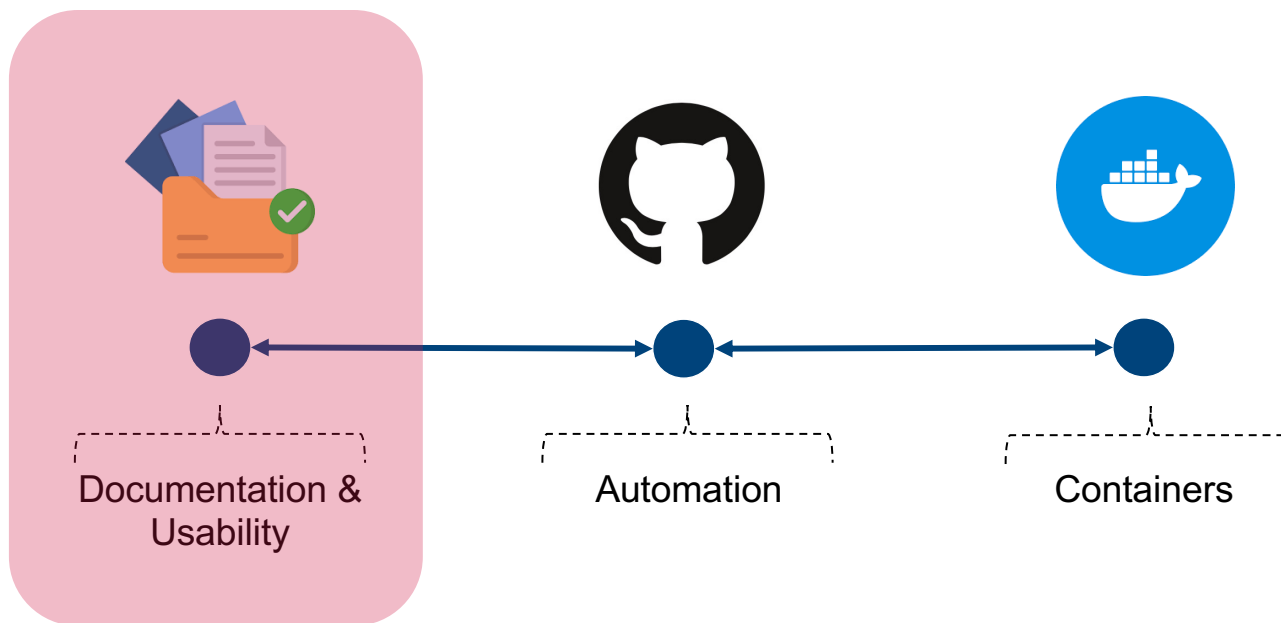


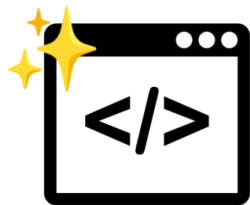
Best practices for reproducibility 🎉





Documentation & Usability

- Finding things takes a lot of time and effort
- Standard and predictable organization saves time
- Great documentation will make an efficient bioinformatics workflow possible, and prepare your data and project for publications, as grant agencies and publishers increasingly require this information.



Your awesome tool



Documentation

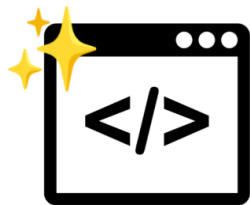
Source (edited) by [Documentation and Usability](#)





Future You is confused with your code

- We tend to feel code we are writing is clear and obvious in the moment we are writing it.
- Future You does not agree! They have no idea what you were thinking and why you were thinking it. 🧐
- And if Future You doesn't understand what's going on, odds are others won't either.



Your awesome tool



Documentation



Future you

Source (edited) by [Documentation and Usability](#)



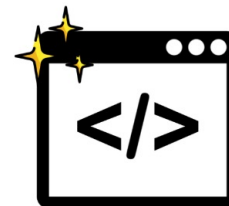


Tool development

After years of tedious work, my informatics tool is working!



Tina the Tool Developer



Tina's awesome tool

Source (edited) by [Documentation and Usability](#)



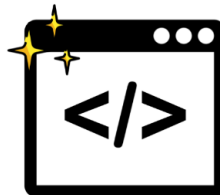


Tool development

Tina's tool is just what I need for my research project! 🎉



Tina the Tool Developer



Uri the Tool User

Source (edited) by [Documentation and Usability](#)

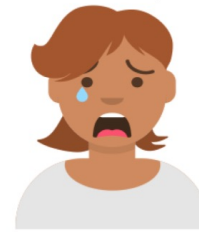






Tool development

I have other projects due! I can't spend more time trying to figure this tool out.



