Tools for (better) computational biology

This manuscript (permalink) was automatically generated from computer-aided-biotech/better-cb@8ec50ea on April 20, 2022.

Authors

- **Daniela C. Soto** *
  0000-0002-6292-655X · dcsoto · dcsoto_cl
  Genome Center, MIND Institute, and Department of Biochemistry & Molecular Medicine, Davis, CA 95616,USA

- **Benjamín J. Sánchez** *
  0000-0001-6093-4110 · BenjaSanchez · BenjaSanchez
  Department of Bioengineering, Technical University of Denmark, Kgs. Lyngby, 2800, Denmark

- **Megan Y. Dennis**
  · meganamsu · meganamsu
  Genome Center, MIND Institute, and Department of Biochemistry & Molecular Medicine, Davis, CA 95616,USA

- **Nikolaus Sonnenschein**
  · phantomas1234 · phantomas1234
  Department of Bioengineering, Technical University of Denmark, Kgs. Lyngby, 2800, Denmark

Abstract

As biotechnological and biomedical research are increasingly fed by the insights arising from computation, the conversation about good practices in computational biology becomes more and more prominent. An increasing body of literature has addressed practices for shareable, reproducible, and sustainable computational research, from high-level principles for data and software stewardship to deep dives into version control or software automation. However, implementing these practices relies on incorporating the right tools into our daily routines, considering the type, scope, and stage of the research project. Here we provide a compendium of relevant tools for computational biology research, emphasizing their time and place within a continuum that traverses personal, collaborative, and community practices. This compendium will serve as a starting point and guide to help navigate the ongoing influx of tools and how to best incorporate them into a computational biologist's working routine, enabling reproducible biomedical and biotechnological research in the long term.

Introduction

Since Margaret Dayhoff pioneered the field of bioinformatics in the sixties, the application of computational tools in the field of biology has vastly grown in scope and impact. At present, biotechnological and biomedical research are routinely fed by the insights arising from novel computational approaches, machine learning algorithms, and mathematical models. The ever-increasing amount of biological data and the exponential growth in computing power will amplify this trend in the years to come.

The use of computing to address biological questions encompasses a wide array of applications usually grouped under the terms "computational biology" and "bioinformatics." Although distinct definitions have been delineated for each one [1,2], here we will consider both under the umbrella term "computational biology," alluding to any application that involves the intersection of computing and biological data. As such, a computational biologist can be a data analyst, a data engineer, a statistician, a mathematical modeler, a software developer, and many other roles. In praxis, the modern computational biologist will be a "scientist of many hats," taking on several of the duties listed above. But first and foremost, we will consider a computational biologist as a scientist whose ultimate goal is to answer a biological question or address a need in the life sciences by means of computation.

Scientific computing requires following specific principles to enable shareable, reproducible, and sustainable outputs. Computing-heavy disciplines, such as software engineering and business analytics, have adopted protocols addressing the need for collaboration, visualization, project management, and strengthening of online communities. However, as a highly interdisciplinary and evolving field, computational biology has yet to acquire a set of universal "best practices." Since most computational biologists come from diverse backgrounds and rely on self-study rather than formal education [3], the absence of guidelines may lead many computational biologists astray, using methods that hinder reproducibility and collaboration, such as unreproducible computational workflows or closed-source software, retarding biomedical and biotechnological research.

In recent years, this "guidelines gap" has been addressed by the establishment of FAIR principles—Findability, Accessibility, Interoperability, and Reusability—in 2016 [4]. Originally developed for data stewardship, FAIR principles have been proposed as universal guidelines for all research-related outputs [5]. However, translating these high-level principles into day-to-day practices requires additional nuances based on the type of research, the size and scope of the project, and the researcher's experience. To address the need for FAIR scientific software, for example, the framework ADVerTS (availability of software, documenting software, version control, testing, and support) has been proposed as a set of "barely sufficient" practices [5]. More broadly, reviews exist covering

general topics for bench scientists new to computational biology—such as programming and project organization [6,7,8,9,10]—to detailed descriptions for the more seasoned data scientist—such as workflow automation [11], software library development [12], software version control with the cloud service GitHub [13], and interactive data science notebooks with Jupyter [14].

Although the above reviews are immensely helpful, an overview of tools for better computational biology is missing. Indeed, guiding principles and general advice are key to establishing a behavior roadmap but their implementation is enabled by incorporating the right tools into our daily working routine. Tool selection has many components, such as availability, suitability, and personal preference; although the latter is left to the reader, here we will shed light on the first two. We premise that good practices in computational biology lie within a continuum that traverses three levels: personal (you), collaboration (your group), and community (your field) (Figure 1). Each of these levels has a different set of requirements and challenges, as well as a specific set of tools that can be used to address them. Here, we compiled a curated list of these tools, emphasizing their time and place in a computational biology research project. Committed to practicality, we illustrated the utility of these tools in case studies covering a wide spectrum of research topics that computational biologists can use to model their own practices, modifying them to suit their own needs and preferences.

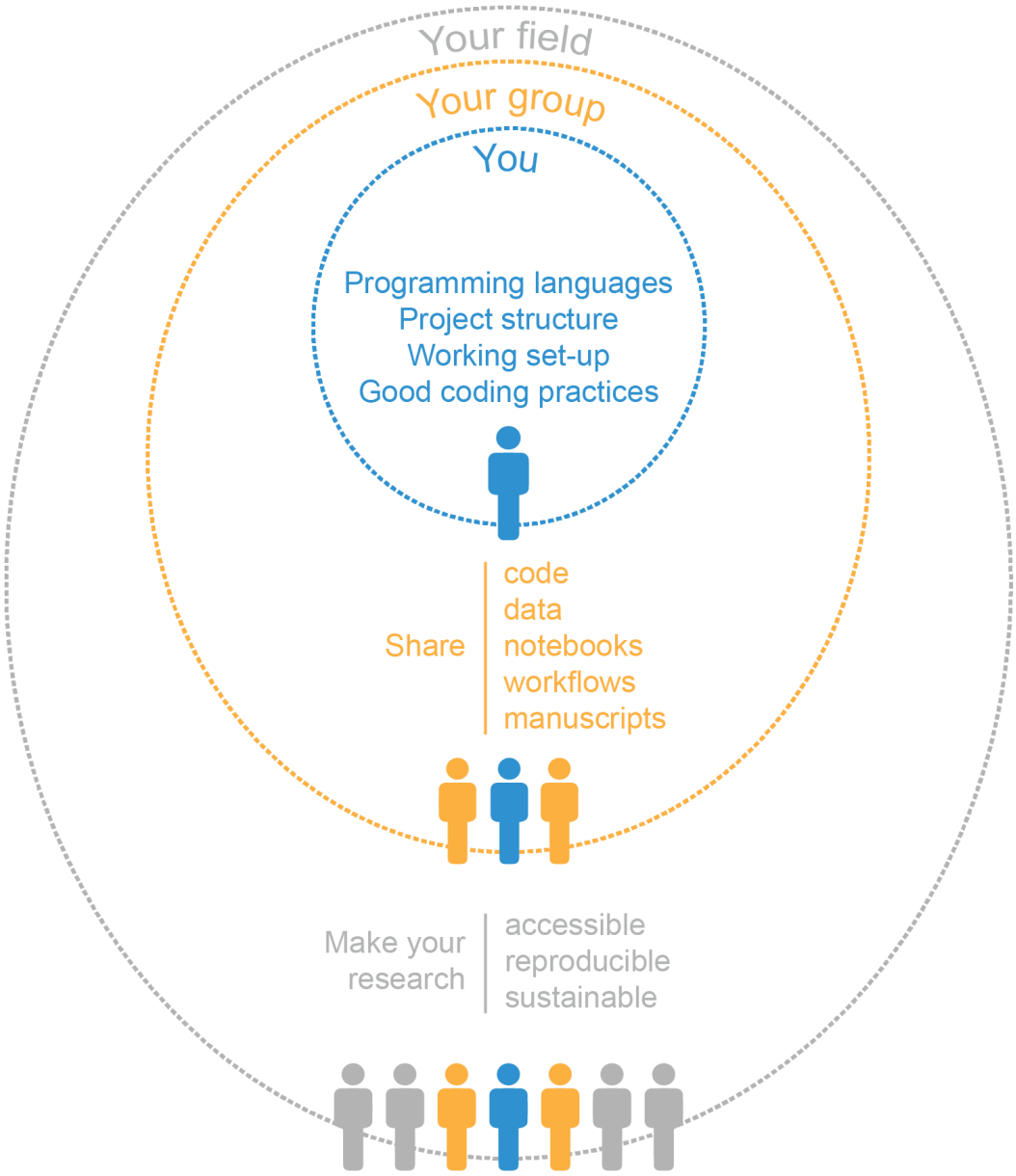

Figure 1: The three "levels" of computational biology include your personal research, your group and collaborators, and the broader scientific community.

Level 1: Personal Research

The computational biology journey begins with you and the set of skills, tools, and practices that you have in place to conduct your research. Taking the time to optimally establish these building blocks will have high payoffs later when you find yourself going back to previous analyses. Consider that your most important collaborator is your future self, be it tomorrow or several years from now. We devised a framework involving four main sequential steps to kickstart any computational biology project (Table 1).

Table 1: Steps involved in starting a computational biology project.

Step	Use case	Common tools
Step 1: Choose your programming languages	Interacting with a Unix/Linux HPC	• **Shell/Bash** [15]
	Data analysis	• **Python** [16], **R** [17]
	Scripts and programs	• *Interpreted*: **Python** [16], **R** [18], **Perl** [19], **MATLAB< [20], Julia< [21]** • ***Compiled*: C/C++ [22], Rust [23]**
	Workflows	• *Linux-based*: **shell script**, **GNU Make** [24] • *Workflow management systems*: **Snakemake** (Python) [25], **Nextflow** (Groovy) [26], • *Worflow specifications:* **CWL** [27], **WDL** [28]
Step 2: Define your project structure	Project structure	• *Templates*: **Cookiecutter Data Science** [29], **rr-init** [30] • *Workflows*: **Snakemake worflow template** [31]
	Virtual environment managers	• *Language-specific*: **virtualenv** (Python) [32], **renv** (R) [33] • *Language agnostic*: **Conda** [34]
	Package managers	• *Language-specific*: **pip** (Python) [35], **Bioconductor** (R) [36], **R Studio package manager** (R) [37] • *Language-agnostic*: **Conda** [34]
Step 3: Choosing your working set-up	Text editors	• *Desktop applications*: **Atom** [38], **Sublime** [39], **Visual Studio Code** [40], **Notepad++** [41] • *Command line*: **Vim**[42], **GNU Emacs**[43]
	IDEs	• *For Python*: **JupyterLab** [44], **JetBrains/PyCharm** [45], **Spyder** [46] • *For R*: **R Studio** [47]
	Notebooks	• **Jupyter** (Python, R) [44], **R Markdown** (R) [48]
Step 4: Follow good coding practices	Coding style	• *Styling guides:* **PEP-8** (Python) [49], **Google** (Python, R) [50] • *Automatic code formatting:* **Black** (Python) [51], **Snakefmt** (Snakemake) [52]
	Literate programming	• **Markdown** [53] • **R Markdown** [48]
	Version control	• *Version control system:* **Git** [54] • *Code repositories:* **GitHub** [55], **GitLab** [56], **Bitbucket** [57] • *Git GUIs:* **GitHub Desktop** [58], **GitKraken** [59]

Step 1: Choose your programming languages

Different programming languages serve distinctive purposes and have unique idiosyncrasies. As such, choosing a programming language for a specific project depends on your research goals, personal

preferences, and skillsets. Additionally, communities usually favor the usage and training of some programming languages over others; utilizing such languages may facilitate integrating your work within the existing ecosystem.

