Journal Club: *Best Practices for Scientific Computing*

Flic Anderson

15/02/2023

HOW TO (RE-)USE THIS MATERIAL

This is a `.html` presentation created in `R Markdown` with `ioslides`.

(It's been written in a .Rmd file, and I generated .html slides by 'knitting' it in Rstudio.)

You can check out the code used to make these slides at the Talk repo on Github, and adapt it for your own presentations if you like - there's a MIT Licence on the repo, which means:

"Basically, you can do whatever you want as long as you include the original copyright and license notice in any copy of the software/source."

Source: tl;drLegal

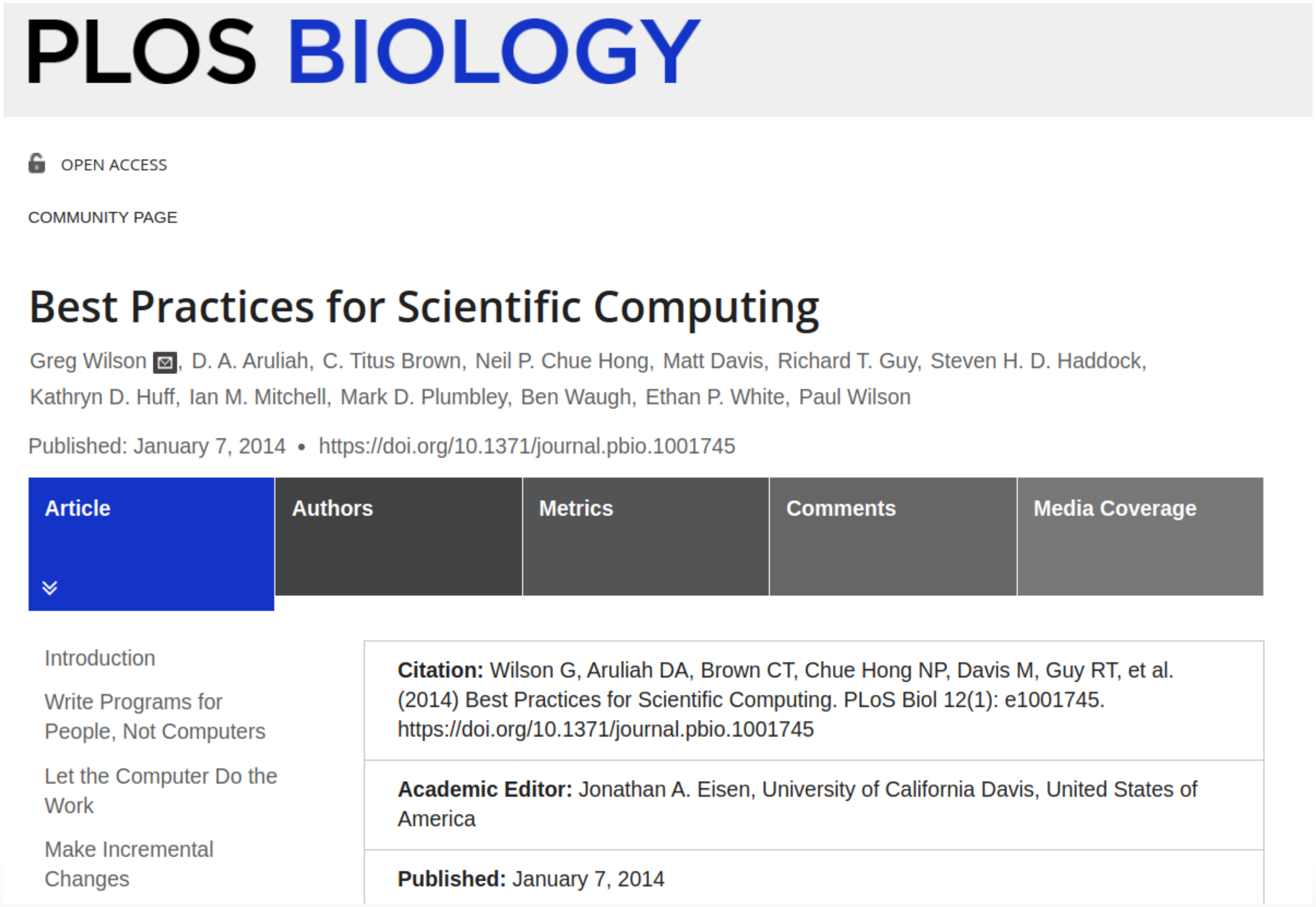
Outline

Article:
- Paper format
- Background / research context
- Paper's aims
- Best Practices
- Authors' conclusions

