OXFORD

The Computer Journal, 2024, **67**, 1381–1404
https://doi.org/10.1093/comjnl/bxad067
Advance access publication date 23 August 2023
Articles

Improving Science That Uses Code

Harold Thimbleby*

See Change Fellow in Digital Health, Swansea University
*Corresponding author: harold@thimbleby.net

As code is now an inextricable part of science it should be supported by competent Software Engineering, analogously to statistical claims being properly supported by competent statistics.

If and when code avoids adequate scrutiny, science becomes unreliable and unverifiable because results — text, data, graphs, images, *etc* — depend on untrustworthy code.

Currently, scientists rarely assure the quality of the code they rely on, and rarely make it accessible for scrutiny. Even when available, scientists rarely provide adequate documentation to understand or use it reliably.
This paper proposes and justifies ways to improve science using code:

1. **Professional Software Engineers can help, particularly in critical fields such as public health, climate change and energy.**
2. **'Software Engineering Boards,' analogous to Ethics or Institutional Review Boards, should be instigated and used.**
3. **The Reproducible Analytic Pipeline (RAP) methodology can be generalized to cover code and Software Engineering methodologies, in a generalization this paper introduces called RAP+. RAP+ (or comparable interventions) could be supported and or even required in journal, conference and funding body policies.**

The paper's Supplemental Material provides a summary of Software Engineering best practice relevant to scientific research, including further suggestions for RAP+ workflows.

'Science is what we understand well enough to explain to a computer.' Donald E. Knuth in $A = B$ **[1]**
'I have to write to discover what I am doing.' Flannery O'Connor, quoted in *Write for your life* [2]
'Criticism is the mother of methodology.' Robert P. Abelson in *Statistics as Principled Argument* [3]
'From its earliest times, science has operated by being open and transparent about methods and evidence, regardless of which technology has been in vogue.' Editorial in *Nature* [4]

Keywords: Computational Science; Software Engineering; reproducibility; scientific scrutiny; reproducible analytic pipeline (RAP & RAP+)

1. INTRODUCTION

Unreliable, often unstated and unexplored, code and computational dependencies (including using AI or ML) in science are widespread. Furthermore, code is rarely published or made accessible in a usable form; it is generally too onerous or impossible to verify or scrutinize. Ironically, computers should be able to make reproducibility easier, yet too often code and results claimed do not contribute to reliable science, and do not support verification, replication or reproduction on which future science can be firmly based.

This paper makes an explicit analogy between the use of statistics, which has clear standards for reliable use and presentation, and the use of code. Like statistics, code is often relied on to support key results in scientific publications, yet code is generally informal, inaccessible and incorrect. Just as sound experimental methods and sound statistics generally rely on professional specialist input, it is argued here that good use of code must rely on professional Software Engineering input [5, 6], and more strategically on professional Computational Thinking [7, 8] — an accessible form of Software Engineering that consciously applies the ideas universally, far more widely than just to software and coding.

This paper was initially motivated by concerns about the poor quality of code used in high-profile epidemiology research because of its significance for informing and driving public health responses to the coronavirus disease 2019 (COVID-19) pandemic. A pilot survey implies that such problems are ubiquitous and by no means limited to epidemiology: see Tables 1 and 2 for a summary of the sample, and see the Supplemental Material for further details of the survey. In the survey, *no* papers claimed or provided evidence that their code was adequately tested or rigorously developed; *none* used methodologies like RAP or RAP+ (described below). Only one paper mentioned any Software Engineering methods, albeit simplistic and without technical details.

In the survey sample, 81% of papers were published in leading journals that have code policies (which themselves are weak), but 42% of surveyed papers published in those journals breached their own policies. One paper declared it had accessible code, but the relevant repository was and still remains empty. The findings are comparable to problems increasingly recognized for data and data access (reviewed in Section 2.1): code problems form part of the *reproducibility crisis* [e.g. 12, 13] discussed throughout this paper.

This paper therefore argues that code should be developed and discussed in a professional, rigorous, and supportive

Received: June 8, 2022. **Revised:** May 12, 2023. **Accepted:** May 24, 2023

Table 1. Overview of peer-reviewed paper sample, broken down further in Table 2. Survey methodology and data are provided in the Supplemental Material. (The survey does not include the motivating papers [9–11], none of which provide code or code summaries; see section 5.1.)