Interacting with high-performance computing (HPC) clusters has become a hallmark for the data-intensive discipline of computational biology. HPC infrastructures commonly use Unix/Linux distributions as their operating system. To interact with these platforms, a command-line interpreter known as the shell must be used. There are multiple versions of shells, with Bash [15] being one of the most widely adopted. In addition to providing an interface, the shell is also a scripting language that allows manipulating files and executing programs through shell scripts. Unix/Linux operating systems have other interesting perks, such as powerful, fast commands for searching and manipulating files (e.g., sed, grep, or join) as well as the language AWK, which can perform quick text processing and arithmetic operations.

One of the most common tasks of any computational biologist is data analysis, which usually involves data cleaning, exploration, manipulation, and visualization. Currently, Python [16] is the most widely used programming language for data analysis [60,61]. Python is also a popular language among computational biologists, a trend that will likely continue as machine learning and deep learning are more widely adopted in biological research. Python usage has been facilitated by the availability of packages for biological data analysis accessible through package managers such as pip [35] or Conda [34]. Likewise, R [18] is another prominent language in the field. Arguably, one of the main strengths of R is its wide array of tools for statistical analysis. Of particular interest is the Bioconductor repository [36], where many gold-standard tools for biological data analysis have been published and can be installed using BiocManager [62]. R usage in data science has deeply benefited from the Tidyverse packages [63] and surrounding community, increasing the readability of the R syntax for both data manipulation via dplyr and visualization via ggplot2.

Computational biologists often must code their own sets of instructions for processing data using scripts or tools. In computational biology, a script often refers to a lightweight single-file program written in an interpreted programming language and developed to perform a specific task. Scripts are quick to edit and can be run interactively but at the expense of computational performance. To automate instructions in HPC clusters, shell scripts are commonly used. For other purposes, the most widely used scripting languages are Python [16] and R [18], but Perl [19], MATLAB [20], and Julia [21] are preferred by some researchers for bioinformatics, systems biology, and statistics, respectively. A computational biology tool, on the other hand, is a more complex program designed to tackle computationally intensive problems like developing new algorithms. Several tools devised for data-intensive biology have been written in compiled languages such as C/C++ [22]. In recent years, however, scientists have been turning to Rust [23] due to its speed, memory safety, and active community [64]. When computational performance is less of a concern, Python and R are suitable alternatives for computational biology tool development.

Biological data processing is rarely a one-step process. To go from raw data to useful insights, several steps need to be taken in a specific order, accompanied by a plethora of decisions regarding parameters. Computational biologists have addressed this need by embracing workflow management systems to automate data analysis pipelines. A pipeline can be a shell script where commands are written sequentially, using shell variables and scripting syntax when needed. Although effective, this approach provides little control over the workflow and lacks features to run isolated parts of the pipeline or track changes of input and output files. To overcome these limitations, a shell script can be upgraded using the GNU Make [24] program, which was originally designed to automate compilation and installation of software, but is flexible enough to build workflows. More sophisticated bioinformatics workflow managers have also been developed such as Snakemake [25] based on Python and Nextflow [26] based on Groovy (a programming language for the Java virtual machine). These tools offer support for environment managers and software containers (discussed in Level 3),

as well as allow for easy scaling of pipelines to both traditional HPC and modern cloud environments. Alternatively, there are available declarative standards to define workflows in a portable and human-readable manner such as the Common Workflow Language (CWL) [27] and Workflow Description Language (WDL, pronounced "widdle") [28], used by the cloud computing platform AnVIL [65,66]. Although these are not executable, they can be run in CWL- or DWL-enabled engines such as Cromwell [67].

Step 2: Define your project structure

The next step after choosing your programming languages but before starting coding is to develop an organized project structure. The project design should be intentional and tailored to the present and future needs of your project—remember to be kind to your future self! A computational biology project requires, at the very least, a folder structure that supports code, data, and documentation. Although tempting, cramming various file types into one unique folder is unsustainable. Instead, separate files into different folders and subfolders, if needed. To simplify this process, base your project structure on research templates available off-the-rack. For data science projects, the Python package Cookiecutter Data Science [29] decreases the effort to minimal. Running the package prompts a questionnaire in the terminal where you can input the project name, authors, and other basic information. Then, the program generates a folder structure to store data—raw and processed—separate from notebooks and source code, as well as pre-made files for documentation such as a readme, a docs folder, and a license. Similarly, the Reproducible Research Project Initialization (rr-init) offers a template folder structure that can be cloned from a GitHub repository and modified by the user [30]. Although rr-init is slightly simpler, both follow an akin philosophy aimed at research correctness and reproducibility [68]. For workflow automation projects, we advise following the Snakemake workflow template [31,69], storing each workflow in a dedicated folder divided into subfolders for workflow-related files, results, and configuration. In all cases, the folder must be initialized as a git repo for version control (see Step 4).

The software and dependencies needed to execute a tool or workflow are also part of the project structure itself. The intricacies of software installation and dependency management should not be underestimated. Fortunately, package and virtual environment managers significantly reduce this burden. A package manager is a system that automates the installation, upgrading, configuration, and removal of community-developed programs. A virtual environment manager is a tool that generates isolated environments where programs and dependencies are installed independently from other environments or the default operating system. Once a virtual environment is activated, a package manager can be used to install third-party programs. We believe that every computational biology project must start with its own virtual environment to boost reproducibility: environments save the project's dependencies and can restore them at will so the code can be run on any other computer. There are multiple options for both package and virtual environment management—some language-specific and some language-agnostic. If you are working with Python, you can initialize a Python environment using virtualenv [32] (where different Python versions can be installed). Inside the environment, you can use the Python package manager pip [35] to import Python code from the Python Package Index (PyPI) repository, GitHub, or locally. For the R language, R-specific environments can be created using renv [33], where packages can be installed via the install.packages function from the Comprehensive R Archive Network (CRAN) and CRAN-like repositories. R also has BiocManager to install packages from the Bioconductor repository, which contains relevant software for high-throughput genomic sequencing analysis. Additionally, RStudio Package Manager [37] works with third-party code available in CRAN, Bioconductor, GitHub, or locally. Conda [34]—a language-agnostic alternative—supports program installation from the Anaconda repository, which contains the channel Bioconda [70] specifically tailored to bioinformatics software. Python dependencies can also be installed via pip inside a Conda environment. Conda is particularly helpful when working with third-party code in various languages—a common predicament in computational biology. The Conda package and environment manager is included in both the Anaconda and Miniconda distributions.

The latter is a minimal version of Anaconda, containing only Conda, Python, and a few useful packages.

Step 3: Choose your working set-up

Before coding, a more practical question needs to be answered first: Where to code? The simplest tools available for this purpose are text editors. Since writing code is ultimately writing text, any tool where characters can be typed fulfills this purpose. However, coding can be streamlined by additional features—including syntax highlight, indentation, and auto-completion—available in code editors such as Atom [38], Sublime [39], Visual Studio Code [40], and Notepad++ [41] (Windows only). Command-line text editors such as Vim [42] and Emacs [43] are also suitable options for coding. These tools share the advantage of being language agnostic, which is handy for the polyglot computational biologist.

In addition to text editors, integrated development environments (IDEs) are also popular options for coding. In their essence, IDEs are supercharged text editors comprising a code editor (with syntax highlight, indentation, and suggestions), a debugger, a folder structure, and a way to execute your code (a compiler or interpreter). Some IDEs are not language-agnostic, often only allowing code in one language. The array of features also comes at a cost—IDEs typically use more memory. For Python, Jupyter Lab [44], Spyder [46], and PyCharm [45] are popular options, while for R, RStudio [47] is the gold standard. Notably, the differences between an IDE and a code editor are somewhat blurry, particularly when employing plugins with a code editor.

In recent years, notebooks have acquired relevance in computational biology research. A notebook is an interactive application that combines live code (read-print-eval loop or REPL), narrative, equations, and visualizations, internally stored using a format called JavaScript Object Notation (JSON). Common notebooks use interpreted languages such as Python or R, and narrative usually uses Markdown—a lightweight markup language. Data analysis greatly benefits from using notebooks instead of plain text editors or even IDEs. The combination of visuals and texts allows researchers to tell compelling stories about their data, and the interactivity of its code enables quick testing of different strategies. Jupyter [44] is a popular web-based interactive notebook developed originally for Python coding but also accepts R and other programming languages upon installation of their kernels—the computing engine that executes the notebook's live code under the hood. Jupyter notebook can also be executed in the cloud using platforms such as Google CoLaboratory (CoLab) [71] and Amazon WebServices, taking advantage of the current trend of cloud computing. In addition, RStudio allows the generation of R-based notebooks known as R Markdown [48], which is especially well suited for generating data analysis reports.

Step 4: Follow good coding practices

With the foundation in place, the next step is to start writing code. Coding, however, requires good practices to ensure correctness, sustainability, and reproducibility for you, your future self, your collaborators, and the whole community. First and foremost, you need to make sure your code works correctly. In computational biology, correctness implies biological and statistical soundness. Although both are topics beyond the scope of this manuscript, a useful approach to evaluate biological correctness is to design positive and negative controls in your program, analysis, or workflow. In scientific experimentation, a positive control is a control group that is expected to produce results; a negative control is expected to produce no results. The same approach can be applied to computation, using input data whose output is previously known. Biological soundness can also be tested by quickly assessing expected orders of magnitude in both intermediate and final files. These checks can be packaged in unit testing (discussed in Level 2).

In addition to correctly functioning code, code appearance, also known as coding style, is important. Code style includes a series of small, ubiquitous decisions regarding where and how to add comments; indentation and white-space usage; variable, function, and class naming; and overall code organization. Although, as in writing, personality and preference differences dictate how you code, coding style rules facilitate collaboration with your future self and others. Indeed, as we sometimes have trouble reading our own handwriting, we can also struggle reading our own code if we disregard guidelines. At the very least, aim to follow internal consistency in writing code. Even better, consider following any of the multiple published coding-style guides such as those from software development teams. Google, for example, has guidelines for Python, R, Shell, C++, and HTML/CSS [50]. Guidelines for Python are available as part of the Python Enhancement Proposal (PEP), known as PEP 8 [49]. To facilitate compliance, tools called linters can be incorporated into most code editors and IDEs to flag stylistic errors in your code based on a given style guide. Furthermore, many editors and tools perform automatic code formatting (e.g., Black [51] that formats Python code to be PEP 8 compliant), which can greatly facilitate stylistic coherence in a collaborative project. In the case of Snakemake files, stylistic errors can be flagged using the Snakemake linter, which can be invoked with the command snakemake –lint [72], or automatically corrected with the tool Snakefmt [52], based on Black.