Discussion & Qs:
- My opinions about the paper
- Your questions about the paper
- Discussion - which best practice do you think is most important?

RS: Research software
SE: Software engineering

The Article



PLOS BIOLOGY

OPEN ACCESS

COMMUNITY PAGE

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, Ben Waugh, Ethan P. White, Paul Wilson

Published: January 7, 2014 • https://doi.org/10.1371/journal.pbio.1001745

Citation: Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745. https://doi.org/10.1371/journal.pbio.1001745

Academic Editor: Jonathan A. Eisen, University of California Davis, United States of America

Published: January 7, 2014

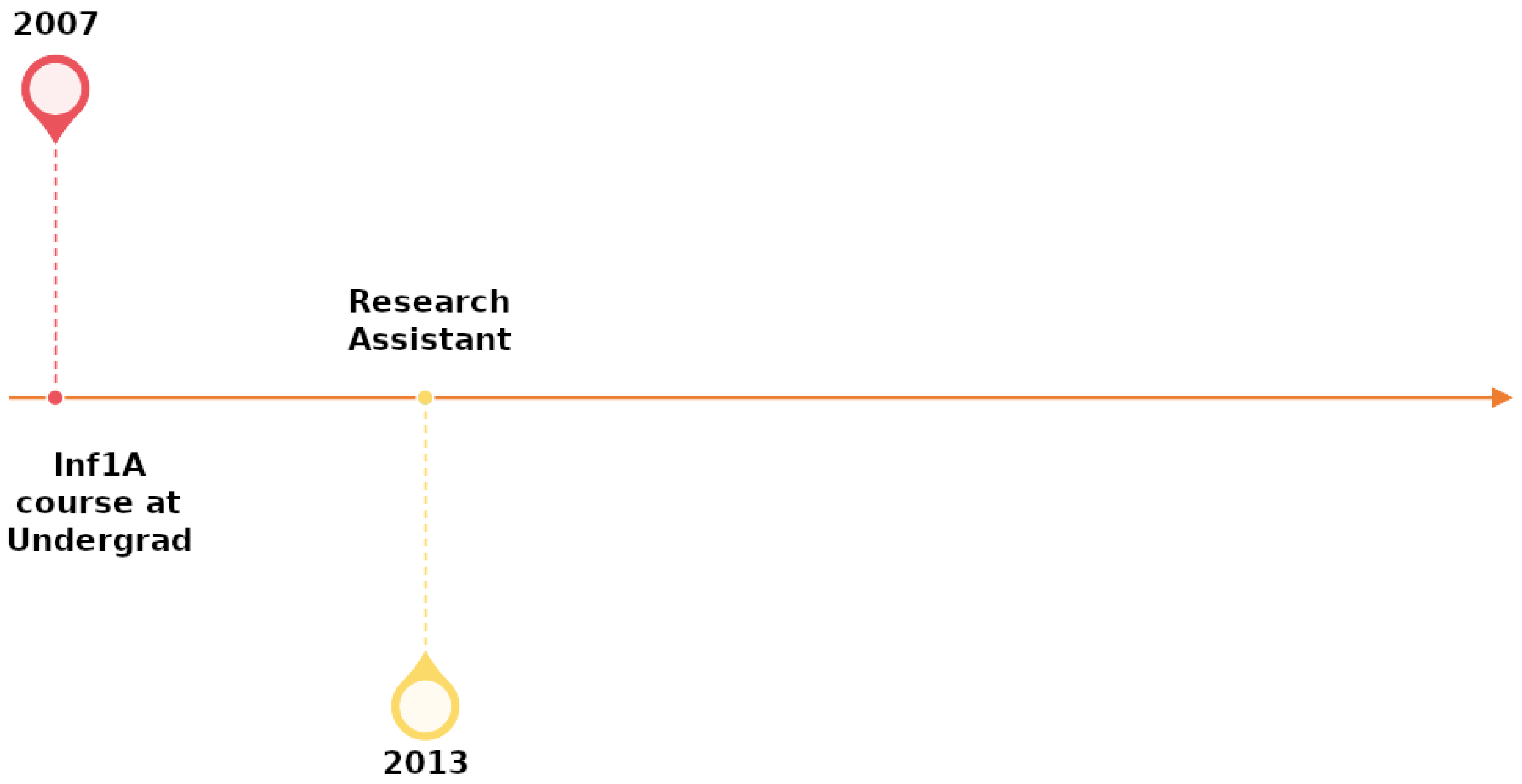
Paper Format

- presenting **8 broad best practice tips** & *24 recommendations*
- no abstract / methods / research Qs
- reasonably informal paper, **accessible and readable**
- *influenced my career*

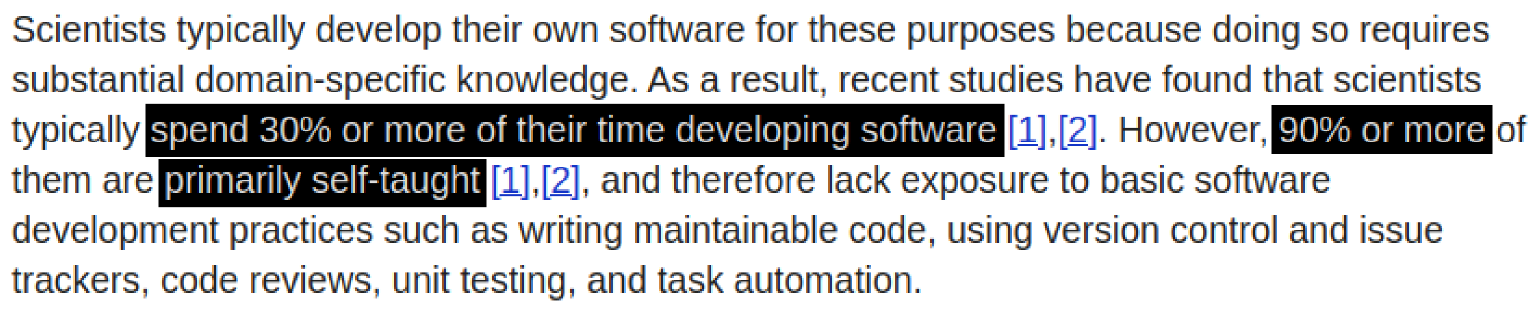
Background / Research Context

- article published 2014 @ PLOS Biology
- Wilson 2006 paper “Software Carpentry: getting scientists to write better code by making them more productive” - prototype ideas
- Heroux & Willenbring 2009 “Barely sufficient software engineering: 10 practices to improve your CSE software” (minimalist!)
- Kelly, Hook & Sanders 2009 “Five Recommended Practices for Computational Scientists Who Write Software”
- Carpentries’ reports (2004+) gathered data from foundational programming/data skills workshop evaluations
- discusses scientific computing, but applies widely to any software used for generating research outputs

MY Background/ Research Context



Paper's Aims


Scientists typically develop their own software for these purposes because doing so requires substantial domain-specific knowledge. As a result, recent studies have found that scientists typically spend 30% or more of their time developing software [1],[2]. However, 90% or more of them are primarily self-taught [1],[2], and therefore lack exposure to basic software development practices such as writing maintainable code, using version control and issue trackers, code reviews, unit testing, and task automation.