3	**Journals**
32	**Papers:**
	6 *Lancet Digital Health*
	12 *Nature Digital Medicine*
	14 *Royal Society Open Science*
264	Published authors
341	Published journal pages
July 2020	Sample month

environment that facilitates quality science with clear presentation and appropriately rigorous scrutiny of code. Its main contribution is to suggest straightforward ways to enable this. The proposals may not be 'the' right or best ways, but it is hoped the case studies and arguments presented here persuade readers that the proposals are at least a productive way to start pointing in the right direction, and to inform raising the profile and constructively debating the issues more widely.

An extensive online Supplemental Material appendix to this paper provides additional resources, including brief details of many Software Engineering practices relevant to supporting quality science. The supplement will be of particular interest to research software engineers supporting non-software-specialist scientists.

2. BACKGROUND

The discoveries and inventions of scientific technologies and instruments like microscopes, telescopes and X-rays, drove and expanded the sciences. There are fascinating periods when new ideas and science unified; for example, thermometers could not measure temperature in any meaningful way until the underlying science was mature. For over a century, there was no agreement on definitions of temperature, how to calibrate thermometers, or what units they measured in.

Paradoxically the science could not mature until there was consensus in scientific methodologies for thermometry, and having that consensus in turn depended on reproducible thermometer measurements; for example, scientists working in different places needed to know they were working with the 'same temperatures' yet there was for a long period no consensus on what that meant. Contributing to reliable science depended on a thorough understanding of principles, including gradually fixing the confounding factors that were misunderstood [14].[1] Science matured from no quantitative interest in temperature, through a complex process of hand-in-hand theoretical-and-technical maturation, until today, when we have robust off-the-shelf instruments that measure temperature in reliable, repeatable, internationally standardized units, that follow international quality standards.

Computers are a unique, new technology, far more flexible and challenging than thermometers. Understanding computers and integrating them into science is far harder than the tortured development of modern thermometry. Computation not only expands science's paradigms and supports new discoveries (particularly with AI), but it also *does* new science — almost all modern laboratory instrumentation, including thermometers, is heavily computerized. Relying on computer models has become routine. The dependency of modern science on computers is far more tangled and complex than the now-resolved dependence on reliable temperature measurement; computers affect how scientists in all disciplines *think*.

Pushing the boundaries of science, then, now involves pushing the boundaries of computer science. The synergy runs deep: for instance, while particle physics relies on powerful supercomputers, quantum physics itself is developing more powerful quantum computing.

'Computational science' has come to mean a particular style of science based on developing and using explicit computational models, but, really, *all* of science is now computational in this sense.

Computational science is not just restricted to specialized fields like computational chemistry, genomics, big data … in all fields of science, computation is used at every step, from calculations of course, through note taking, sound and image processing, literature searches, analysis and statistics, correspondence with co-authors and editors, through to typesetting, distributing and archiving the final publications. All areas of science are being profoundly computerized. Furthermore, developments in computer science themselves drive science such that earlier science is even becoming obsolete as the computer technology moves on [15].

2.1. Code quality concerns

Publishing high quality computer code has been strongly advocated since the earliest times, such as the *Communications of the ACM* in its first issue in its first volume published in 1958, where it outlined its algorithm publication policy. The new policy was illustrated with a square root algorithm [16]. However, publishing code in the computer science literature is distinct from publishing high quality general science that depends on code, which is the particular concern of the present paper. Of course, as a special case, Computer Science too can also benefit from improving ways to reliably use code in general science.

In almost all published science, the code it relies on is taken for granted, just as in routine chemistry the quality of the glassware is not at issue. While chemists are trained in reliable methodologies, so taking quality glassware for granted is reasonable. Code is a newer innovation, and quality code is a current research programme in its own right, and has resulted in calls for a Grand Challenge research effort [17]. Inevitably, because of these reasons taken collectively, much published science depends on unreliable code that is not explicitly discussed, was not peer-reviewed, and is not open to scrutiny, reproduction or reuse by the scientific community. The situation with code is analogous to chemists using contaminated glassware with no awareness how its effects might be controlled or might affect results.

Is it a problem? A study of 863 878 Python-coded Jupyter notebooks [18] found a 76% failure rate for code to complete execution successfully. Trisovic *et al.* [19] performed a study of 9000 research codes written in the language R on Dataverse, an open-source repository maintained by Harvard University's Institute for Quantitative Social Sciences. They found a comparable result that 74% of the code files analyzed failed. These results are consistent with this paper's findings, as summarized in Tables 1 and 2.

