

Motivation and Overview of Best Practices in HPC Software Development

Presented by



<https://pesoproject.org> <https://rapids.lbl.gov>

Members of



*Consortium for the Advancement
of Scientific Software*
<https://cass.community>

With prior support from



David E. Bernholdt (he/him)
Oak Ridge National Laboratory

Software Sustainability track @ Argonne Training Program on Extreme-Scale Computing summer school

Contributors: David E. Bernholdt (ORNL), Anshu Dubey (ANL), Patricia A. Grubel (LANL), Rinku K. Gupta (ANL), Katherine M. Riley (ANL), David M. Rogers (ORNL), Gregory R. Watson (ORNL)



See slide 2 for
license details

License, Citation and Acknowledgements

License and Citation

- This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/) (CC BY 4.0).
- **The requested citation the overall tutorial is:** Anshu Dubey, David E. Bernholdt, and Todd Gamblin, Software Sustainability track, in Argonne Training Program on Extreme-Scale Computing, St. Charles, Illinois, 2025. DOI: [10.6084/m9.figshare.29816981](https://doi.org/10.6084/m9.figshare.29816981).
- Individual modules may be cited as *Speaker, Module Title, in Tutorial Title, ...*



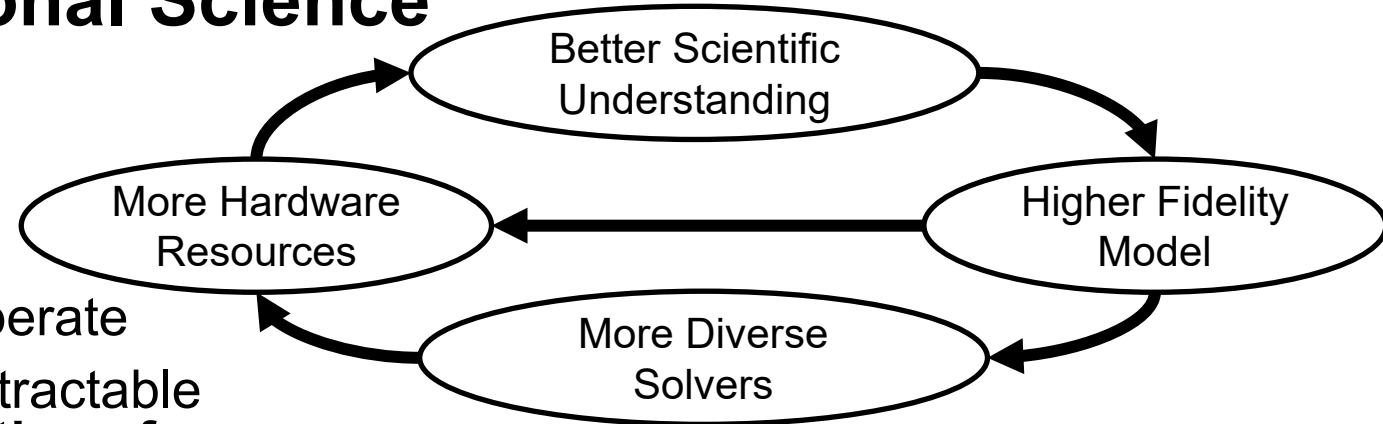
Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Next-Generation Scientific Software Technologies (NGSST) program.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No. 89233218CNA000001.
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Science through computing is,
at best,
as credible as the software that produces it!

The Success of Computational Science Creates the Challenges of Computational Science

- Positive feedback loop
 - More complex codes, simulations and analysis
 - More moving parts that need to interoperate
 - Variety of expertise needed – the only tractable development model is through **separation of concerns**
 - **It is more difficult to work on the same software in different roles without a software engineering process**
- Onset of higher platform heterogeneity
 - Requirements are unfolding, not known *a priori*
 - **The only safeguard is investing in flexible design and robust software engineering process**



Supercomputers change fast
Especially now!

