest a backup solution (like Dropbox). It might also understand the question subtly incorrectly and suggest implementing a self-destruct functionality, or paying for insurance, or buying a bag with a lock for the next laptop. If the user is lucky, the LLM will also mention version control as an alternative, alongside the rest of the possibilities.

These are all *good* answers, and some of them should be implemented also in general, but they are not what the user needed to know because the question was ill-formed or incomplete. By construction, the user lacks the knowledge and intuition to choose one of these over the rest. This chicken-and-egg situation does not have an easy solution that does not involve learning and becoming familiar with the field, be it software engineering, coding, or the use of a particular library or framework. One has to be aware of this kind of bias when trying to learn from an LLM.

As a more practical and focused example, to the prompt "I want to import one of my Python scripts from within another one of my Python scripts, which is located somewhere different from the first" an LLM will almost invariably suggest something similar to the following code (generated by ChatGPT 4o on June 18, 2025):

```
# In script2.py
import sys
import os

# Add the directory containing your
↪  other script
script_dir = os.path.abspath("/path/to
↪  /other/script")
if script_dir not in sys.path:
  sys.path.insert(0, script_dir)

# Now you can import the script
import script1

# Call functions from my_script1
script1.some_function()
```

This is a terribly misleading and hurtful answer for a novice. It will signal that the way to import a script is to modify the `sys.path` variable, which is not only unnecessary, but also dangerous. This is because it will make the code dependent on the absolute path of the script, which is not portable and will break if the script is moved to a different location or if the script runs on a different machine. Instead, a better answer would involve explaining that a healthy project structure should be

adopted, and the projects containing the scripts should be installed as packages so that they can be imported from anywhere. Current conversational LLMs (even the ones finetuned for coding) are designed to be complacent and provide quick and satisfactory answers, which is not always what the user needs. They will seldom explain that the question is ill-posed. Ironically, one of the main complaints the community used to have about StackOverflow (`https://stackoverflow.com`), the main Q&A site for programmers, was the fact that answers tended to include one similar to "Why would you want to do that?", and which were perceived as rude and unhelpful. Another issue with current coding LLMs is also evidenced in the snippet generated above. The mannerisms adopted by these chatbots when producing code (likely a byproduct of the datasets that were used to fine-tune them) are not examples of healthy and robust code. While often these snippets are helpful and informative, they should not be placed verbatim into a larger codebase. For instance, and this is not particular to a specific language or task, almost invariably the code produced in these prompts is structured as:

- A white line
- A redundant comment
- An actual line of code

Most reputable coding style resources will explain the dangers of such spurious comments, which not only increase the cognitive load of the code but are bound to rot when the line of code below it changes, making the comment then unrelated to the actual logic. Of course, the true issue in this case is educational, as it is signaling to a novice that this structure is the way to go. Granted, all these quirks can be prompted away. However, we enter into the circular argument that someone with the expertise to recognize and amend these issues perhaps did not need the LLM to solve the issue in the first place. A person who has been taught how to produce software by such an LLM will be hindered from becoming the expert they need to be in order to detect its quirks.

A possible counterargument is that conversational LLMs are not the only resources for LLM-assisted software development. Recently, agentic systems and LLM-enhanced code editors have gained popularity in this regard. While these certainly help address some of the aforementioned shortcomings of LLM-generated code, they bring other concerns into the picture. The term *vibe-coding* has arisen to describe an approach to software development that relies heavily on LLMs to generate code based on natural language prompts. Vibe coding promises to be the ultimate declarative paradigm, in which the programmer describes *what* the software should do, without being concerned with the how. One issue with this paradigm comes from the degree of control that the developer delegates to the generative tools. Each time the model intervenes, it is inevitably adding complexity to our project while at the same time reducing the developers' awareness and knowledge of the project. This does not mean these tools are to be avoided; rather, a word of caution is that their usage must be accompanied by a highly increased effort to mitigate and prevent complexity. That entails following good software engineering practices, such as writing a solid test battery, focusing on effective design of software and interfaces, and so on and so forth. LLMs excel at coding when used for boilerplate or repetitive code. As well as the implementation of small and constrained components, such as functions for which we have already written a signature and documentation. The LLM can also aid us in producing the signature and docstring, but it is important that this process is directed by the developer in order for it to be effective. Even so, latest studies suggest that AI tools might even be detrimental to productivity when measuring completion time[49,50].