The authors of [18] make technical recommendations to improve reproducibility, such as 'Abstract code into functions, classes and modules and test them.' It will not be obvious for most practicing scientists how to do this, so, more generally,

[1] One example: if the volume of mercury is chosen to measure temperature, a confounding factor is that the volume of the container measuring the volume of mercury also increases with temperature (but at a different rate), so the volume measurement is inaccurate.

Table 2. Breakdown of pilot survey of peer-reviewed science papers relying on code.

Number of papers sampled relying on code	32	100%	
Access to code			
Some or all code available	12	38%	
Some or all code in principle available on request	8	25%	
Requested code actually made available (within 2 years 11 months*)	0	0%	
Evidence of any software engineering practice			
Evidence program designed rigorously	0	0%	
Evidence source code properly tested	0	0%	
Evidence of any tool-based development	0	0%	
Team or open source based development	0	0%	
Other methods, e.g. independent coding methods	1	3%	
Documentation and comments			
Substantial code documentation and comments	2	6%	
Comments explain some code intent	3	9%	
Procedural comments (e.g. author, date, copyright)	10	31%	
No usable comments	17	53%	
Repository use			
Used code repository (e.g. GitHub)	9	28%	
Used data repository (e.g. Dryad or GitHub)	9	28%	
Empty repository	1	3%	
Evidence of documented processes			
Evidence of RAP/RAP+ or any other principles in use to support scrutiny	0	0%	
Adherence to journal code policy (if any)			
Papers published in journals with code policies	26	81%	
Clear breaches of journal code policy (if any)	11	42%	(N = 26)

*Time of 2 years 11 months is wait between code request and date of generating this table.

Table 3. The TOP committee's recommended levels for journal article code transparency. Level 0 is provided for a comparison that does not meet any TOP requirements. Concerns about the interpretation of "reproduced independently," as required at level 4, are raised in section 5.2.

Level 0	Level 1	Level 2	Level 3
Journal encourages code sharing – or says nothing.	Article states whether code is available and, if so, where to access them.	Code must be posted to a trusted repository. Exceptions must be identified at article submission.	Code must be posted to a trusted repository, and reported analyses will be reproduced independently before publication.

the authors of [19] recommend establishing Working Groups to support reproducible research — and in fact Dataverse (the source of the [19] data) has already done so. These ideas may be compared to the present paper's proposal of Software Engineering Boards (SEBs), as discussed in detail in Section 6, supported by more specific suggestions in the Supplemental Material.

These concerns about code are part of the reproducibility crisis for science generally [12, 20–23]. Concern has led to new journals, *Journal of Open Source Software* (JOSS) [24], *ReScience C* [13] and others, to explore and encourage the explicit replication of previously published research.

To start to address code quality issues the Transparency and Openness Promotion Committee met in 2014, and has since been promoting Transparency and Openness Promotion, TOP, starting with journal publication policies [25]. TOP covers citation standards, replication standards and code standards amongst others. TOP recommends levels of compliance to their recommendations, where level 0 does not meet the standard, and levels 1–3 are increasingly stringent. The TOP levels for code are shown in Table 3. The TOP standards continue to develop, and are now maintained on a wiki at osf.io/9f6gx/wiki/Guidelines [26]. TOP can be compared with the Findable, Accessible, Interoperable and Reusable (FAIR) initiative, which places greater emphasis on data rather than code; FAIR itself is critiqued in Section 5.4.

As the present paper argues, developing quality code is widely under-appreciated, which leads to a vicious cycle of lack of acknowledgement, invisibility, and being unable to recruit adequately competent coders (see Section 3). The term *research software engineer* was coined in 2012 to help address this problem, and to stimulate thinking about researchers' career paths. The Society of Research Software Engineering (society-rse.org) has been established to further promote research software engineer interests.

There is no shortage of computational tools available to help address the problems. However, it should be noted that such tools are not a panacea, as the study [18] cited above makes clear. This suggests that human support for improving coding and reproduction quality, perhaps in the form of Working Groups or Boards, as this paper suggests, will be critical.

In addition to unintentional problems with code quality and reproducibility, actual scientific misconduct occurs when the outcome is intentional. While pure plagiarism, which is a form of misconduct, generally does not affect the quality of reported science, when data or code is fraudulently manipulated to have deceptive properties, the results are likely to be destructive. The notorious Wakefield MMR fraud claiming to link vaccines and autism published in *The Lancet* took 12 years before it was retracted; this misconduct has been extraordinarily destructive [27].