Challenges Developing Scientific Applications Today

Technical

- All parts of the model and software system can be under research
- Requirements change throughout the lifecycle as knowledge grows
- Verification complicated by floating point representation
- Real world is messy, so is the software
- Increasing architectural diversity

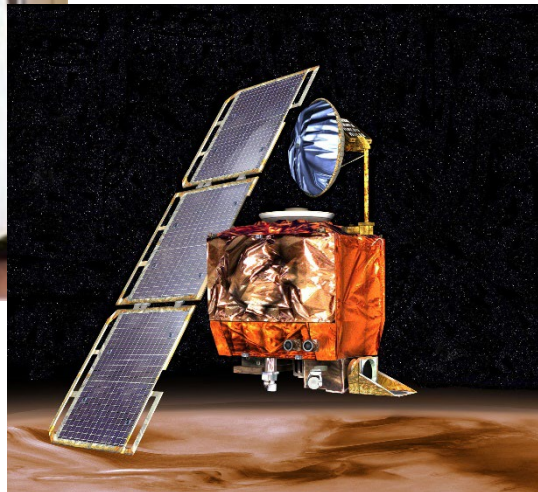
Sociological

- Competing priorities and incentives
 - Sponsors often care more about scientific publications than software per se
 - Balancing development and maintenance with research
- Limited resources
- Need for interdisciplinary interactions
 - Many different kinds of expertise required to be successful

High-Consequence Software-Related Scientific Failures

Therac-25 (1985-1987)

- Computer-controlled radiation therapy system
- **Poor software design, development and testing practices** allowed flaws that led to at least six cases of substantial radiation overdoses, three fatal



Mars Climate Orbiter (1999)

- Incorrect trajectory adjustment caused loss of the orbiter as it was supposed to enter Martian orbit
- Discrepancy in the units used in two different software components
- One component **didn't follow specifications**
- **Inadequate testing** at the interface
- Concerns raised earlier in the mission were ignored because they **weren't properly documented**

Just two of many examples

Software Under the Microscope

- Mar. 16 2020: Epidemiologist Neil Ferguson (Imperial College) briefed UK Parliament on computational modeling of COVID-19 pandemic
 - Epidemiological models like this helped prompt government action, but have lots of assumptions
- April 1 2020: Nicholas Lewis (independent climate science researcher in UK) can't easily see where some of the assumptions come from
 - publishes a blog article
 - “Moreover, the computer code... is old, unverified, and documented inadequately, if at all...”

<https://doi.org/10.25561/77482>

<https://www.nicholaslewis.org/imperial-college-uk-covid-19-numbers-dont-seem-to-add-up/>

<https://www.nature.com/articles/d41586-020-01003-6>

16 March 2020

Imperial College COVID-19 Response Team

Report 9: Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand

Neil M Ferguson, Daniel Laydon, Gemma Nedjati-Gilani, Sangeeta Bhatia, Adhiratha Boonyasiri, Zulma Cucunubá, Dorigatti, Han Fu, Katy Gaythorpe, Will Green, Arran Hall, Elsie Elvén, Hayley Thompson, Robert Verity, Erik Volz, Haowei Cao, Caroline Walters, Peter Winskill, Charles Whittaker, Christopher Jarvis

On behalf of the Imperial College COVID-19 Response Team

WHO Collaborating Centre for Infectious Disease Modelling
MRC Centre for Global Infectious Disease Analysis
Abdul Latif Jameel Institute for Disease and Emergency Analytics
Imperial College London