VI. CONCLUSIONS

Scientific software development is inherently challenging due to the unbounded growth in complexity that naturally accompanies evolving software projects. However, these challenges can be effectively managed through adopting practical software engineering techniques and leveraging well-established community standards. By incrementally embracing standardized practices, scientists can greatly enhance the maintainability, reproducibility, and longevity of their software projects.

Throughout this article, we emphasized that software complexity is inevitable and that actively addressing it from the project's inception is essential. Early investment in adopting these techniques significantly reduces long-term maintenance costs and facilitates collabo-

ration and transparency within scientific communities. Moreover, adherence to widely recognized standards makes software predictable, approachable, and seamlessly integrable with existing tools and workflows.

The role of Large Language Models (LLMs) in scientific software development has also been examined critically, highlighting their substantial potential alongside their inherent limitations. While LLMs offer tremendous opportunities to streamline software development, their usage must be informed, cautious, and accompanied by rigorous testing, thoughtful design, and careful oversight to mitigate the introduction of unnecessary complexity or misleading guidance.

Ultimately, scientists who write software are encouraged to seek out and adopt established solutions provided by the broader software engineering community. Institutions and senior scientists should proactively foster environments that prioritize and reward software engineering best practices and invest in training and resources that equip researchers with essential skills.

Several important practices remain outside this article, such as conventional commits, git hooks, pull-requests and code reviews, code linters and formatters, and licensing. Nevertheless, awareness and gradual incorporation of these additional elements can further improve the quality and sustainability of scientific software. By continuously striving to detect and align with industry-standard practices, scientists can create software that transcends the solving of immediate research problems to become a robust foundation for future scientific endeavors.

VII. ACKNOWLEDGEMENTS

The author thanks Professor Brennan Sprinkle and Ryker Fish for their valuable feedback on this article. The author also extends gratitude to Professor Aleksandar Donev, whose guidance initiated the journey that inspired the exploration of the practices described herein.

Partial funding from the FASTCOMET project (HORIZON-EIC-2023-PATHFINDEROPEN-01 proposal 101130615) is acknowledged.

[1] M. M. Lehman. Programs, Cities, Students- Limits to Growth? In David Gries, editor, *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, pages 42–69. Springer, New York, NY, 1978. ISBN 978-1-4612-6315-9. doi:10.1007/978-1-4612-6315-9_6. URL https://doi.org/10.1007/978-1-4612-6315-9_6.

[2] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980. ISSN 1558-2256. doi:10.1109/PROC.1980.11805. URL https://ieeexplore.ieee.org/document/1456074. Conference Name: Proceedings of the IEEE.

[3] LA Belady and MM Lehman. *The characteristics of large systems*. IBM Thomas J. Watson Research Division, 1977.

[4] R: The R Project for Statistical Computing. URL https://www.r-project.org/.

[5] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[6] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Addison-Wesley, 3rd edition, April 2024. ISBN 978-0-13-830868-1.

[7] The MathWorks Inc. Matlab version: 9.13.0 (r2022b), 2022. URL https://www.mathworks.com.

[8] L. Torvalds and D. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperCollins, 2002. ISBN 9780066620732. URL https://books.google.es/books?id=6zSWd8Ou8BAC.

[9] D. Thomas and A. Hunt. *The Pragmatic Programmer: Your journey to mastery, 20th Anniversary Edition*. Pearson Education, 2019. ISBN 9780135956915. URL https://books.google.es/books?id=LhOlDwAAQBAJ.