On the matter of code styling, two topics merit additional attention: variable naming and comments. Variable names should be descriptive enough to convey information about the variable, function, or class content and use. The goal is to produce self-documented code that reads close to plain English. To do so, multi-word variable names should be used if necessary. In such cases, the most common conventions include Camel Case, where the second and subsequent words are capitalized (camelCase); Pascal Case, where all words are capitalized (PascalCase); and Snake Case, where words are separated by underscores (snake_case). Notably, these conventions can be used in the same coding style to differentiate variables, functions, and classes. For example, PEP-8 recommends Snake Case for functions and variables and Pascal Case for class names. As most modern code editors and IDEs provide autocompletion of variable, function, and class names, it is no longer a valid excuse to use cryptic one-character variable names (e.g., x, y, z) to save a few keystrokes.

In addition to mastering variable naming, code comments—explanatory human-readable statements not evaluated by the program—are necessary to enhance the code's readability. No matter how beautiful and well-organized your code is, high-level code decisions will not be obvious unless stated. As a corollary, code explanations that can be deduced from the syntax itself should be omitted. Comments can span a single line or several lines, and can be found in three strategic parts: at the top of the program file (header comment), which describes what the code accomplishes and sometimes the code's author/date; above every function (function header), which contains the purpose and behavior of the function; and in line, next to difficult code with behavior that is not obvious or warrants a remark.

Code-styling rules also apply to data science notebooks. However, when writing notebooks, you must also engage in literate programming—a programming paradigm where the code is accompanied by a human-readable explanation of its logic and purpose. In other words, notebooks must tell a story about the analysis, connecting the dots between the code, the results, and the figures. Human-readable language is often written in Markdown [53] when working in Jupyter, or R Markdown [48] when working in R. Little has been written about good practices for literate programming, but our suggested good practices are to include the purpose and interpretation of results for each section of code.

When working with a sizable codebase, we advise modular programming—the practice of subdividing a computer program into independent and interchangeable sub-programs, each one tackling a specific functionality. Modularity enhances code readability and reusability, as well as expedites testing and maintenance. In practice, modularity can be implemented at different levels, from using

functions within a single-file program to separating functionalities into different files in a more complex tool. In Python, subdivisions are defined as follows: modules are a collection of functions and global variables, packages are a collection of modules, libraries are a collection of packages, and frameworks are a collection of libraries. Modules are files with .py extension, while packages are folders that contain several .py files, including one called **init**.py which can be empty or not and allows the Python interpreter to recognize a package.

Finally, there is version control, one of the most important personal practices. Version control entails tracking and managing changes in your code. A popular version-control system is Git [54], which requires a folder to be initiated as a Git repository, after which changes to any of the files inside would be tracked. File modifications must be staged (using git add) and then committed (using git commit). The commit will serve as a screenshot of your project at that time and stage, which you can review or recover later (using git checkout). Additionally, version control allows you to satry new functions in branches (using git branch and git checkout)—independent carbon copies of the main original branch (known as main) that you can optionally merge back to the original copy. Currently, there are multiple hosting services that provide online storage of Git repositories, such as GitHub [55], GitLab [56], or Bitbucket [57], that users can navigate using the web browser or via a graphic user interface (GUI) such as GitHub Desktop [58] or GitKraken [59]. These platforms have the additional benefit of backing up your code in the cloud, keeping your work safe and shareable, which is especially relevant for collaboration.

Level 2: Collaboration

Collaboration is a key aspect of scientific research, but it is especially relevant in computational biology, where interdisciplinary knowledge is often needed. Although collaborators can have a wide range of involvement with your project, here we will consider individuals that share a direct relationship with you and your research. Each type of collaboration requires its own set of good practices, which will be covered in the next paragraphs.

2.1 Share code

Sharing code is one of the most common practices in software development, where large teams work together to develop complex functions and scripts. Although computational biology projects are usually not as big, proper sharing code is still essential. Hosting services, such as GitHub [55], GitLab [56], and Bitbucket [57] (Table 2), allow for a Git repository to be stored online by creating a copy of the repository known as the remote, which becomes the official version of the repository. The key advantage of using a remote is that there will be no direct interaction between different local copies of the repository, also known as clones; instead, each clone will interact with the remote exclusively, updating only if no conflicts between the two exist. This way, if a collaborator updates the remote repository, other collaborators will not be able to send their changes until they update their local copy.

Table 2: Tools for collaborative research.

Goal	Tools
Share code	• *Hosting services*: **GitHub** [55], **GitLab** [56], **Bitbucket** [57]. • *Git branching strategies:* **GitHub flow** [73]. • *Tests:* correctness (e.g. **pytest** [74], **testthat** [75]), style (e.g. **flake8** [76]), vulnerabilities (e.g. **Safety** [77]), coverage (e.g. **codecov** [78]). • *Continous integration:* **tox** [79], **Travis CI** [80], **Circle CI** [81], **Github Actions** [82]). • *Code reviews*: **Github** [83], **Crucible** [84], **Upsource** [85].

Goal	Tools
Share data	• *FAIR principles* [4]: **FAIRshake** [86]. • *Tidy data* [87]. • *Data version control* [88].
Share data science notebooks	• *Static:* **GitHub** [55], **GitLab** [56], **NBviewer** [89]. • *Interactive:* **Binder** [90], **Google CoLab** [71]. • *Comparative:* **nbdime** [91], **ReviewNB** [92].
Share workflows	• *General hosting services*: **GitHub** [55], **GitLab** [56], **Bitbucket** [57]. • ***Dedicated workflow repositories:*** Snakemake Workflow Catalog** [93] , **WorkflowHub** [94].
Share manuscripts	• *General-purpose word processors:* **Google Docs** [95], **Office 365** [96]. • *Scholarly word processors:* **Authorea** [97]. • *Online applications supporting Markup Languages:* **Overleaf** (LaTeX) [98], **Manubot** (Markdown + GitHub) [99].

To guarantee that different collaborators can work simultaneously in the same repository, it is best to implement a branching strategy in the repository (Table 2). In a small team, the most common strategy is to have a single main branch and generate branches from it that each different developer can work on. Then, whenever the developer is ready, they can request to combine—or merge—the changes from their branch into the main branch. This occurs via a process known as pull request (PR). Once a PR has been opened, collaborators can review, approve, and subsequently merge it into the main branch, preserving the commit history. This branching strategy is sometimes referred to as GitHub Flow [73] and will suffice for most projects. For more complex branching systems, see Level 3.

Using Git hosting services for collaboration has many additional benefits. The commit history both shows what was done at each point in time but also specifies the collaborator who made the changes; this allows users to take responsibility for their changes so that if, for example, a bug was introduced, commands such as git blame can pinpoint the cause. To ensure bugs can be easily tracked, descriptive commit messages that follow a standard are recommended [100,101]. Git hosting services can be accessed interactively online or from the terminal with tools such as GitHub CLI [102]. Finally, Git hosting services also allow collaborators to open issues [103] for listing pending tasks and/or asking questions, acting as an open forum for development discussions, which has the advantage of remaining accessible for the future (as opposed to closed email discussions).

Another important concept to consider when developing code, especially with other collaborators, is to develop tests, meaning scripts that will run to find errors in the code (Table 2). Tests can be executed at different levels, from the individual units/components to the system/software as a whole [104]. Unit tests, in particular, are used to determine if specific modules/functions work as intended within the codebase so that if later the function grows in scope, its proper basic functioning is ensured. For instance, if a function was defined for adding numbers, a simple test would be to assess if the function outputs 13 when the inputs 6 and 7 are provided. Besides unit tests, computational biology projects can benefit from implementing integration tests to evaluate the correct interaction between different modules and smoke tests to indicate if any core functionality has been impacted. Test runners, such as pytest [74] for Python and testthat [75] for R, exist to facilitate incorporating tests to the codebase. It is good practice to develop tests at the same time you develop code, as adding tests a posteriori is significantly harder. It is an even better practice to test every single step of the code (from data loading to figure plotting), a concept known in software development as end-to-end testing [105].

Going beyond testing correctness, flake8 [76] will test styling preferences (for complying with PEP8), Safety [77] will test for vulnerabilities among the software's dependencies, and Codecov [78] will test coverage, or the percentage of the codebase tested. As a rule of thumb for testing coverage, the more lines of code tested, the more reliable the software will be. Different types of tests can be funneled

into a single testing pipeline—in a process known as continuous integration (CI)—that can be tuned to run locally whenever commits are made, or online whenever a pull request is opened and/or merged. When running locally, an environment manager/command-line tool, such as tox [79], can help to ensure all tests are executed under different Python versions. Different tools, such as Travis CI [80] or Circle CI [81], can be used to set up the CI cycle online. More recently, GitHub Actions [82] was developed to run integrations directly from GitHub.

Having tests is a great way to ensure that code fulfills a certain level of correctness and styling. However, it is no replacement for human assessment to determine if the code is correct, necessary, and useful. Therefore, peer code review is essential whenever developing code in collaboration (Table 2). While tools, such as Crucible [84] and Upsource [85], exist for making in-line reviews of each file, the most common approach is for you and/or others to directly review the code using the online review tools provided by various hosting services. In the case of GitHub [83], this not only allows the reviewer to open a comment in any line of the code, which creates a thread for the original author to reply but also to suggest changes that can be approved or dismissed. Reviewers can assess many features of the code, from functionality to documentation, while also following good practices, such as using constructive phrasing, which is outside of the scope of this review but presented in detail elsewhere [106,107].

2.2 Share data

The practices of sharing data are similar to sharing code: we should store our datasets, and any changes to them, in a repository and ensure it complies with standards by testing its quality. However, since data has a more consistent structure than code, often existing in standard formats, we should consider additional criteria when sharing it with collaborators (and later with the community). The main set of guidelines that represent these criteria was outlined in what is known as the FAIR principles [4]: data should be Findable (easy to locate online); Accessible (easy to access once found); Interoperable (easy to integrate with other data/applications/workflows/etc); and Reusable (presented in a way that allows for others to use it for the same or different purposes). Tools like FAIRshake [86] can be used to determine if data fits FAIR criteria.

For making data findable, research repositories such as Zenodo [108] and Figshare [109] allow you to assign a digital object identifier (DOI) to any group of files you upload, including data and/or code. Alternatively, regular code repositories like GitHub can be used instead, as you can employ commits and/or releases to identify specific versions of the data, in combination with extensions for Large File Storage (LFS), such as git LFS [110], in the case of data files larger than 100 MB [111]. GitHub can also integrate with Zenodo to automatically archive repositories and assign them a DOI. A final alternative is the Data Version Control (DVC) initiative [88], which is especially useful when performing machine learning, as it can keep track of data, machine learning models, and even scoring metrics.