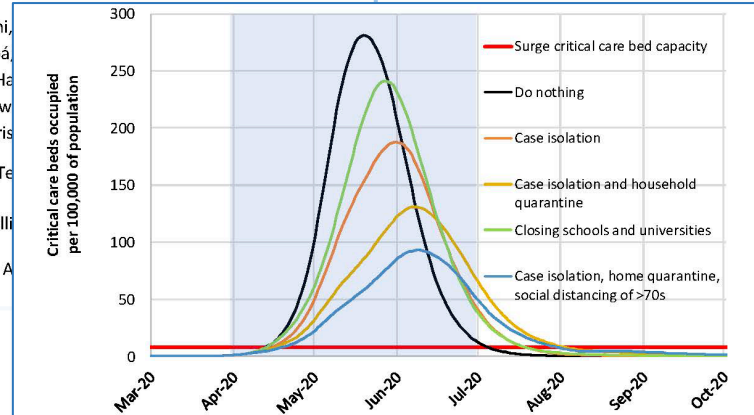


Figure 2: Mitigation strategy scenarios for GB showing critical care (ICU) bed requirements. The black line shows the unmitigated epidemic. The green line shows a mitigation strategy incorporating closure of schools and universities; orange line shows case isolation; yellow line shows case isolation and household quarantine; and the blue line shows case isolation, home quarantine and social distancing of those aged over 70. The blue shading shows the 3-month period in which these interventions are assumed to remain in place.

Imperial College UK COVID-19 numbers don't seem to add up

Introduction and summary

A study published two weeks ago by the COVID-19 Response Team from Imperial College (Ferguson20¹) appears to be largely responsible for driving UK government policy actions. The study is not peer reviewed; indeed, it seems not to have been externally reviewed at all. Moreover, the computer code used to produce the estimates in the study – which on Ferguson's own admission is old, unverified and documented inadequately, if at all – has still not been published. That, in my view, shows a worrying approach to a matter of vital public concern.

The Press Picks Up the Story

Headline and quotes from the Fox News article

“In our commercial reality, **we would fire anyone for developing code like this** and any business that relied on it to produce software for sale would likely go bust,” David Richards, co-founder of British data technology company WANdisco, told the Daily Telegraph.

“**Models must be capable of passing the basic scientific test of producing the same results given the same initial set of parameters**...otherwise, there is simply no way of knowing whether they will be reliable,” said Michael Bonsall, Professor of Mathematical Biology at Oxford University.

CORONAVIRUS · Published May 16

Imperial College model Britain used to justify lockdown a 'buggy mess', 'totally unreliable', experts claim

Scientists from the University of Edinburgh have further claimed that it is **impossible to reproduce** the same results from the same data using the model. The team got different results when they used different machines, and even different results from the same machines.

“There appears to be a bug in either the creation or re-use of the network file. If we attempt two completely identical runs, only varying in that the second should use the network file produced by the first, the results are quite different,” the Edinburgh researchers wrote on the Github file. A fix was provided, but it was **the first of many bugs** found within the program.

<https://www.foxnews.com/world/imperial-college-britain-coronavirus-lockdown-buggy-mess-unreliable>

<https://www.telegraph.co.uk/technology/2020/05/16/coding-led-lockdown-totally-unreliable-buggy-mess-say-experts/>

What you May Not Have Heard

- April 22 2020: Imperial collaborates with Microsoft to refactor and clean up the code, which is released on GitHub
- May 10 2020: Phil Bull rebuts criticisms of the Imperial code
 - Which spurs further discussions within some groups focused on scientific software
- May 29 2020: CODECHECK independently reproduces results of Imperial's Report 9


<https://github.com/mrc-ide/covid-sim/>
<https://philbull.wordpress.com/2020/05/10/why-you-can-ignore-reviews-of-scientific-code-by-commercial-software-developers/amp/>
<http://doi.org/10.5281/zenodo.3865491>

Many scientists write code that is crappy stylistically, but which is nevertheless scientifically correct (following rigorous checking/validation of outputs etc). Professional commercial software developers are well-qualified to review code style, but most don't have a clue about checking scientific validity or what counts as good scientific practice. Criticisms of the Imperial Covid-Sim model from some of the latter are overstated at best.

CODECHECK certificate 2020-010

<https://doi.org/10.5281/zenodo.3865491>



Item	Value
Title	Report 9: Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand. March 16, 2020.
Authors	Neil Ferguson  , COVID-19 Response Team
Reference	Imperial College Preprint https://doi.org/10.25561/77482

Some Observations

- Your code is likely to live longer than you expect ...
and may be used in ways you don't expect ...
by people you don't know ...