Uri the Tool User

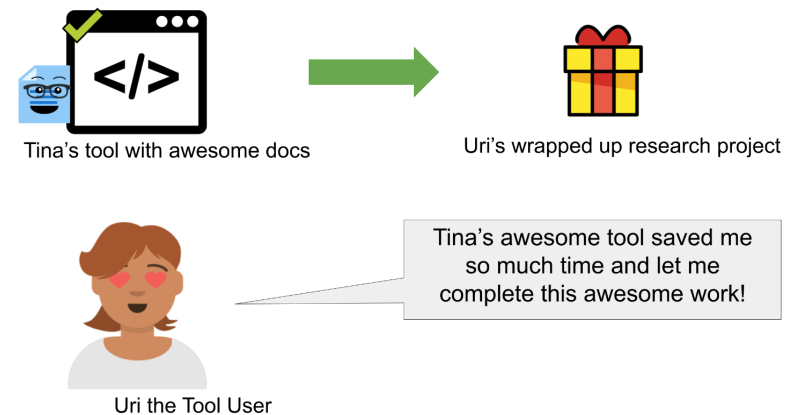
Source (edited) by [Documentation and Usability](#)





Documentation is worth the time!

- Tool developers feel unenthused about the process of creating documentation.
- Documentation process requires a different skill set from the tool development itself; meaning many developers were likely not attracted to tool development because of documentation and may not be sure how to craft good documentation ([Wolf 2016](#)).
- Thorough and easy-to-digest documentation not only benefits users but tool developers themselves!



Source (edited) by [Documentation and Usability](#)





Current problems on usability in the bioinformatics tool development

1. Tools developed in academia are often left to deprecate after publication **because novelty is often prioritized over long-term maintenance and usability** ([Mangul et al. 2019](#)).
2. Bioinformatics tool development teams generally don't have the resources to hire **user-centered design experts**, and the small and specialized user communities are often overlooked and not incentivized to give feedback ([Pavelin et al. 2012](#)).
3. There is **a lack of resources/education about usability** specific to bioinformatics tool developing communities ([Pavelin et al. 2012](#)).





General principles about documentation and usability



1. User-centered development is a form of empathy.
2. Identify your user community
3. Create user friendly guides and templates to maximize the usability of developed workflows and pipelines
4. Document code clearly
5. Communicate while minimizing the impact of jargon
6. Navigability: Make help easy to find
7. Get helpful user feedback and provide users with a method of contact





Project organization and tidiness

OVERVIEW	
Questions	Objectives
1. What research questions am I interested in investigating?	Think about and clarify as best as possible the research questions to tackle in this project.
2. What type of sequencing technologies/assays will I use? What metadata should I collect?	Think about and understand the types of sequencing data and the overall experiment.
3. What type of tools/methods are available for my analyses? Do I need to benchmark them?	Understand the importance of using the best method available, benchmarking, standards: is this method well documented, well maintained and accepted across the research community?
4. How should I structure my sequencing data and analyses?	Think about timelines and prioritize your research questions. Understand the purpose and execution challenges of each analysis module and how each module serves as the next step in the computational framework but also in the context of your research aims.