For making data accessible, we encourage as much as possible to make your repositories open access. In cases in which you or your collaborators prefer some restrictions, you can create guest accounts to provide access to private repositories. For making data interoperable, distinctions between raw and clean data have been made [68], with raw data being the files that came out of the measuring device, and clean data representing the files that are ready to be used for any computational analysis. An important characteristic that clean data should have is to be tidy, which is reviewed in detail elsewhere [87]. Finally, for making data reusable, thorough documentation of the data is required, including experimental design, measurement units, and possible sources of error.

2.3 Share data science notebooks

Jupyter Notebooks have become a fundamental tool for data analysis, which can be shared with collaborators using either static or interactive options. The former shares computational notebooks as rendered text, written internally in HTML. Static notebooks are a good option when you want to avoid any modifications and can work as an archive of past analyses, although interacting with its content is cumbersome—the file must be downloaded and run in a local Jupyter installation. Git-based code repositories, such as GitHub [55] and GitLab [56], automatically render notebooks that can be later shared using the repository's URL. To facilitate this process, Project Jupyter provides a web application called NBviewer [112], where you can paste a Jupyter Notebook's URL, publicly hosted in GitHub or elsewhere, and renders the file into a static HTML web page with a stable link.

Interactive notebooks, on the other hand, not only render the file but also allow collaborators to fully interact with it, tinkering with parameters or trying new input data—no installation required. Binder Project [90] enables users to fully interact with any notebook within a publicly-hosted Git-based repository via a Jupyter Notebook interface, although changes will not be saved to the original file. The platform supports Python and R, among other languages, and additional packages required to run the analysis need to be specified in a configuration file within the repository. Similarly, Jupyter Notebooks can be run interactively using Google CoLab [71] by anyone with a Google account. Notebooks can be updated locally, from any public GitHub repository, or from Google Drive. As an added bonus, Google CoLab notebooks can be edited by multiple developers in real-time. In both cases, the machines provided by these services are comparable to a modern laptop, hence these tools may not be suitable for computing-intensive tasks.

Notebooks should be treated like any other piece of code: updates from different collaborators should be managed with version control in a platform such as GitHub. The problem, however, is that git and other version control systems use line-based differences that are not very well suited for the internal JSON representation of Jupyter notebooks. The extension nbdime [91] can be installed locally to enable content-aware diffing and merging. Additionally, NBreview [92] can be integrated with GitHub to enable content-aware diffing, displaying the old and new versions of a notebook in parallel to facilitate code review.

2.4 Share computational workflows

Computational biology projects often demand using multi-step analyses with dozens of third-party software and dependencies. Although these steps can be described in the documentation, complex workflows are better shared as stand-alone code that can be easily run with minimal file manipulation from collaborators. Doing so can safeguard the reproducibility and replicability of the analysis, leading to better science and fewer challenges downstream.

The simplest way to share a pipeline is through a shell script that receives input files via the command line, allowing flexibility to run analyses with different input data; however, shell scripts offer little control over the overall workflow and cannot re-run specific parts of the pipeline. To address these issues, pipelines are better shared using a workflow automation system. Theoretically, all of the instructions regarding the workflow could be written in the main pipeline file: in Snakemake, this would be the .smk file (or Snakefile); in Nextflow, te .nf file; in CWL, the .cwl file; and in WDL, the .wdl. However, to ensure reproducibility, it is a good practice to share complete pipelines, meaning folder structures, additional files, and software specifications, as well as custom scripts developed for the analysis. These files can be shared using the same tools as code, namely GitHub or any other Git hosting service. Alternatively, they can be uploaded to hosting services specialized in workflows, like Snakemake Workflow Catalog [93] or WorkflowHub [94], currently in beta.

When sharing workflows, consider that sharing software versioning is necessary for your collaborators to reproduce your pipeline using their own computing setup. Conda environments, for example, can be easily created from an environment file (in YAML language), which can be exported from an

existing environment. Notably, Snakemake and Nexflow can be configured to automatically build isolated environments for each rule or step, enabling the running of different versions of a program within the same pipeline, which is especially helpful when using both Python 2 and 3 in the same pipeline, for example. In addition to sharing the specifications of an environment, it is possible to share the environment itself via containers, which we will discuss in Level 3.

2.5 Write manuscripts collaboratively

Writing articles is the primary way we share our research with the scientific community at large. However, writing manuscripts collaboratively comes with its challenges when using classical word processing tools, often resulting in files with different names, jumping from one email inbox to another, and contradictory final versions. The tools we suggest will help to avoid these issues. Companies have become aware of the need for collaborative writing, developing online applications that can be simultaneously edited by multiple people. Google Docs [95] and Microsoft Office 365 [96] are well-known word processors designed for this purpose, with text displayed as it would appear as a printout (known as What-You-See-Is-What-You-Get , or WYSIWYG) and formatting performed using internal features of the application. These platforms are extremely user-friendly and require no specialized knowledge making them a good option when collaborators seek simplicity. Although these applications are not specifically tailored for scientific writing, third-party companies have developed plugins enabling useful features, such as adding scientific references to your document (e.g., Paperpile and Zotero). Companies like Authorea [97] have developed online applications specifically designed for writing manuscripts that offer templates for different types of research projects and allow easy reference additions using identifiers (DOI, PubMed, etc.).

In addition to word processors, text editors are a viable option to write manuscripts when combined with a markup language—a human-readable computer language that uses tags to delineate formatting elements in a document that will be later rendered. Since the formatting process is internally handled by the application, styling elements (e.g., headers, text formatting, and equations) are easily written in text, achieving greater consistency than word processors. Disciplines closely related to computational biology, such as statistics and mathematics, have historically used the markup language LaTeX for writing articles. This language has simple and specific syntax for mathematical constructs making it a popular choice for papers with many equations. To aid collaborative writing, platforms like Overleaf [98] provide online LaTeX editors, supporting features like real-time editing. In addition to LaTeX, an emerging trend in collaborative writing uses the lightweight markup language Markdown within the GitHub infrastructure. The software Manubot [99] provides a set of functionalities to write scholarly articles within a GitHub repository, leveraging all the advantages of Git version control and the GitHub hosting platform [113]. For example, it provides cloud storage and version control. The GitHub user interface also allows offline manuscript discussions using issues and task assignments (see Level 3 for tips on project management). Manubot, in particular, accepts citations using manuscript identifiers and automatically renders the article in PDF, HTML, and Word .doc formats. As a drawback, it requires technical expertise in Git and familiarity with GitHub; as an upside, its reliable infrastructure scales well to large and open collaborative projects. The document you are reading now was fully written using Manubot!

Level 3: Community

The third and final step of this journey is presenting your research to the community. Your main goal should be to share and maintain an open and reproducible project that can sustain community engagement over time. In this section, we will distinguish three sub-goals to make your research: (1) accessible, (2) reproducible, and (3) sustainable. The latter is especially relevant when your research involves developing code that will be used by others in the future (e.g., a tool or workflow), but we believe that our recommendations are relevant to any computational biology project.

3.1. Make your research accessible

Making your research accessible includes ensuring that anyone can access your research long after your paper is published. It is extremely frustrating for any researcher to look for software or a set of scripts from a paper published a few years ago, only to find a "404 error" when accessing the source weblink. Equally frustrating is when authors offer code as "available upon reasonable request," as this often leads to dead-ends and unavailable code.

There are three main ways to publish accompanying code: the supplementary material of the manuscript, privately-owned domains, or uploaded to public repositories. Publishing code as supplementary material has low accessibility for non-open access papers. Moreover, the code will remain completely static and cannot be updated with new features or to correct errors. Making code available via privately-owned domains lacks sustainability, as it requires maintenance of the domain. Therefore, in addition to providing the code as supplementary material and/or via private domains, we recommend uploading it to public repositories, enabling open access and sustainability over time. There are several hosting services for this purpose [55,56,57] (Table 3), all equally valid and typically dependent on established practices in your specific field.

Table 3: Tools for making your research accessible.

Goal	Tool options	Additional remarks
Publish your code	• **GitHub** [55] • **GitLab** [56] • **Bitbucket** [57]	All three options allow you to host your repository online for free. Choose whichever is more common in your own field.
Introduce your code	• **README file** [114]: First file that shows up in a repository.	Provide a landing page to any repository with a short overview of the code (installation, usage, acknowledgments, etc).
Share your code	• Several licensing options [115].	Indicate with a license file what restrictions apply when using your code. If you don't include this, you will loose many users.
Archive your code	• **Github Releases** [116] • **Zenodo** [108]: Provides DOI. • **figshare** [109]: Provides DOI.	Share progressive stable versions of your code as you develop it. Use **semantic versioning** [117] for assigning standard identifiers to your releases.
Publish a tool	• **PyPI** [118]: Python. • **CRAN** [119]: R. • **Bioconductor** [36]: R. • **Bioconda** [70]: Language-agnostic.	Produce a package easy to install and use. Especially useful if you think you could have a userbase that will run the same analysis as you on other datasets and/or conditions.
Publish an interactive web app	• **Dash** [120]: Python. • **R-Shiny** [121]: R.	Provide easy and interactive data exploration to your users. Especially useful if you have large datasets that can be explored in different ways.

When publishing your code in a public repository, two files are fundamental to include: A readme file and a license. A readme file [114] introduces users to the code (Table 3) and should include a description of its main intended use, an overview of the installation, the most commonly-used commands, contact information of the developers, and acknowledgments, if appropriate. We recommend keeping the readme file short and written in a markup language such as Markdown [122] or reStructuredText [123] that will render automatically on the repository's landing page, below the repository file structure.

Adding a license to a repository is also a crucial step (Table 3). Licenses indicate how the code can be used: Is it free to use for any application? Can users modify the code as they please? Does it come with a warranty that it will work? Can it be used for profit? If no license information is provided, researchers might assume that the code is free to use but copyright law in fact prohibits use without

explicit permission by the copyright holder [124]. Many options exist for licensing code [115], from permissive licenses that allow any kind of use with few or no conditions, like the Unlicense and MIT licenses, to more restrictive licenses that enforce disclosing the source and requiring that any adaptation of the code uses the same license, like the GNU licenses. When deciding on a license, as a rule of thumb, consider that the more requirements you add, the fewer potential users you will have, but the more credit you will receive when users utilize your code for their own needs. Academic researchers must also consider what open-source licenses their university supports, as in many cases it will be the university that owns the copyrights.

As a computational biologist, you will likely continue lines of work from scripts or software you have already published. For instance, you could improve the performance of a given function or add a new set of features entirely. Therefore, you should not only be interested in making your code accessible but also in having different versions available. Creating and archiving successive releases of your code (Table 3) allows the organization of different versions of your code as you develop them. GitHub Releases [116] is one way to maintain versions with minimal effort. Research repositories, such as Zenodo [108] or Figshare [109], not only store your code, notebooks, and data, but also provide a DOI for each version allowing it to be included as a citation in a manuscript. This is especially useful when the publication is not available yet or the current version of the code differs widely from what was published. Research repositories can be combined with code repositories; for example, GitHub has a Zenodo integration that will trigger a new archived version every time a new version is released. Regardless of the solution, we recommend keeping logical order to the releases, using a standard such as semantic versioning [117].