- authors' goals in sharing these best practices:
  - **faster development** of RS, results
  - code is more **reliable**, inspires confidence in results
  - **reproducibility** of results
- justifies importance in terms of **reducing errors/retractions**; *improving productivity*

1 - **Write programs for people, not computers**

2 - **Let the computer do the work**

3 - **Make incremental changes**

4 - **Don’t repeat yourself (or others)**

5 - **Plan for mistakes**

6 - **Optimise software only after it works correctly**

7 - **Document design and purpose, not mechanics**

8 - **Collaborate**

‘Best Practices for Scientific Computing’, Wilson et al. 2014.

1. Write programs for people, not computers

- *beware memory limits* in users, not just computers: reduce cognitive load!
    - **break up programs** into *task-oriented functions* or chunks
- self documenting **descriptive names**
- **consistent_styling_of_code** -> faster reading, fewer errors

2. Let the computer do the work

- **CLI** commands & tasks -> **scripts**
- **automate workflows** with a *Workflow Management System* or *build tool* like Make
- **store provenance info** (common file formats plz):
  - raw data IDs, version numbers
  - *parameter* values generating specific outputs
  - *library/program versions*
- cite your software!

WfMS examples: nextflow, snakemake

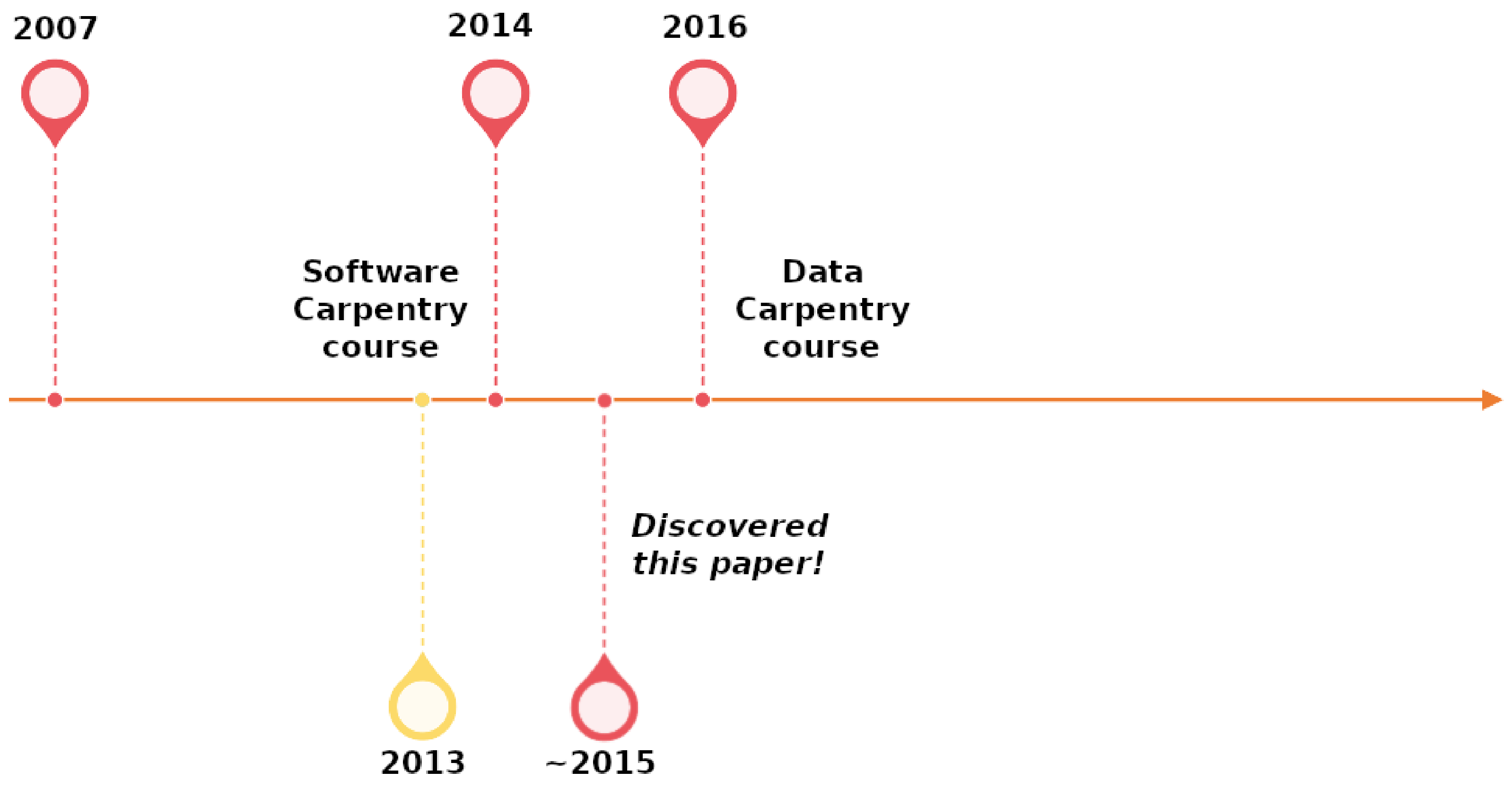
3. Make incremental changes

- RS uses an *iterate-improve* process; similar to Agile development process
  - *no 'requirements' available* upfront for RS