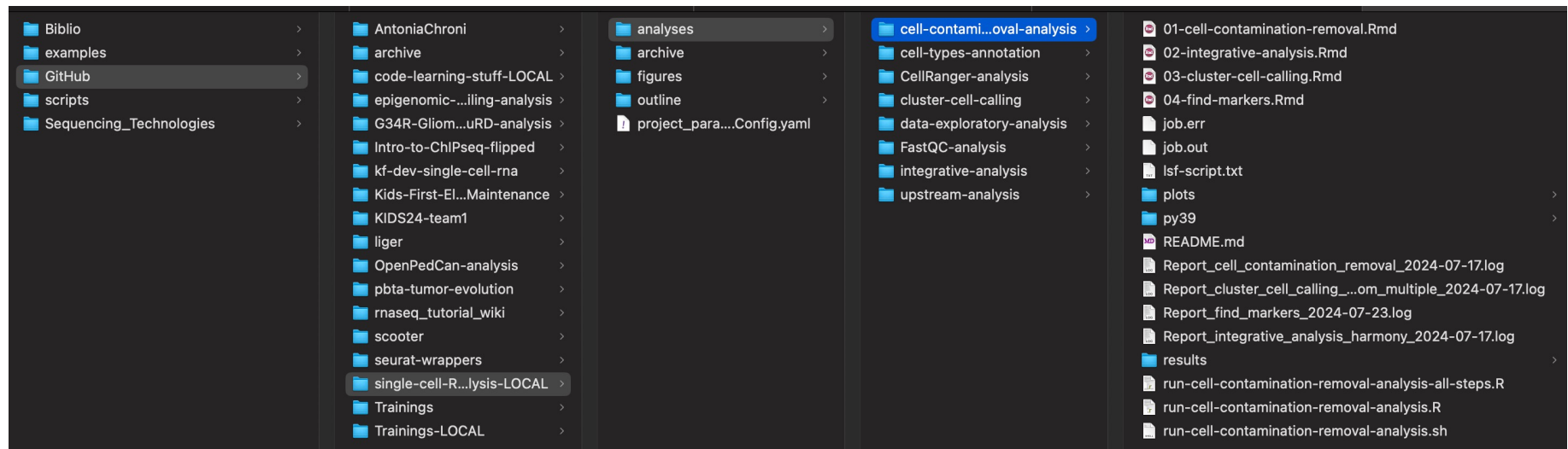
Project organization and tidiness

- **Use Folders/Directories**
- **Keep separate projects separated**
- **Separate sections for units within a project**
 - analyses/<module_name>
 - code
 - plots
 - results
 - reports
 - data
 - figures
 - project_parameters.Config.yaml
 - ...
- **Documentation throughout:** Describe what files do and how they are organized/folder.



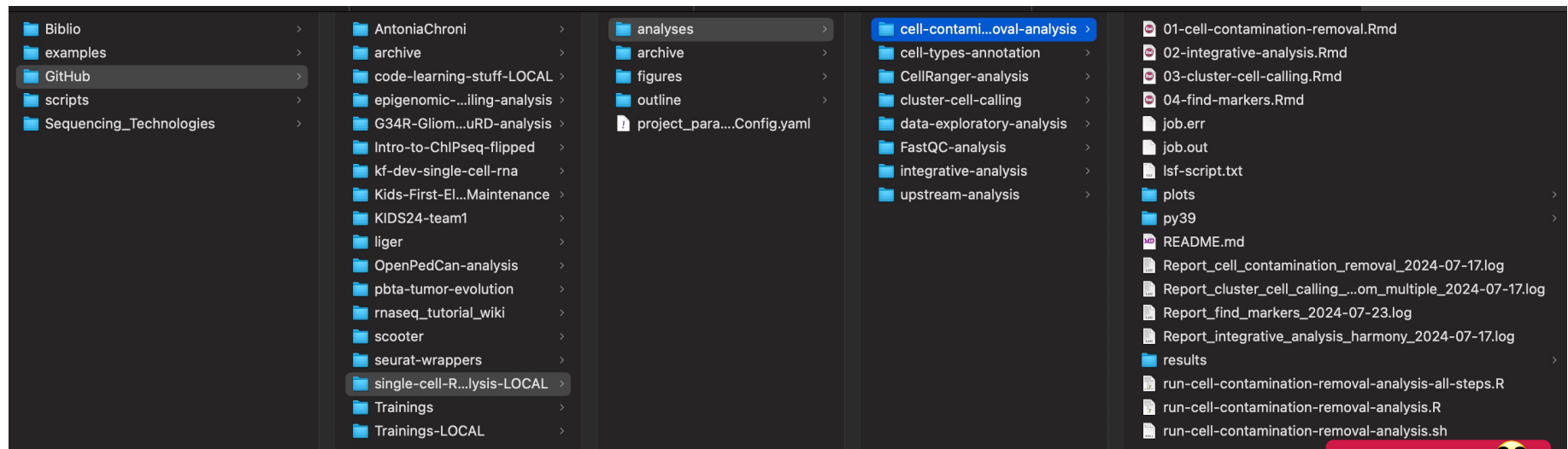


My typical project organization system





My typical project organization system



``reports`` 🧐





Naming of [Files] is a difficult task...