In most cases, providing your code as an organized set of scripts and/or notebooks is sufficient for anyone to consult if they wish to reproduce and/or re-utilize it. However, if your code might be used routinely by other researchers, for instance for studying other organisms or other experimental conditions, consider packaging your code as a tool (Table 3) and publishing through a software repository such as Bioconda [70], PyPI [118] if written for Python, or CRAN [119] and Bioconductor [36] if written for R. These increase your possible userbase, as published packages are searchable and can be installed locally with minimal effort.

To increase the accessibility of results to users, an interactive web app or data dashboard can be developed (Table 3). Such apps allow users to interact with data by displaying different sets of variables or changing parameter settings (e.g., the significance of a statistical test). Common options for this goal are Dash [120] for Python, R, and Julia, and Shiny [121] for R. Both platforms can include interactive graphics generated with plotly data visualization libraries [125].

3.2. Make your research reproducible

In addition to having accessible code/data, you also need to ensure anyone can execute your code and obtain the same results. This is especially relevant in computational biology where users will come from different backgrounds and experience. A cornerstone for reproducibility is documentation that explains how the code functions and how to practically achieve your same results. We have distinguished four levels of documentation [126]:

- Tutorials: A group of lessons that teach the reader how to become a user of your code;
- How-to guides: A set of documents that clarify how to solve common problems/tasks;
- Explanations: Discussions that clarify particular topics related to your code;
- References: Technical descriptions of your code's variables/classes/functions.

The extent of required documentation will depend on the number of expected users and, relatedly, can affect how many users you attract. If you foresee that your code has little usability outside of your own research, documenting each function using docstrings [127] might be sufficient. However, if you

aim for a broader userbase, you might want to add a tutorial for beginners, how-to guides for frequently used routines, and explanations for clarifying the science behind your code, which can be re-used in a manuscript. To publish comprehensive documentation online, consider using (1) a standard documentation language such as reStructuredText [123] or Markdown [122], and (2) a documentation platform such as Readthedocs [128], Gitbook [129], or Bookdown [130] (Table 4). Alternatively, you can use a service like GitHub Pages [131] to host the documentation on a dedicated website.

Table 4: Tools for making your research reproducible.

Goal	Tool options	Additional remarks
Document your code	• **Readthedocs** [128]: Uses **reStructuredText** [123]. • **Gitbook** [129]: Uses **Markdown** [122]. • **Bookdown** [130]: Uses **R Markdown** [48]. • **Github Pages** [131]: Separate website.	Comprehensive documentation: from tutorials and how-to guides all the way down to function documentation based on all compiled **docstrings** [127].
Reproducible environments	• **Virtual environment managers**: See Table 1. • **pip-tools** [132]: Administer several environments in a single project.	As a recommendation, try having the minimum number of dependencies needed to reproduce your results.
Reproducible software	• **Docker** [133] • **Singularity** [134]	Package your research as a container ready to run in any computer.
Reproducible commands	• **Make** [24]	Build a program by following a series of steps in a single Makefile.
Reproducible workflows	• **Workflow management systems**: See Table 1.	Run a pipeline of commands on NGS data in a reproducible way.
Reproducible notebooks	• **Interactive notebooks**: See Table 2.	Make your notebooks interactive and reproducible.

Another key aspect of reproducibility is software and dependencies installation. To facilitate this process, you can (1) provide configuration instructions, (2) share dependencies with a virtual environment manager, or (3) share a runtime environment as a container. When setting up software from instructions, it is necessary to ensure the user follows a series of sequential commands in a specific order. To automate this process, Linux systems provide the tool GNU Make [24]. Virtual environment managers handle dependencies and facilitate software installation by building virtual environments from requirements files. To achieve repeatable environments, however, it is recommended to include the specific version of software and libraries, a practice known as dependency pinning. Tools such as pip-tools [132] allow to define different Python environments for a single project depending on the type of user (e.g., end-user versus developer).

Beyond dependency trackers, we recommend ensuring your tool functions as expected across computing infrastructures, even between two different operating systems (e.g., Mac and Windows). This can be achieved via containerization, also known as lightweight virtualization (Table 4). Containers are standardized software that packages an entire runtime environment, meaning everything needed to run your tool: code, dependencies, system libraries and binaries, environmental variables, settings, etc. Instructions for deploying containers are stored in read-only templates called images. Two free tools available for creating containers from images are Docker [133] and Singularity [134]. While Docker is the most popular framework for containerization [60], HPC clusters with shared filesystems favor Singularity due to security issues. In most cases, this is not a problem, since Singularity is compatible with all Docker images.

3.3. Make your research sustainable

Now that your research can be accessed and reproduced by anyone, the final step is to sustain this over time—also known as code maintenance. This is especially relevant if you continue to develop tools by integrating new features requested by users, which can foster a strong community over time. However, even in the case in which your research is a self-contained project, it is important to ensure that the user community can contact you, in case bugs are discovered or parts of your code malfunction due to dependency updates (part of the "software rot" phenomenon [135]). In the following section, we review useful techniques for making your code/software/research sustainable over time.

You can employ a variety of tools to communicate with users, depending on the size of your user base and the scope of questions/comments received (Table 5). For smaller projects, a single-channel solution like Gitter [136] offers a simple way for anyone in the community to ask questions and the developers to answer in threads. For larger projects, however, it could become unmanageable to have all discussions in the same channel, so a multiple-channel solution (i.e., forums), such as Google groups [137] is better suited. GitHub also allows issues to be opened, where collaborators or users can inform developers about bugs or ask questions. Additionally, GitHub recently introduced discussions [138] to maintain questions organized in different threads.

Table 5: Tools for making your research sustainable.

Goal	Tool options	Additional remarks
Tell users how to contact you	• Specific/shorter questions: **Gitter** [136]. • Larger issues / how-to's: **Google groups** [137], **GitHub Discussions** [138].	Provide ways for users to contact you for questions, requests, etc. Remember to visit them periodically!
Track to-do's in your research	• **Github Issues** [139]	Detail specific pending to-do's in your research / allow others to request changes and/or highlight bugs.
Encourage user contributions	• **Contribution guidelines** [140]: How to open issues / contribute code. • **Github Wikis** [141]: More specific how-to guides.	Provide as much information as you can to guide your users. You can also include administrator guidelines.
Foster a respectful community	• Smaller projects: **Contributor Covenant** [142]. • Larger projects: **Citizen Code of Conduct** [143].	Essential when you would like researchers to contribute code.
Branch your repo sustainably	• **Gitflow** [144]	Useful when several developers contribute code to the project. Allows users to get access to stable versions of your research in an ongoing project.
Keep track of your issues	• **Kanban flowcharts** [145]: **Github Projects** [146], **GitKraken Boards** [147]. • **Scrum practices** [148]: **Zenhub** [149], **Jira** [150].	Keep track of your pending tasks in different projects with **Agile** [151] software development practices. Especially useful if your research is split in many different repositories, each with multiple features/fixes to do.
Automate your repo	• **bump2version** [152]: Easier releasing. • **Danger-CI** [153]: Easier reviewing.	Do less, script more!

Now that users know where to contact you, ensure you have developed contribution guidelines [140] (Table 5), detailing how users should (1) open issues and (2) contribute with their own code changes via PRs. These guidelines are intended for new users/contributors, so should be written in the style of a how-to guide; however, they may also include additional instructions for the main developers or the administrator of the repository. Alternatively, the detailed guidelines can be included in a supplemental wiki, which hosting services offer as part of the repository [141]. Equally important is a

code of conduct (Table 5), which includes expectations on how users should behave in the repository and consequences when someone does not comply, promoting a respectful community. Several code-of-conduct templates exist, such as the Contributor Covenant [142] for smaller projects and the Citizen Code of Conduct [143] for larger projects.

Finally, consistent development and maintenance of your software as it grows in scope and number of users will ensure the sustainability of your project. Tools that aid in this include:

1. Branching System: When many developers are involved in a project, more advanced branching methods, such as GitFlow [144], ensure that users can access functional versions of your code while you work on it. (Table 5). Briefly, GitFlow includes two branches with an infinite lifetime: the main and the development (often named as devel). New branches will be based on the development branch, leaving the main one for stable versions of the code. Every time the development branch is merged into the main branch, a version release is created.
2. Project Management: Tools exist to track, organize, and prioritize user issues (Table 5), all based on Agile [151] principles. The simplest approach is implementing a Kanban board [145] (as found in GitHub Projects [146] or GitKraken Boards [147]), where issues are organized in columns that clearly layout the current state of a given task. For larger projects comprising multiple collaborators and/or repositories, a more structured approach, such as a Scrum framework [148] (as implemented by Zenhub [149] and Jira [150]), allows you to prioritize issues by setting milestones and estimating difficulties.
3. Additional Automation: As your project develops, you will find that many aspects can be automated to improve efficiency. bump2version [152] ensures all sections of your code get updated with the new release. Danger-CI [153] and git hooks [154] ensure contributors comply with certain standards in their pull requests. If you are no longer actively maintaining a project, you can use CI (e.g. GitHub Actions [82]) to schedule regular tests to discover if your tool/code starts malfunctioning due to software rot and/or dependency issues. Finally, we advise against implementing too many automation tools at the start of a project, but adding them as needed. If you find yourself routinely performing a task, consider automating it.

Case Studies

We will now exemplify the effective use of the introduced tools by presenting three different computational biology projects from the literature (Figure 2) (two more cases are presented in the Supplementary Material). Note that our list of projects is not meant to be comprehensive, but rather is intended to be a short overview of how projects in computational biology benefit from robust tools and software development practices. Additionally, it will be evident that there is considerable redundancy in chosen tools across case studies. For instance, all projects include an environment manager such as Conda, and a version control system like Git. This redundancy is intentional as it highlights the ubiquity of some tools.

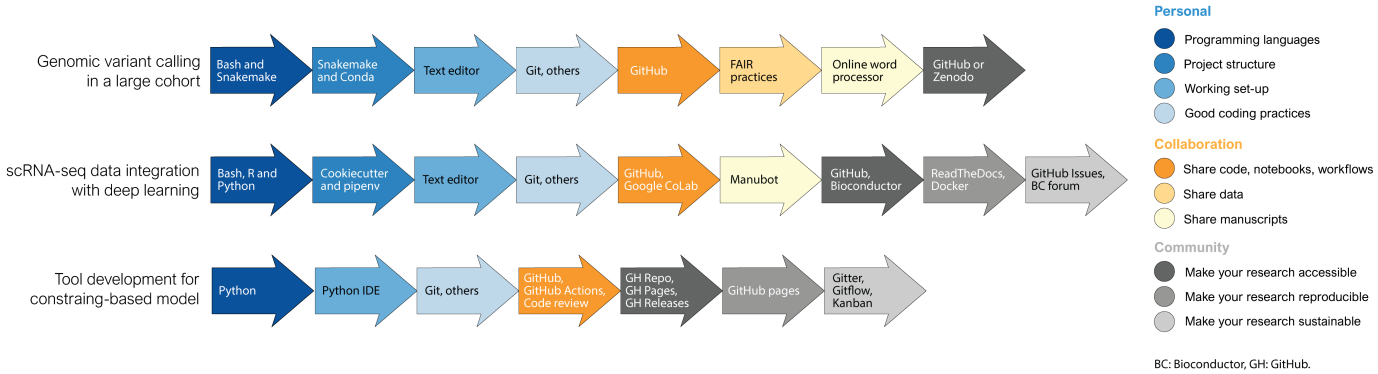

Figure 2: Examples of computational biology projects and associated depending on the nature of the research and the number of people involved.