- **keep iterations SHORT** (~1 week), get feedback, improve
- use **version control system** for code AND data
  - **vc manually created things**…
  - *regenerate* the rest (archive binaries, store metadata)

4. Don’t repeat yourself (or others)

- *copies of data / ‘code clones’ is bad*
- **only one version of a file / variable / ID** that can be updated in ONE PLACE ONLY
- write IKEA code but avoid DIY:
    - **modularity** -> fixable, reusable and easy to remember (*mental model* of code)
    - **use others’ libraries and packages**

Update...


2007
2014
2016
Software Carpentry course
Data Carpentry course
Discovered this paper!
2013
~2015

5. Plan for mistakes

- *defensive programming*: **use assertions** in your code
- run *automated testing* with **testing libraries**
- **turn bugs into tests** (prevents silent reappearance of ‘fixed’ issues) for RS
- **use debuggers**

6. Optimise software only after it works correctly

- **write, profile THEN fix** performance bottlenecks
- **write high-level code**:
    - more efficient *per line of code per unit of time*
    - more understandable / self-documenting
    - prototype in high-level languages, THEN implement optimised versions

7. Document design and purpose, not mechanics

- share knowledge *(bus factor!)*
- **explain design decisions**
- **refactor** instead of trying to explain complex code
- *documentation generators*: **embed docs** into code (Try Sphinx)
- *“literate programming”* - code into docs! (Jupyter Notebooks for Python+)