- Some “good” file names:
 - `Res_0.1_Markers_all.tsv`
 - `Report_cell_contamination_removal_2024-07-17.html`
 - `04-find-markers.Rmd`
- Some “not so good” file names:
 - `script.py`
 - `AVRS638GVEW4.fastq.gz`
 - `tastingnotes (3).docx`
 - `My final cohort FINAL3 for mouse UPDATED.docx`

Source (edited) by [Data Lab Reproducibility Workshop](#)





Machine friendly

- Avoid spaces
 - Old computer systems get confused by spaces
 - All computer systems are old underneath
 - Use underscores or dashes to separate words instead
- Use “standard” characters:
 - Letters, numbers, underscores, and dashes
 - Periods only for file extensions (**.txt**, **.tsv**, **.R**, **.tar.gz**)
 - Many characters have special meanings in code. Avoid them! (e.g. * + ? | \$ / “)
- Be consistent with case
 - Don’t *assume* case has meaning: on some systems it does, and on some it doesn’t
 - But always *act* as if it does!
 - Never have two files that are the same but for case

Source (edited) by [Data Lab Reproducibility Workshop](#)





Human friendly

- Names should contain information about file content
- Short names are tempting, but you may regret choosing them!
 - **01.R**
 - **data.txt**
 - **tests.py**
- Use long descriptive names
 - **01_download-mouse-data.sh**
 - **Res_0.1_Markers_top_10_Heatmap.png**
- Which files do you want to look for before a deadline?
- Which files do you want to get from your collaborator?

Source (edited) by [Data Lab Reproducibility Workshop](#)





Files you didn't create

- All the guidelines and suggestions for file names are great for your files, but sometimes files come from other sources
 - If you are lucky, they will follow nice conventions! 🎉
 - but often they won't 😞
- To rename or not to rename, that is the question
 - Leaving the name as it was sent can make it easier to track in correspondence
 - Reasons to rename:
 - uninformative generic names: data.txt
 - add source or date information
 - converting spaces or other special characters (but try to write code that can handle these!)
 - If you choose to rename, do it with a script and document the original name and source.

Source by [Data Lab Reproducibility Workshop](#)





Read style guides and use them (with judgement)

- Which style doesn't really matter, but find some agreement with collaborators.
 - [R tidyverse](#)
 - [Google style guides](#) (R, Python, others):
 - Packages are a mix of styles... use what fits and find your level of comfort.

Source by [Data Lab Reproducibility Workshop](#)





Read style guides and use them (with judgement)

- Variable and function name styles
 - `my_variable` vs. `myVariable`
 - `do_task()` vs. `task_doer()`
- Indentation and spacing
 - `counter=1` vs. `counter = 1`
 - tabs vs. spaces for indentation (very controversial!)
 - line length (try not to let lines get too long)
- Commenting style and format

Source by [Data Lab Reproducibility Workshop](#)





How to design a script and note content

- A descriptive header comment first
 - describe the purpose of the script or notebook
- Setup code next
 - Packages and imported code
 - Inputs, outputs, and other “constant” parameters
 - Option parsing (scripts e.g. functions)
 - Custom function definitions
- Finally: the body of the code and content
 - Break up sections with comments and headings as appropriate

Source by [Data Lab Reproducibility Workshop](#)





Explicit “namespaces” to avoid conflicts

- Sometimes multiple packages have functions with the same name

- R: Use `package::function()` syntax

to avoid ambiguity

- `dplyr::filter()` vs. `stats::filter()`

- Also provides in-code documentation:

What package did this strange function come from?

- Bonus: you don't need a `library()` statement

- Python: `package.function()` syntax

is standard

- avoid `from package import function`

- use `import pandas as pd` and similar if there is a common standard

```
> library(dplyr)
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':
```

```
filter, lag
```

```
The following objects are masked from 'package:base':
```

```
intersect, setdiff, setequal, union
```

Source by [Data Lab Reproducibility Workshop](#)





Random number seeds

- Some code makes use of random numbers
 - simulations
 - fitting statistical models/machine learning
 - PCA, UMAP, tSNE
- But sometimes we want perfect reproducibility! (debugging, testing)
 - luckily, computers don't use real random numbers
 - random number generators are functions: given the same starting point (seed), they will give the same results
- Start your code by setting the seed for replicability
 - R: `set.seed(2024)`
 - Python: `random.seed(2024)`
 - other packages and tools: look at the docs for the correct option, some packages don't use the language defaults

Source by [Data Lab Reproducibility Workshop](#)





Always 🧐 Be 🧐 Documenting

Technical tips and tricks

- Consistent documentation across each module: why this module is useful, what type of data/input is needed, what method(s) were selected and why (add publications, benchmarking when necessary), what output files are generated.
- ALWAYS comment your code! Do this from the beginning of the process. Think of your coworkers who will use the code later.
- Use a consistent naming strategy and meaningful names. Filenaming looks and acts different on different operating systems. We will use hyphens, not underscores, to separate words—for example, `my-script.Rmd`. Search engines interpret hyphens in file and directory names as spaces between words.
- Use informative file names, and use `_` and `-` strategically in the file names so you can use regex on filenames to subset them.