Case study 1: Genomic variant detection in a large cohort

The availability and affordability of NGS allow for the routine assessment of dozens to thousands of genomes. Resequencing experiments enable the discovery and genotyping of genomic variation within large cohorts to answer key questions regarding population history and susceptibility to disease. For this example, let's consider a project including whole-genome Illumina sequencing and variant identification in thousands of individuals such as Aganezov et al. [155]. Herein, the challenge resides in applying a multi-step variant-calling pipeline on many samples in a reproducible manner.

In this particular project, the authors utilized the AnVIL cloud computing platform [65,66], which uses WDL for workflow description. However, if you have access to an HPC cluster, then a project of this nature can be performed using the workflow automation tool Snakemake [25], employing Python to parse sample names and perform other data handling operations, and following Snakemake workflow template [31] for folder structure. A Conda [34] environment can hold all necessary software since a wide array of software designed for genomic analyses is available via the Bioconda [70] repository. Coding the workflow can be done in any text editor that offers easy integration with Git tracking and hosting, such as Visual Studio Code [40]. For code styling, you can run Snakefmt [52] to follow best practices.

A project of this magnitude usually requires collaborators from other research groups. The pipelines and scripts can be shared using a GitHub repository [55]. If privacy is a concern, the repository can be set as private and made public in later stages of the project. To write the manuscript, general-purpose word processors such as Google Docs [95] would suffice. Considering that these types of data are a valuable resource for the community, FAIR principles [4] for data sharing should be followed. In addition to uploading the raw data in a public repository like the European Nucleotide Archive (ENA) or the National Center for Biotechnology Information (NCBI), we encourage open sharing of your code and notebooks in a GitHub repository archived in Zenodo [108] with a DOI.

Case study 2: Single-cell (sc)RNA-seq data integration

scRNA-seq is a rapidly evolving technology that has enabled the study of cell heterogeneity and developmental changes of a cell lineage, otherwise intractable with bulk RNA-seq. Current scRNA-seq experiments deliver the transcriptomic profiles of thousands to millions of cells [156], making them a suitable target for machine- or deep-learning approaches. Among the many challenges imposed by this technology, integration of scRNA-seq datasets is key, especially in case-control studies where cell types should be functionally matched across datasets before evaluating differences across conditions. For this case study, we will consider the development of an unsupervised deep-learning method for data integration as described in Johansen and Quon [157].

This kind of project often uses a combination of Python, R, and shell scripting. Python can be used to write and train deep-learning models with TensorFlow [158] or PyTorch [159] libraries. R enables straightforward data pre-processing with tools such as Seurat [160,161]. Shell scripting can process large-scale raw data files in HPC clusters. Additionally, we advise using Python's reticulate library [162] to incorporate Python tools into the existing R ecosystem. To set up your working directory, we recommend a structure like Cookiecutter Data Science [29], which includes separate folders for trained models and other components of a deep-learning project. To establish a software environment, Python virtual environments, such as virtualenv [32], work well with Tensorflow and PyTorch. Coding can be performed in any general-purpose text editor, such as Visual Studio Code [40], where updates can be easily pushed/pulled to/from GitHub. As a good practice, maintain

modular, properly-commented code and name files with data stamps and model parameters to facilitate revisiting projects. Additionally, take advantage of tools such as TensorBoard [163] to diagnose, visualize, and experiment with your models.

When working with collaborators, code should be shared through a Git hosting service like GitHub. When multiple users need to edit the code in real-time, Google CoLab [71] offers interactive coding and GPU access. In addition to the code repository, a Manubot [99] can be created to write the manuscript collaboratively. To make your tool accessible to a larger community, publish it to a public GitHub and include a readme [114] and an appropriate license file [115]. Considering that most users in the field use R, you can go one step further and share your code as a Bioconductor package [36], making sure your method can be called directly in R and that interacts with standard data structures in the field. For better reproducibility, document your method including example tutorials in a platform like ReadTheDocs [128], and share the software environment needed to deploy the models as a Docker image [133]. GitHub issues [139] and Bioconductor forums [164] are suitable platforms to promptly reply to users' questions, bug reports, and requests for code enhancements.

Case study 3: Tool development for constraint-based modeling

The last case study we will present is related to constraint-based modeling, a common approach used for simulating cellular metabolism. In this approach, the metabolic network of a given organism is inferred from its genome and/or literature and converted to a matrix that contains the reaction's stoichiometry. Using a few simple assumptions, this matrix can be used to perform simulations under different experimental conditions to obtain additional insight into cellular physiology [165]. Several tools have been developed for working with these types of models. Here, we will consider cobrapy [166], a community tool for reading/writing constrained-based models and performing basic simulation operations.

A tool of this nature is especially useful if developed in Python, as it should ideally be presented as a package that can be easily installed with pip [35]. The use of an IDE is ideal for this case, as it will provide additional features for testing changes in the tool. Practices that for other case studies were useful now become essential, like complying with coding style and using version control, as hundreds of people will likely read your code. Furthermore, the code should be (1) available via a hosting service such as GitHub [55], (2) tested with a continuous development tool such as GitHub Actions [82], (3) manually reviewed by collaborators to ensure correctness, (4) released following semantic versioning standards [117], and (5) documented with a companion documentation website, rich with tutorials and how-to guides. As a branching strategy, Gitflow [144] is probably the best suited, as it allows all changes to existing code in a development branch and stable releases in the main branch.

Finally, due to the large scope of this project, additional considerations must be made to maintain a healthy user base. Offer a place for users to raise questions, such as Gitter [136], Google groups [137], or GitHub Discussions [138], and make sure to reply to new questions often. Guidelines should also be provided for everything, including how to: open issues with example templates, contribute using pull-request templates, communicate within the community via a code of conduct, and perform other routine tasks with development guidelines and/or wikis. Addressing issues routinely and quickly is also essential in a project of this nature to avoid giving the impression of a stagnant project. Additional tools such as a Kanban flowchart with the help of GitHub Projects [146] will help prioritize issues, or Jira [150] or Zenhub [149] if several repositories require joint coordination.

Final words

Good practices in computational biology have gained the spotlight among researchers thanks to several guiding principles published, as well as the increasing usage of Git-based repositories and

workflow managers. This review adds to the existing literature by introducing a comprehensive list of good practices and associated tools that can be applied to any computational biology project, regardless of the specific subfield or the experience of the researcher.

We are aware that the many tools and practices introduced in this study and their ever-changing nature may seem overwhelming, especially for someone new to the field. To overcome this, we encourage you to implement only a few practices and tools first, starting from your personal research, and expanding your repertoire over time. More important than any specific tool is keeping a mindset of striving for reproducibility. We also note that our highlighted list of tools is not comprehensive, with many new tools being released. Updated reviews will be essential to help new computational biologists enter the field as well as to keep experienced computational biologists up to date with the latest trends.

The consequences of not following good computational practices are often not seen immediately but become evident and detrimental towards project progress over time. As with all scientific endeavors, computational biology heavily relies on previous knowledge; as such, the good practices we adopt serve as building blocks for the overall reproducibility of the field, propelling novel and exciting future discoveries.

Acknowledgments

We would like to thank Nelson Johansen for his insights on the scRNA-seq data integration case study.

References

1. **NIH working definition of bioinformatics and computational biology**
   Huerta Michael, Downing Gregory, Haseltine Florence, Seto Belinda, Liu Yuan
   (2000-07-17)
   https://www.kennedykrieger.org/sites/default/files/library/documents/research/center-labs-cores/bioinformatics/bioinformatics-def.pdf

2. **What is bioinformatics? A proposed definition and overview of the field.**
   NM Luscombe, D Greenbaum, M Gerstein
   Methods of information in medicine (2001) https://www.ncbi.nlm.nih.gov/pubmed/11552348
   PMID: 11552348

3. **How do scientists develop scientific software? An external replication**
   Gustavo Pinto, Igor Wiese, Luiz Felipe Dias
   2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2018-03) https://doi.org/ggk6nc
   DOI: 10.1109/saner.2018.8330263 · ISBN: 9781538649695

4. **The FAIR Guiding Principles for scientific data management and stewardship**
   Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, … Barend Mons
   Scientific Data (2016-12) https://doi.org/bdd4
   DOI: 10.1038/sdata.2016.18 · PMID: 26978244 · PMCID: PMC4792175

5. **Barely sufficient practices in scientific computing**
   Graham Lee, Sebastian Bacon, Ian Bush, Laura Fortunato, David Gavaghan, Thibault Lestang, Caroline Morton, Martin Robinson, Philippe Rocca-Serra, Susanna-Assunta Sansone, Helena Webb
   Patterns (2021-02) https://doi.org/gjpcb6
   DOI: 10.1016/j.patter.2021.100206 · PMID: 33659915 · PMCID: PMC7892476

6. **Good enough practices in scientific computing**
   Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, Tracy K. Teal
   PLOS Computational Biology (2017-06-22) https://doi.org/gbkbwp
   DOI: 10.1371/journal.pcbi.1005510 · PMID: 28640806 · PMCID: PMC5480810

7. **Best Practices for Scientific Computing**
   Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, … Paul Wilson
   PLoS Biology (2014-01-07) https://doi.org/qtt
   DOI: 10.1371/journal.pbio.1001745 · PMID: 24415924 · PMCID: PMC3886731

8. **Software engineering for scientific big data analysis**
   Björn A Grüning, Samuel Lampa, Marc Vaudel, Daniel Blankenberg
   GigaScience (2019-05-01) https://doi.org/gf4f4m
   DOI: 10.1093/gigascience/giz054 · PMID: 31121028 · PMCID: PMC6532757

9. **So you want to be a computational biologist?**
   Nick Loman, Mick Watson

Nature Biotechnology (2013-11) https://doi.org/p3j
DOI: 10.1038/nbt.2740 · PMID: 24213777

10. **Ten simple rules for biologists learning to program**
    Maureen A. Carey, Jason A. Papin
    PLOS Computational Biology (2018-01-04) https://doi.org/gft4n3
    DOI: 10.1371/journal.pcbi.1005871 · PMID: 29300745 · PMCID: PMC5754048