Plan for it!

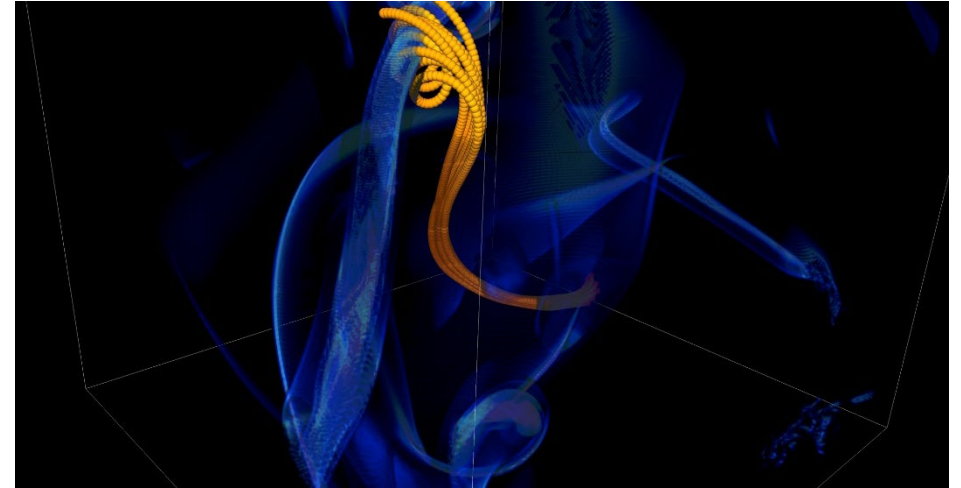
- Increasingly, consequential decisions are made based on computational results
 - The codes generating those results may (justifiably) be subject to greater scrutiny
- The scientific credibility of software is strongly connected to good software engineering practices
 - Documentation
 - Testing, verification, and (where possible) validation
 - Code readability and quality metrics

Question: Should we excuse scientific software for being “crappy stylistically”?

Hint: crappy code can hide bugs

More Subtle Impacts on Scientific Productivity

- In 2005, the FLASH astrophysics team was offered a unique opportunity to access one of the biggest machines in the world at that time (BG/L) for a dedicated run
- Short notice to prepare
 - **< 1 month to get ready for 1.5 week run**
- Quick and dirty development of particle capability in code
- Error in tracking particles resulted in duplicated tags from round-off
- Had to develop post-processing tools to correctly identify trajectories
 - **6 months to process results**



FLASH had a software process in place. It was tested regularly. This was one instance when the full process could not be applied because of time constraints.

Technical Debt

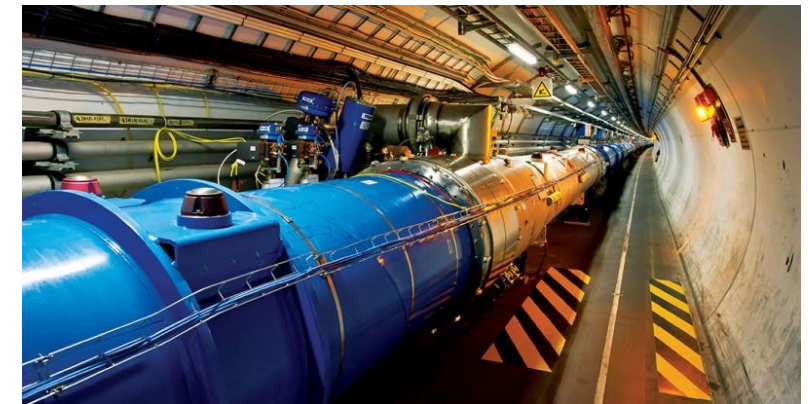
The implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.
-- Wikipedia

Like monetary debt, the more you accumulate, the harder it is to pay off

- Increases cost of maintenance
- Parts of software may become unusable over time
- Inadequately verified software produces questionable results
- Increases ramp-up time for new developers
- **Overall, reduces software and science productivity**