[10] J.K. Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018. ISBN 9781732102200. URL https://books.google.es/books?id=pD6-swEACAAJ.

[11] A. Brown and G. Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. The Achrictecture of Open Source Applications. Creative Commons, 2011. ISBN 978-1-257-63801-7. URL https://books.google.es/books?id=pgI1AwAAQBAJ.

[12] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. Good enough practices in scientific computing. *PLoS Computational Biology*, 13(6):e1005510, 2017. doi:10.1371/journal.pcbi.1005510. URL https://doi.org/10.1371/journal.pcbi.1005510.

[13] Amy Brown and Greg Wilson, editors. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, volume 1. Lulu.com, 2011. ISBN 978-1257638017. URL https://aosabook.org/en/index.html. Open-access edited collection; Volume I of AOSA.

[14] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Pearson Education, 1995. ISBN 9780132119160. URL https://books.google.es/books?id=Yq35BY5Fk3gC.

[15] The conda-forge Community. conda-forge: A community-led collection of recipes, build infrastructure and distributions for the conda package manager. https://conda-forge.org, 2024. Accessed 20 July 2025.

[16] Flickr founder plans to kill company e-mails with slack. URL https://www.cnet.com/tech/tech-industry/flickr-founder-plans-to-kill-company-e-mails-with-slack/.

[17]Zoom. URL https://zoom.us/.

[18]Brian Foote and Joseph Yoder. Big ball of mud. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, Software Patterns Series, chapter 29. Addison-Wesley, 2000. Originally presented at PLoP '97/EuroPLoP '97.

[19]Review criteria, journal of open source software (joss). https://joss.readthedocs.io/en/latest/review_criteria.html, 2025. Accessed: 2025-09-26.

[20]Author guidelines, biophysical journal. https://www.cell.com/pb-assets/journals/society/biophysj/PDFs/author-guidelines.pdf, 2025. Accessed: 2025-09-26.

[21]Linus Torvalds and Junio C. Hamano. The git version control system. https://git-scm.com.

[22]Scott Chacon and Ben Straub. Pro git. https://git-scm.com/book/en/v2, n.d. URL https://git-scm.com/book/en/v2.

[23]Visual studio code. URL https://code.visualstudio.com/.

[24]The Python Packaging Authority. pip: The python package installer. https://pip.pypa.io/, 2023. Version 23.0.1. Accessed 20 July 2025.

[25]Raul P. Pelaez. superpunto: v5.2.1, July 2025. URL https://doi.org/10.5281/zenodo.16206804.

[26]Raul P. Pelaez. spreadinterp: v1.0.0, July 2025. URL https://doi.org/10.5281/zenodo.16206301.

[27]Kent Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003. ISBN 978-0-321-14653-3.

[28]Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications, 2025. URL https://arxiv.org/abs/2503.01245.

[29]Raul P. Pelaez. Meansquaredisplacement: 3.0.2, July 2025. URL https://doi.org/10.5281/zenodo.16206359.

[30]Conda contributors. conda: A system-level, binary package and environment manager running on all major operating systems and platforms. https://github.com/conda/conda. URL https://docs.conda.io/projects/conda/.

[31]Microsoft Corporation. vcpkg: A c++ package manager. https://github.com/microsoft/vcpkg, 2024. Accessed 20 July 2025.

[32]JFrog Ltd. Conan: The c/c++ package manager. https://conan.io, 2024. Accessed 20 July 2025.

[33]Kitware, Inc. *CMake: Build System Generator*. Kitware, Inc., 2024. URL https://cmake.org. Version 3.29.2, Accessed 20 July 2025.