11. **Streamlining Data-Intensive Biology With Workflow Systems**
    Taylor Reiter, Phillip T. Brooks, Luiz Irber, Shannon E. K. Joslin, Charles M. Reid, Camille Scott, C. Titus Brown, N. Tessa Pierce
    Bioinformatics (2020-07-01) https://doi.org/gg353v
    DOI: 10.1101/2020.06.30.178673

12. **A Padawan Programmer's Guide to Developing Software Libraries**
    James T. Yurkovich, Benjamin J. Yurkovich, Andreas Dräger, Bernhard O. Palsson, Zachary A. King
    Cell Systems (2017-11) https://doi.org/gg8tqz
    DOI: 10.1016/j.cels.2017.08.003 · PMID: 28988801

13. **Ten Simple Rules for Taking Advantage of Git and GitHub**
    Yasset Perez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, ... Juan Antonio Vizcaíno
    PLOS Computational Biology (2016-07-14) https://doi.org/gbrb39
    DOI: 10.1371/journal.pcbi.1004947 · PMID: 27415786 · PMCID: PMC4945047

14. **Ten Simple Rules for Reproducible Research in Jupyter Notebooks**
    Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, Peter W. Rose
    arXiv (2018-10-13) https://arxiv.org/abs/1810.08055v1

15. **Bash - GNU Project - Free Software Foundation** https://www.gnu.org/software/bash/

16. **Welcome to Python.org**
    Python.org
    https://www.python.org/

17. **R: The R Project for Statistical Computing** https://www.r-project.org/

18. **R: The R Project for Statistical Computing** https://www.r-project.org/

19. **The Perl Programming Language - www.perl.org** https://www.perl.org/

20. **MathWorks - Makers of MATLAB and Simulink** https://www.mathworks.com/

21. **The Julia Programming Language** https://julialang.org/

22. **cplusplus.com - The C++ Resources Network** https://www.cplusplus.com/

23. **Rust Programming Language** https://www.rust-lang.org/

24. **Make - GNU Project - Free Software Foundation** https://www.gnu.org/software/make/

25. **Snakemake - A framework for reproducible data analysis** https://snakemake.github.io/

26. **A DSL for parallel and scalable computational pipelines | Nextflow** https://www.nextflow.io/

27. **Home**
    Common Workflow Language (CWL)
    Common Workflow Language (CWL) https://www.commonwl.org/

28. **OpenWDL** https://openwdl.org/

29. **Home - Cookiecutter Data Science** https://drivendata.github.io/cookiecutter-data-science/

30. **GitHub - Reproducible-Science-Curriculum/rr-init: Research project initialization and organization following reproducible research guidelines**
    GitHub
    https://github.com/Reproducible-Science-Curriculum/rr-init

31. **GitHub - snakemake-workflows/snakemake-workflow-template: A template for standard compliant snakemake-workflows**
    GitHub
    https://github.com/snakemake-workflows/snakemake-workflow-template

32. **Virtualenv — virtualenv 20.14.2.dev2+g8d78ee0 documentation**
    https://virtualenv.pypa.io/en/latest/

33. **Project Environments** https://rstudio.github.io/renv/index.html

34. **Conda — Conda documentation** https://docs.conda.io/en/latest/

35. **pip documentation v22.0.4** https://pip.pypa.io/en/stable/

36. **Bioconductor - Home** https://www.bioconductor.org/

37. **RStudio Package Manager** https://www.rstudio.com/products/package-manager/

38. **A hackable text editor for the 21st Century**
    Atom
    https://atom.io/

39. **Sublime Text - the sophisticated text editor for code, markup and prose**
    https://www.sublimetext.com/

40. **Visual Studio Code - Code Editing. Redefined** https://code.visualstudio.com/

41. **Notepad++** https://notepad-plus-plus.org/

42. **welcome home : vim online** https://www.vim.org/

43. **GNU Emacs - GNU Project** https://www.gnu.org/software/emacs/

44. **Project Jupyter** https://jupyter.org

45. **PyCharm: the Python IDE for Professional Developers by JetBrains**
JetBrains
https://www.jetbrains.com/pycharm/

46. **Home — Spyder IDE** https://www.spyder-ide.org/

47. **RStudio | Open source & professional software for data science teams**
https://www.rstudio.com/

48. **R Markdown** https://rmarkdown.rstudio.com/

49. **PEP 8 – Style Guide for Python Code | peps.python.org** https://peps.python.org/pep-0008/

50. **GitHub - google/styleguide: Style guides for Google-originated open-source projects**
GitHub
https://github.com/google/styleguide

51. **Black 22.3.0 documentation** https://black.readthedocs.io/en/stable/

52. **GitHub - snakemake/snakefmt: The uncompromising Snakemake code formatter**
GitHub
https://github.com/snakemake/snakefmt

53. **Markdown Guide** https://www.markdownguide.org/

54. **Git** https://git-scm.com/

55. **GitHub: Where the world builds software**
GitHub
https://github.com/

56. **Iterate faster, innovate together | GitLab** https://about.gitlab.com/

57. **Bitbucket | The Git solution for professional teams**
Atlassian
Bitbucket https://bitbucket.org/product

58. **GitHub Desktop**
GitHub Desktop
https://desktop.github.com/

59. **GitKraken Legendary Git Tools | GitKraken** https://www.gitkraken.com

60. **Stack Overflow Developer Survey 2021**
Stack Overflow
https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021

61. https://www.kaggle.com/kaggle/kaggle-survey-2021

62. **GitHub - Bioconductor/BiocManager: CRAN Package For Managing Bioconductor Packages**
GitHub
https://github.com/Bioconductor/BiocManager

63. **Welcome to the Tidyverse**
Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, … Hiroaki Yutani
Journal of Open Source Software (2019-11-21) https://doi.org/ggddkj
DOI: 10.21105/joss.01686

64. **Why scientists are turning to Rust**
Jeffrey M. Perkel
Nature (2020-12-03) https://doi.org/ghqc7g
DOI: 10.1038/d41586-020-03382-2 · PMID: 33262490

65. **Migrate Your Genomic Research to the Cloud**
The AnVIL
https://anvilproject.org/

66. **Inverting the model of genomics data sharing with the NHGRI Genomic Data Science Analysis, Visualization, and Informatics Lab-space**
Michael C. Schatz, Anthony A. Philippakis, Enis Afgan, Eric Banks, Vincent J. Carey, Robert J. Carroll, Alessandro Culotti, Kyle Ellrott, Jeremy Goecks, Robert L. Grossman, … Jason Walker
Cell Genomics (2022-01) https://doi.org/gn7rf9
DOI: 10.1016/j.xgen.2021.100085 · PMID: 35199087 · PMCID: PMC8863334

67. **Home - Cromwell** https://cromwell.readthedocs.io/en/stable/

68. **A Quick Guide to Organizing Computational Biology Projects**
William Stafford Noble
PLoS Computational Biology (2009-07-31) https://doi.org/fbbpkn
DOI: 10.1371/journal.pcbi.1000424 · PMID: 19649301 · PMCID: PMC2709440

69. **Distribution and Reproducibility — Snakemake 7.3.8 documentation**
https://snakemake.readthedocs.io/en/stable/snakefiles/deployment.html

70. **News — Bioconda documentation** https://bioconda.github.io/

71. **Google Colaboratory** https://colab.research.google.com/

72. **Best practices — Snakemake 7.3.8 documentation**
https://snakemake.readthedocs.io/en/stable/snakefiles/best_practices.html

73. **GitHub flow**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/get-started/quickstart/github-flow

74. **pytest: helps you write better programs — pytest documentation**
https://docs.pytest.org/en/stable/

75. **Unit Testing for R** https://testthat.r-lib.org/

76. **Flake8: Your Tool For Style Guide Enforcement — flake8 4.0.1 documentation**
https://flake8.pycqa.org/en/latest/

77. **Safety CLI - Security for your Python dependencies** https://pyup.io/safety/

78. **Codecov - The Leading Code Coverage Solution**
Codecov
https://about.codecov.io/

79. **Welcome to the tox automation project — tox 3.25.1.dev3 documentation**
https://tox.wiki/en/latest/

80. **Homepage | Travis CI – Start building today!**
Travis CI
https://www.travis-ci.com/

81. **Continuous Integration and Delivery**
CircleCI
https://circleci.com/

82. **Features • GitHub Actions**
GitHub
https://github.com/features/actions

83. **Reviewing changes in pull requests**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests

84. **Crucible Code Review Tool for Git, SVN, Perforce and More**
Atlassian
Atlassian https://www.atlassian.com/software/crucible

85. **Upsource: Code Review and Project Analytics by JetBrains**
JetBrains
https://www.jetbrains.com/upsource/

86. **FAIRshake** https://fairshake.cloud/

87. **Tidy Data**
Hadley Wickham
Journal of Statistical Software (2014) https://doi.org/gdm3p7
DOI: 10.18637/jss.v059.i10

88. **Data Version Control · DVC**
Data Version Control · DVC
https://dvc.org/

89. **nbviewer** https://nbviewer.org/

90. **The Binder Project** https://mybinder.org/

91. **nbdime – diffing and merging of Jupyter Notebooks — nbdime 3.1.1.dev documentation**
https://nbdime.readthedocs.io/en/latest/

92. **ReviewNB - Jupyter Notebook Code Reviews & Collaboration** https://www.reviewnb.com/

93. **Snakemake workflow catalog** https://snakemake.github.io/snakemake-workflow-catalog/

94. **WorkflowHub** https://workflowhub.eu/

95. **Google Docs: Free Online Document Editor | Google Workspace**
https://www.facebook.com/GoogleDocs/

96. **Microsoft 365 - Subscription for Office Apps | Microsoft 365** https://www.microsoft.com/en-us/microsoft-365

97. **Open Research Collaboration and Publishing - Authorea** https://www.authorea.com/

98. **Overleaf, Online LaTeX Editor** https://www.overleaf.com

99. **Manubot - Manuscripts, open and automated** https://manubot.org

100. **Semantic Commit Messages**
Sparkbox
https://sparkbox.com/foundry/semantic_commit_messages

101. **Conventional Commits**
Conventional Commits
https://www.conventionalcommits.org/en/v1.0.0/

102. **GitHub CLI**
GitHub CLI
https://cli.github.com/

103. **About issues**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/issues/tracking-your-work-with-issues/about-issues

104. **Types of Software Testing**
GeeksforGeeks
(2017-08-01) https://www.geeksforgeeks.org/types-software-testing/

105. **Combine API and UI Testing For Confidence At Every Layer Of Your Application**
smartbear.com
https://smartbear.com/en/solutions/end-to-end-testing/

106. **How to do a code review**
eng-practices
https://google.github.io/eng-practices/review/reviewer/

107. **Code Review Guidelines for Humans**
Philipp Hauer
Philipp Hauer's Blog (2018-07-31) https://phauer.com/2018/code-review-guidelines/

108. **Zenodo - Research. Shared.** https://zenodo.org/

109. **about figshare** https://knowledge.figshare.com/about

110. **Git Large File Storage**
Git Large File Storage
https://git-lfs.github.com/