Scientific Facilities Provide Valuable Resources

- Major supercomputers often cost O(\$100M)
- All cost millions more to operate, annually
- Significant allocations on large supercomputers can be worth millions
- Even if you don't pay the \$ you have to spend the time and effort to get the allocation
- *Alternative scenario:* someone else is using an experimental user facility, being guided by your simulations
- **Sponsors' concern: Are you being a good steward of the resources?**
- **Your concern: Are you getting the most science possible out of your work (aka scientific productivity)?**



**Good scientific process
requires
good software practices**



**Good software practices
increase
software sustainability**



**Good software practices
increase
scientific productivity**



**Software sustainability
increases
scientific productivity**

So, What Are Good Software Practices?

- There is no fixed, universally agreed set of best practices for scientific software
 - Specifics of what's appropriate will depend on the software, how it is used, and the team
- Let's look at a few recommendations from different perspectives...

Example 1: Best Practices for Scientific Computing (1/2)

Wilson, et al., (2014) <https://doi.org/10.1371/journal.pbio.1001745>

1. **Write programs for people, not computers.**
 - a. A program should not require its readers to hold more than a handful of facts in memory at once.
 - b. Make names consistent, distinctive, and meaningful.
 - c. Make code style and formatting consistent.
2. **Let the computer do the work.**
 - a. Make the computer repeat tasks.
 - b. Save recent commands in a file for re-use.
 - c. Use a build tool to automate workflows.
3. **Make incremental changes.**
 - a. Work in small steps with frequent feedback and course correction.
 - b. Use a version control system.
 - c. Put everything that has been created manually in version control.
4. **Don't repeat yourself (or others).**
 - a. Every piece of data must have a single authoritative representation in the system.
 - b. Modularize code rather than copying and pasting.
 - c. Re-use code instead of rewriting it.
5. **Plan for mistakes.**
 - a. Add assertions to programs to check their operation.
 - b. Use an off-the-shelf unit testing library.
 - c. Turn bugs into test cases.
 - d. Use a symbolic debugger.

Example 1: Best Practices for Scientific Computing (2/2)

Wilson, et al., (2014) <https://doi.org/10.1371/journal.pbio.1001745>

6. **Optimize software only after it works correctly.**
 - a. Use a profiler to identify bottlenecks.
 - b. Write code in the highest-level language possible.
7. **Document design and purpose, not mechanics.**
 - a. Document interfaces and reasons, not implementations.
 - b. Refactor code in preference to explaining how it works.
 - c. Embed the documentation for a piece of software in that software.
8. **Collaborate.**
 - a. Use pre-merge code reviews.
 - b. Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
 - c. Use an issue tracking tool.

Example 2: Good Enough Practices in Scientific Computing (1/2)

Wilson, et al., (2017) <https://doi.org/10.1371/journal.pcbi.1005510>

1. Data management

- a. Save the raw data.
- b. Ensure that raw data are backed up in more than one location.
- c. Create the data you wish to see in the world.
- d. Create analysis-friendly data.
- e. Record all the steps used to process data.
- f. Anticipate the need to use multiple tables, and use a unique identifier for every record.
- g. Submit data to a reputable DOI-issuing repository so that others can access and cite it.

6. Manuscripts (out of order to save space)

- a. Write manuscripts using online tools with rich formatting, change tracking, and reference management.
- b. Write the manuscript in a plain text format that permits version control.

2. Software

- a. Place a brief explanatory comment at the start of every program.
- b. Decompose programs into functions.
- c. Be ruthless about eliminating duplication.
- d. Always search for well-maintained software libraries that do what you need.
- e. Test libraries before relying on them.
- f. Give functions and variables meaningful names.
- g. Make dependencies and requirements explicit.
- h. Do not comment and uncomment sections of code to control a program's behavior.
- i. Provide a simple example or test data set.
- j. Submit code to a reputable DOI-issuing repository.