Always 🧐 Be 🧐 Documenting

Technical tips and tricks

- Use simple functions to do repeated operations or complex tasks (>2 times).
- Keep lines of code fairly short (say <400 lines of code) and let RStudio indent and clean your code for you, e.g., use Lint R package.
- Shorten the name of variables and/or names of objects and keep them informative.
- Do not consider a module or tool fix done before its relevant documentation update is also completed.
- Use numbers for consistent sorting of your scripts (if multiple) in the module properly – write a bash file to run all scripts in the proper order.
- Don't overwrite data files. If data files change, create a new file. At the top of an analysis file define paths to all data files (even if they are not read in until later in the script).





Always 🧐 Be 🧐 Documenting

Technical tips and tricks

- For complex figures, it can make sense to pre-compute the items to be plotted as its own intermediate output data structure. The code to do the calculation then only needs to be adjusted if an analysis changes, while the things to be plotted can be reused any number of times while you tweak how the figure looks.
- Keep one copy of all code files and keep this copy under revision management.
- Use version control (github) to keep track of code versioning.
- Use packrat, checkpoint or docker to ensure the versions of programs and packages used for the entire process.
- If you are making changes, always consider backward compatibility. Sometimes creating a new function (with a new name) not a new version is more appropriate.

[Making YOUR Code Reproducible: Tips and Tricks](#)





Always 🧐 Be 🧐 Documenting

Rules for writing helpful code source comments

1. Comments should not duplicate the code.
2. Good comments do not excuse unclear code.
3. If you can't write a clear comment, there may be a problem with the code.
4. Comments should dispel confusion, not cause it.
5. Explain unidiomatic code in comments.
6. Provide links to the original source of copied code.
7. Include links to external references where they will be most helpful.
8. Add comments when fixing bugs.
9. Use comments to mark incomplete implementations.

[Best practices for writing code comments](#)





Always 🧑🏻 Be 🧑🏻 Documenting

Good code source documentation

- Is part of writing good code
- Increases the readability of your code
- Clarify where the code requires explanation
- Can help you write out your thought process
- ⚠️ Be aware of comments when updating code. You may need to update the comments too! ⚠️

Source (edited) by [Documentation and Usability](#)





Technical tips and tricks about data organization



- Leave the raw data raw - do not change it!
- Put each observation or sample in its own row in datasets/cohorts.
- Put all your variables in columns - the thing that vary between samples, like 'strain' or 'DNA-concentration'.
- Have column names be explanatory, but without spaces. Use '-', '_' or camel case instead of a space. For instance 'library-prep-method' or 'LibraryPrep' is better than 'library preparation method' or 'prep', because computers interpret spaces in particular ways.
- Do not combine multiple pieces of information in one cell. Sometimes it just seems like one thing, but think if that's the only way you'll want to be able to use or sort that data. For example, instead of having a column with species and strain name (e.g. E. coli K12) you would have one column with the species name (E. coli) and another with the strain name (K12). Depending on the type of analysis you want to do, you may even separate the genus and species names into distinct columns.
- Export the cleaned data to a text-based format like CSV (comma-separated values) format. This ensures that anyone can use the data and is required by most data repositories.





Get helpful feedback and user contact method








- **Direct users to file a GitHub issue – create templates.**
- Have a Slack channel that you direct users to.
- Have a separate email inbox that you have a notification set up for.
- Have a link to a form for users to submit (Google forms are free).





Determine a plan for usability testing



-  Create your well-documented project with data, code, plots, results, and reports
-  Determine cohorts and type of experiments
-  Choose a test-dataset – explain why
-  Use Docker (or other) to keep packages versioning consistent
-  GitHub your project out and get it code reviewed
-  Use your test-dataset for validation during the code review process
-  Use another one (or more) datasets for validation purposes and fixing bugs





More resources

- [Documentation and Usability](#)
- [Benefits of Software Documentation - SDLC Best Practices](#)
- [Software Documentation Types and Best Practices](#)
- [Making YOUR Code Reproducible: Tips and Tricks](#)
- [Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks](#)
- Data tidiness and data wrangling for NGS projects, please see [Project Organization and Management for Genomics](#)