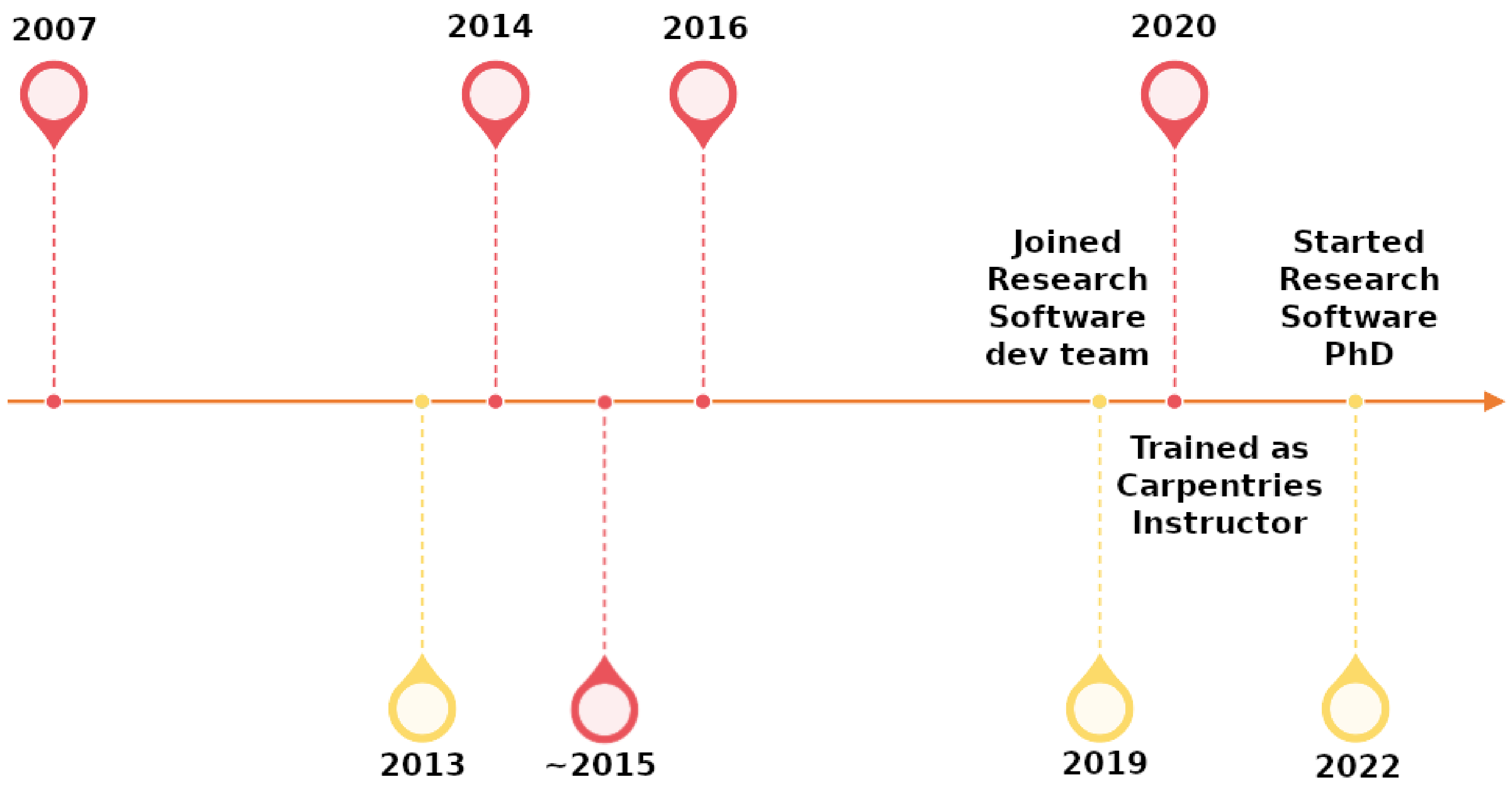
8. Collaborate

- “code review: *get other people to read your code to find bugs fast & cheap!*“ - says decades of research
    - require review **before merging**!
- **pair programming** for complex coding tasks or onboarding new folks to project
- use **issue tracking** (eg. Github)
- your future self…

Authors’ Conclusions

- useful for **solo coders or multi-dev RS teams**
- **add practices gradually**; overlap helps
- scientists need to start adopting tools and approaches to **improve quality & efficiency** of software
    - time costs offset “almost immediately” by productivity gains
- *universities / funders must support researchers* to write better software

This Paper Might Change Your Career…


2007
2014
2016
2020
Joined Research Software dev team
Started Research Software PhD
Trained as Carpentries Instructor
2013
~2015
2019
2022

Opinions, Discussion, Qs…

My Opinions

- well-written, supports aims of the paper: **successful!**
- **explains SE concepts well** to non-specialists
- **escalating order** of recommendations
- *possibly off-putting* to extreme novices/experts? (e.g. optimisation/profiling)
- Wilson et al. 2017: Good Enough Practices in Scientific Computing in PLOS Computational Biology

Q: did publishing in PLOS Biology limit reach into other fields?

Reminder of Best Practices

1 - **Write programs for people, not computers**

2 - **Let the computer do the work**

3 - **Make incremental changes**

4 - **Don’t repeat yourself (or others)**

5 - **Plan for mistakes**

6 - **Optimise software only after it works correctly**

7 - **Document design and purpose, not mechanics**

8 - **Collaborate**

‘Best Practices for Scientific Computing’, Wilson et al. 2014.

Any Questions?