Example 2: Good Enough Practices in Scientific Computing (2/2)

Wilson, et al., (2017) <https://doi.org/10.1371/journal.pcbi.1005510>

3. Collaboration

- a. Create an overview of your project.
- b. Create a shared "to-do" list for the project.
- c. Decide on communication strategies.
- d. Make the license explicit.
- e. Make the project citable.

4. Project organization

- a. Put each project in its own directory, which is named after the project.
- b. Put text documents associated with the project in the doc directory.
- c. Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory.
- d. Put project source code in the src directory.
- e. Put external scripts or compiled programs in the bin directory.
- f. Name all files to reflect their content or function.

5. Keeping track of changes

- a. Back up (almost) everything created by a human being as soon as it is created.
- b. Keep changes small.
- c. Share changes frequently.
- d. Create, maintain, and use a checklist for saving and sharing changes to the project.
- e. Store each project in a folder that is mirrored off the researcher's working machine.
- f. Add a file called CHANGELOG.txt to the project's docs subfolder.
- g. Copy the entire project whenever a significant change has been made.
- h. Use a version control system.

Example 3: OpenSSF Best Practices Badge Program

<https://www.bestpractices.dev/en>

- Not specifically intended for scientific software
 - OpenSSF = Open Source Security Foundation
- Three levels
 - **Passing** focuses on best practices that well-run FLOSS projects typically already follow. Getting the passing badge is an achievement; at any one time only about 10% of projects pursuing a badge achieve the passing level.
 - **Silver** is a more stringent set of criteria than passing but is expected to be achievable by small and single-organization projects.
 - **Gold** is even more stringent than silver and includes criteria that are not achievable by small or single-organization projects.
- Combination of MUST and SHOULD criteria

OpenSSF Best Practices Criteria Summary

- **Basics**

- Basic project website content (P, S)
- FLOSS license (P)
- Documentation (P, S)
- Project oversight (S, G)
- Accessibility and internationalization (S)

- **Change control**

- Public version controlled source repo. (P, G)
- Unique version numbering (P)
- Release notes (P)
- Previous versions (S)

- **Reporting**

- Bug-reporting process (P, S)
- Vulnerability reporting process (P, S)

(P, S, G) denotes additional criteria required at passing, silver, or gold certification levels

Each topic area listed will have one or more specific criteria

- **Quality**

- Working build system (P, S, G)
- Automated test suite (P, S, G)
- New functionality testing (P, S)
- Warning flags (P, S)
- Coding standards (S, G)
- Installation system (S)
- Externally-maintained components (S)

- **Security**

- Secure development knowledge (P, S)
- Use basic good crypto. practices (P, S, G)
- Secured delivery against MITM attacks (P, G)
- Publicly known vulnerabilities fixed (P)
- Secure release (S)

- **Analysis**

- Static code analysis (P, S)
- Dynamic code analysis (P, S, G)

Example 4: Good Practices for High-Quality Scientific Computing

Anshu Dubey (2022) <https://doi.org/10.1109/MCSE.2023.3259259>

Good Practices for Software Development

- **Invest in design**
 - Create the framework into which all of the capabilities you need will fit naturally
- **Invest in development of tests and automated testing**
 - How can you convince yourself that the code is correct...after every change?
- **Capture and summarize the thought process**
 - Document the reasoning behind the design and other choices
- **Use optimal rigor in your software process**
 - Too little can compromise code quality
 - Too much can reduce developer productivity
- **Pick and choose from software engineering research**
 - Often requires adaptation for scientific software

We'll expand on some of these points in the coming slides...

Good Practices for Using Software

- **Understand the capabilities and limitations**
 - Is your science case breaking assumptions built into the code?
 - How would you know?
- **Know the parameters**
 - Are you using the knobs correctly?
- **Practice rigorous verification and validation for each new science use**
 - New science cases push your code in new ways
- **Develop a methodology for capturing, analyzing, and archiving results**
- **Fork code for science use**
 - Separate new development from supporting the science runs

We'll return to some of these points in the module on reproducibility...