111. **About large files on GitHub**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/repositories/working-with-files/managing-large-files/about-large-files-on-github

112. **nbviewer** https://nbviewer.org/

113. **Open collaborative writing with Manubot**
Daniel S. Himmelstein, Vincent Rubinetti, David R. Slochower, Dongbo Hu, Venkat S. Malladi, Casey S. Greene, Anthony Gitter
PLOS Computational Biology (2019-06-24) https://doi.org/c7np
DOI: 10.1371/journal.pcbi.1007128 · PMID: 31233491 · PMCID: PMC6611653

114. **Make a README**
Make a README
https://www.makeareadme.com

115. **Licenses**
Choose a License
https://choosealicense.com/licenses/

116. **Managing releases in a repository**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository

117. **Semantic Versioning 2.0.0**
Tom Preston-Werner
Semantic Versioning https://semver.org/

118. **PyPI · The Python Package Index**
PyPI
https://pypi.org/

119. **The Comprehensive R Archive Network** https://cran.r-project.org/

120. **Dash Overview** https://plotly.com/dash

121. **Shiny** https://shiny.rstudio.com/

122. **Daring Fireball: Markdown Syntax Documentation**
https://daringfireball.net/projects/markdown/syntax

123. **reStructuredText** (2021-10-22) https://docutils.sourceforge.io/rst.html

124. **The Legal Side of Open Source**
Open Source Guides
(2022-04-19) https://opensource.guide/legal/

125. **Plotly** https://plotly.com/api/

126. **Documentation System** https://documentation.divio.com/

127. **Python Docstrings**
GeeksforGeeks
(2017-06-01) https://www.geeksforgeeks.org/python-docstrings/

128. **Home | Read the Docs** https://readthedocs.org/

129. **GitBook - Where software teams break knowledge silos.** https://www.gitbook.com/

130. **Home | Bookdown** https://bookdown.org/home/

131. **GitHub Pages**
GitHub Pages
https://pages.github.com/

132. **GitHub - jazzband/pip-tools: A set of tools to keep your pinned Python dependencies fresh.**
GitHub
https://github.com/jazzband/pip-tools

133. **Home**
Docker
https://www.docker.com/

134. **Home**
Sylabs
https://sylabs.io/

135. **What is Software Rot? - Definition from Techopedia**
Techopedia.com
http://www.techopedia.com/definition/22202/software-rot

136. **Gitter** https://gitter.im/

137. **Redirecting to Google Groups** https://groups.google.com/forum/m

138. **GitHub Discussions Documentation**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/discussions

139. **About issues**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/issues/tracking-your-work-with-issues/about-issues

140. **Setting guidelines for repository contributors**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors

141. **About wikis**
GitHub Docs
http://ghdocs-prod.azurewebsites.net:80/en/communities/documenting-your-project-with-wikis/about-wikis

142. **Contributor Covenant: A Code of Conduct for Open Source and Other Digital Commons Communities** https://www.contributor-covenant.org/

143. **policies/citizen_code_of_conduct.md at master · stumpsyn/policies**
GitHub
https://github.com/stumpsyn/policies

144. **A successful Git branching model**
nvie.com
http://nvie.com/posts/a-successful-git-branching-model/

145. **Kanban - A brief introduction**
Atlassian
Atlassian https://www.atlassian.com/agile/kanban

146. **GitHub Issues · Project planning for developers**
GitHub
https://github.com/features/issues

147. **Boards Timelines FAQ | GitKraken** https://www.gitkraken.com/boards-and-timelines

148. **What is Scrum?**
Scrum.org
https://www.scrum.org/resources/what-is-scrum

149. **ZenHub - Productivity Management for Software Teams** https://www.zenhub.com/

150. **Jira | Issue & Project Tracking Software**
Atlassian
Atlassian https://www.atlassian.com/software/jira

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

152. **GitHub - c4urself/bump2version: Version-bump your software with a single command**
GitHub
https://github.com/c4urself/bump2version

153. **Danger - Stop Saying "You Forgot To..." in Code Review**
Orta Therox
http://danger.systems/ruby/index.html

154. **Git - Git Hooks** https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

155. **A complete reference genome improves analysis of human genetic variation**
Sergey Aganezov, Stephanie M. Yan, Daniela C. Soto, Melanie Kirsche, Samantha Zarate, Pavel Avdeyev, Dylan J. Taylor, Kishwar Shafin, Alaina Shumate, Chunlin Xiao, … Michael C. Schatz
Genomics (2021-07-13) https://doi.org/gk6dwc
DOI: 10.1101/2021.07.12.452063

156. **Exponential scaling of single-cell RNA-seq in the past decade**
Valentine Svensson, Roser Vento-Tormo, Sarah A Teichmann
Nature Protocols (2018-04) https://doi.org/gc5ndt
DOI: 10.1038/nprot.2017.149 · PMID: 29494575

157. **scAlign: a tool for alignment, integration, and rare cell identification from scRNA-seq data**
Nelson Johansen, Gerald Quon
Genome Biology (2019-12) https://doi.org/gh5jhj
DOI: 10.1186/s13059-019-1766-4 · PMID: 31412909 · PMCID: PMC6693154

158. **TensorFlow**
TensorFlow
https://www.tensorflow.org/

159. **PyTorch** https://www.pytorch.org

160. **Tools for Single Cell Genomics** https://satijalab.org/seurat/

161. **Integrated analysis of multimodal single-cell data**
Yuhan Hao, Stephanie Hao, Erica Andersen-Nissen, William M. Mauck, Shiwei Zheng, Andrew Butler, Maddie J. Lee, Aaron J. Wilk, Charlotte Darby, Michael Zager, … Rahul Satija
Cell (2021-06) https://doi.org/gkchrs
DOI: 10.1016/j.cell.2021.04.048 · PMID: 34062119 · PMCID: PMC8238499

162. **Interface to Python** https://rstudio.github.io/reticulate/

163. **TensorBoard**
TensorFlow
https://www.tensorflow.org/tensorboard

164. **Bioconductor Forum** https://support.bioconductor.org/

165. **Constraint-based models predict metabolic and associated cellular functions**
Aarash Bordbar, Jonathan M. Monk, Zachary A. King, Bernhard O. Palsson
Nature Reviews Genetics (2014-02) https://doi.org/f5sk8s
DOI: 10.1038/nrg3643 · PMID: 24430943

166. **COBRApy: COnstraints-Based Reconstruction and Analysis for Python**
Ali Ebrahim, Joshua A Lerman, Bernhard O Palsson, Daniel R Hyduke
BMC Systems Biology (2013-12) https://doi.org/gb3qmh
DOI: 10.1186/1752-0509-7-74 · PMID: 23927696 · PMCID: PMC3751080

167. **RNA sequencing: the teenage years**
Rory Stark, Marta Grzelak, James Hadfield
Nature Reviews Genetics (2019-11) https://doi.org/gf6vfx
DOI: 10.1038/s41576-019-0150-2 · PMID: 31341269

168. **Genome-scale metabolic modelling of P. thermoglucosidasius NCIMB 11955 reveals metabolic bottlenecks in anaerobic metabolism.**
Viviënne Mol, Martyn Bennett, Benjamín J. Sánchez, Beata K. Lisowska, Markus J. Herrgård, Alex Toftgaard Nielsen, David J Leak, Nikolaus Sonnenschein
Microbiology (2021-02-01) https://doi.org/gh4s89
DOI: 10.1101/2021.02.01.429138

169. **A systematic assessment of current genome-scale metabolic reconstruction tools**
Sebastián N. Mendoza, Brett G. Olivier, Douwe Molenaar, Bas Teusink
Genome Biology (2019-12) https://doi.org/gh3pjm
DOI: 10.1186/s13059-019-1769-1 · PMID: 31391098 · PMCID: PMC6685185

170. **MEMOTE for standardized genome-scale metabolic model testing**
Christian Lieven, Moritz E. Beber, Brett G. Olivier, Frank T. Bergmann, Meric Ataman, Parizad Babaei, Jennifer A. Bartell, Lars M. Blank, Siddharth Chauhan, Kevin Correia, … Cheng Zhang
Nature Biotechnology (2020-03-01) https://doi.org/gh4s88
DOI: 10.1038/s41587-020-0446-y · PMID: 32123384 · PMCID: PMC7082222

Supplementary Material

Additional case study 1: RNA-seq differential gene expression

Differential gene expression (DGE) analysis is commonly used in the field of functional genomics. Its main goal is to determine quantitative changes in gene expression levels between different experimental conditions or different populations. Nowadays, given the availability of NGS technologies, most DGE analyses are based on RNA-seq data, being the primary application of the technology [167]. Here, we will consider a study designed to gain insights regarding a specific condition of interest, leading to interest genes that can be functionally characterized in animal models afterward. The experimental setup included a control group and an experimental group with the condition of interest, both sequenced using RNA-seq. The experiment was conducted four times independently to get four replicates per group.

In this example, the most relevant part is personal research. The first step is to decide which programming languages to use. Considering that the bioinformatics analysis will include a first step performed in the command line, where raw reads will undergo quality control and the number of reads per transcripts will be counted, followed by a second step for data filtering and statistical analysis, the programming languages to use will be Shell and R. The folder structure must have separated spaces for raw data, results, documentation, and scripts used in the analysis. We recommend cloning the RR-init template into your HPC cluster. For this project, the best option will be to use a Conda environment with R installed, where you can download packages from Bioconductor and Bioconda. A shell script will be a suitable option for the first part of the analysis in the HPC cluster, and R Studio for the second part of analysis and visualization. We advise to follow literate programming, especially when writing R code, and to track changes using Git.

Additional case study 2: Genome-scale metabolic model

A sub-group of constraint-based models (Case Study 3) are genome-scale metabolic models, where the model represents the complete metabolism of a cell, inferred from genome sequencing. These models are significantly larger in size, making the model generation and curation steps hard to trace back if we don't use adequate tools. As a reference paper, consider the generation, curation, and validation of a genome-scale model for *Parageobacillus thermoglucosidasius* [168], a thermophilic facultative anaerobic bacteria with promising traits for industrial metabolic engineering.

The first step in a project of this nature is to use one of the many reconstruction algorithms available [169] to start from what is referred to as a draft reconstruction; therefore, the choice of programming language for that section will depend on the selected algorithm. After that, there is a lengthy step of model curation and gap filling, in order to end up with a model that can produce all necessary building blocks of the cell. For this step, we recommend a basic setup of Python as programming language and Conda as environment manager, due to their ease of use and the growing number of Python packages being developed in the field. Additionally, we advise using Jupyter Notebook as the main working setup and Git for version control, as you can use different notebooks (or different versions of a notebook) as logs of analysis performed on your working draft. When collaborating, tools that will be especially useful while working on a genome-scale model are unit-testing, to ensure your model maintains a certain quality as you and others develop it [170], and ReviewNB, to keep track of

changes in notebooks across commits and/or branches. Finally, when sharing the model within the community, Zenodo is a great option for defining different versions of the model, and any issue tracker will make it possible for users to pinpoint mathematical or biological inconsistencies in the network.