[34]Raul P. Pelaez, Guillem Simeon, Raimondas Galvelis, Antonio Mirarchi, Peter Eastman, Stefan Doerr, Philipp Thölke, Thomas E. Markland, and Gianni De Fabritiis. Torchmd-net 2.0: Fast neural network potentials for molecular simulations. *Journal of Chemical Theory and Computation*, 20 (10):4076–4087, 2024. doi:10.1021/acs.jctc.4c00253. URL https://doi.org/10.1021/acs.jctc.4c00253. PMID: 38743033.

[35]Peter Eastman, Raimondas Galvelis, Raúl P. Peláez, Charlles R. A. Abreu, Stephen E. Farr, Emilio Gallicchio, Anton Gorenko, Michael M. Henry, Frank Hu, Jing Huang, Andreas Krämer, Julien Michel, Joshua A. Mitchell, Vijay S. Pande, João PGLM Rodrigues, Jaime Rodriguez-Guerra, Andrew C. Simmonett, Sukrit Singh, Jason Swails, Philip Turner, Yuanqing Wang, Ivy Zhang, John D. Chodera, Gianni De Fabritiis, and Thomas E. Markland. Openmm 8: Molecular dynamics simulation with machine learning potentials. *The Journal of Physical Chemistry B*, 128(1):109–116, 2024. doi: 10.1021/acs.jpcb.3c06662. URL https://doi.org/10.1021/acs.jpcb.3c06662. PMID: 38154096.

[36]Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, February 2022. ISSN 0010-4655. doi:10.1016/j.cpc.2021.108171.

[37]Raúl P. Peláez, Pablo Ibáñez-Freire, Pablo Palacios-Alonso, Aleksandar Donev, and Rafael Delgado-Buscalioni. Universally adaptable multiscale molecular dynamics (uammd). a native-gpu software ecosystem for complex fluids, soft matter, and beyond. *Computer Physics Communications*, 306:109363, 2025. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2024.109363.

[38]Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1–2: 19–25, September 2015. ISSN 2352-7110. doi: 10.1016/j.softx.2015.06.001.

[39]Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[40]James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

[41]Kristof T. Schütt, Stefaan S. P. Hessmann, Niklas W. A. Gebauer, Jonas Lederer, and Michael Gastegger. Schnetpack 2.0: A neural network toolbox for atomistic machine learning. *The Journal of Chemical Physics*, 158(14), April 2023. ISSN 1089-7690. doi:10.1063/5.0138367. URL http://dx.doi.org/10.1063/5.0138367.

[42]Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature Communications*, 13(1), May 2022. doi:

10.1038/s41467-022-29939-5. URL https://doi.org/10.1038/s41467-022-29939-5.

[43] Thomas Plé, Olivier Adjoua, Louis Lagardère, and Jean-Philip Piquemal. Fennol: An efficient and flexible library for building force-field-enhanced neural network potentials. *The Journal of Chemical Physics*, 161(4):042502, July 2024. ISSN 0021-9606. doi: 10.1063/5.0217688.

[44] Timothy T. Duignan. The potential of neural network potentials. *ACS Physical Chemistry Au*, 4(3):232–241, May 2024. doi: 10.1021/acsphyschemau.4c00004.

[45] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, Dec 2024. ISSN 1573-7616. doi: 10.1007/s10664-024-10602-0. URL https://doi.org/10.1007/s10664-024-10602-0.

[46] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study, 2024. URL https://arxiv.org/abs/2403.08604.

[47] Akash Dhruv and Anshu Dubey. Leveraging large language models for code translation and software development in scientific computing, 2025. URL https://arxiv.org/abs/2410.24119.

[48] Mohamed Nejjar, Luca Zacharias, Fabian Stiehle, and Ingo Weber. Llms for science: Usage for code generation and data analysis, 2024. URL https://arxiv.org/abs/2311.16733.

[49] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity, 2025. URL https://arxiv.org/abs/2507.09089.

[50] June Sallou, Thomas Durieux, and Annibale Panichella. Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 102–106. ACM, April 2024. doi:10.1145/3639476.3639764. URL http://dx.doi.org/10.1145/3639476.3639764.