Software Engineering Advice Often Needs Adaptation for Scientific Software

- The OpenSSF Best Practices are a good example of software engineering advice “in the wild”
- Experiences reported in the wild often don’t consider the special nature of scientific software
- But that doesn’t mean we should ignore software engineering research experience
 - Many useful concepts, approaches, and tools we can just adopt
- Some approaches may need to be adapted to work for scientific software
 - Find out how colleagues have addressed the challenges you’re facing
 - Probably you will find multiple ways
 - In the end, some approaches may not work well
- Don’t be afraid to experiment with adaptations
 - Consider using the PSIP process (coming up)

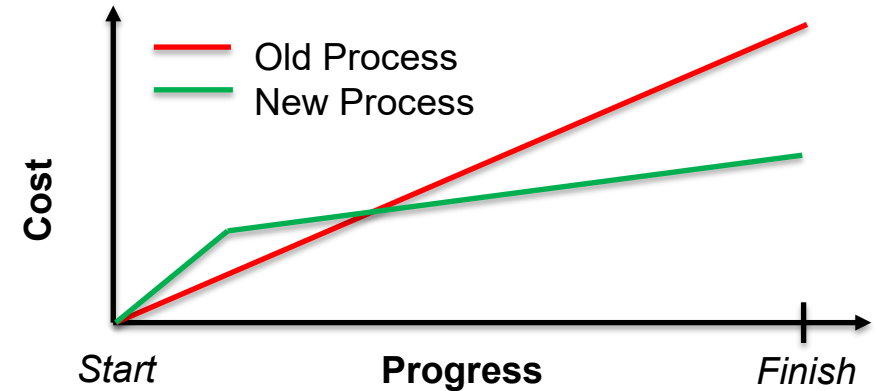
How Much (Time, Effort) Should I Spend on Software Engineering?

- Your project should include “just enough” software engineering so that you can meet your short-term and longer-term scientific goals effectively
- Take a graduated approach
 - Early on, maybe your project is exploratory and you’re the only user (and developer), basic practices may suffice
 - As the project gains developers and/or users, give more thought to software engineering and adjust practices
 - As the science being done with the code gains greater visibility, give more thought to software engineering and adjust practices
 - If you anticipate the software being used to make consequential decisions, probably better to consider “more” software engineering from the start
 - Make a practice of re-evaluating your software engineering needs on a regular basis
 - Changes may creep up on you without you noticing
 - Consider putting an annual reminder on your calendar (seriously!)
 - Plus, whenever significant changes take place

Continual, Incremental Software Process Improvement

Target: your project should include “just enough” software engineering so that you can meet your short-term and longer-term scientific goals effectively

1. Identify your team’s “pain points” in your software development processes
 - Help: RateYourProject assessment tool:
<https://rateyourproject.org/>
 2. Set a goal for something to improve
 - Target processes and behaviors, not just tasks
 - Pick something that you can address in a few months that will give you a noticeable benefit
 3. Agree on a plan to address it, identify markers of progress and what is “done”
 - Write them down
 - Help: Progress tracking card examples:
<https://bssw-psip.github.io/ptc-catalog/catalog>
 4. Work your plan, track your progress
 5. When you are done, celebrate...
- ...then pick a new pain point to address



The new process costs something to implement, but it pays off over time

Productivity and Sustainability Improvement Planning
<https://bssw.io/psip>



A goal of [BSSw.io](https://bssw.io) is to provide resources for improving your software processes. If you find useful resources that aren't on BSSw.io, consider contributing. Its easy and quick.

About Today's Tutorial

- There are many useful topics that could help you improve your scientific software development process
- We're going to focus on a few topics where the software engineering advice in the wild typically doesn't address scientific software
 - Designing software for flexibility and extensibility
 - Software packaging for HPC environments
 - Testing strategies for complex high-performance computing software
 - Code refactoring
 - Considerations for licensing your software
 - Reproducibility