A recent meta-analysis of surveys of scientific misconduct estimates that nearly 2% of scientists have at least once fabricated, falsified or modified results, and over a third have undertaken other questionable research practices [28]. The meta-analysis qualifies the figures carefully, but these are alarming rates regardless of the qualifications; indeed, the authors suggest that as misconduct is a sensitive issue, the rates are likely to be underestimates.[2]

While technical solutions like using AI may help, it is notable that many misconduct issues can be detected and constructively managed prior to publication using exactly the same methods as will improve research reproducibility, as discussed throughout the present paper.

2.2. Computable papers

'Electronic lab notebooks' (ELNs) [29], emphasize computer tools that specifically support laboratory notebook authoring and editing. In contrast, so-called 'computable papers' aim to support the scientific paper authoring workflow: the emphasis is that papers should produce faithful results from code embedded in (or easily accessible from) the text of the paper.

If, for example, HTML is being used for a computational paper, the paper's text could include Javascript code like

```
<script>
  document.write(responses.total)
</script>
```

This illustrative code would insert the result of running it into the paper, perhaps like 'We collected data and obtained 754 responses,' where the 754 is the value of the variable `responses.total` when the paper was formatted. The point is that the number 754 (or whatever) is computed automatically from the data, so any changes or improvements to the data or the methodology analyzing it will translate into the published number being updated and inserted into the paper at the appropriate point.

Systems like LaTeX (which was used for the current paper) can combine advanced typesetting with computable paper calculations. Here, as a simple example, we calculate that 10! is equal to 3 628 800 just by writing the LaTeX code `\factorial{10}` *in* this paper, which can calculate arbitrary factorials and group the result into conventional blocks of three digits by inserting small spaces. Note that although `factorial` is not a standard LaTeX function, it was readily defined by code also *in* the present paper, so it can be easily modified by the author for other purposes as required.

More practically for helping author complex scientific papers, a separate program can generate a file of LaTeX definitions for data, tables, and cross-references, etc, as needed for a paper. Here is a small example, where each generated fact (taken from the present paper's analysis) has been highlighted in **bold**:

> For the present paper, as a concrete example of this approach, the pilot survey analyzed **32** papers with **264** authors. The paper [9], discussed below in Section 5.1, relied on code composed of **229** files, with over **25 thousand** lines of code (how this compares with files in the survey is summarized in table **9** in the Supplemental Material).

In the 'old days' such numbers (32; 264; etc) quoted above would have been manually worked out, eyeballed, typed up and then (hopefully) double-checked by the authors. Instead, in this paper, those numbers (and many others) were computed automatically, and were then inserted into the text of this paper automatically. They are auditable back to their sources. Furthermore, they will update automatically — with no further work from the author — when the data or calculations change.

The idea is easy to implement. Continuing using LaTeX as the illustrative word processing system, code could generate text like

```
\newcommand{\numberOfAuthors}{264}
```

where the 264 is some number calculated from the relevant data.

Such a line of text is then saved to a file, which is then imported into the paper so the value is named and can be used easily and reliably. Then, when and wherever the authors write the name `\numberOfAuthors` in their paper's text, the typeset paper says 264, or whatever the actual value is at the time — it will update automatically whenever the data or analysis is improved. Some easy LaTeX coding can then present the numbers in the author's or publisher's preferred style, such as zero, one,... nine, 10, 11,..., 1 000, 1 001, etc, as done more realistically elsewhere in this paper (for instance, see footnote 5).

The key concept in computable papers is that as the authors of a paper collect or revise data or calculations or otherwise update it, the results written up in the paper *also* update automatically, and all the statistics, graphs, and analysis reported in the paper update and remain correct *with no further work from the authors*. Indeed, if an author corrects a mistake, the correction will apply automatically to all future revisions.

However, there is no structure to using general-purpose systems like HTML, LaTeX or Rmarkdown, which means authors may make unnoticed mistakes. Many authors therefore prefer a more structured approach imposed by tools designed for the purpose, where their structure helps impose uniformity and helps prevent and check for errors.

A taut review of such systems is Perkel [30], whereas [31] discusses in-depth the design trade-offs of one powerful approach, Maneage; the paper [31] includes a substantial and useful literature review (in its appendix). Here, four representative tools (WEB, *Mathematica*, Jupyter and knitr) serve to sample the variety of approaches that are available:

- WEB is the earliest tool reviewed here. WEB was developed by Donald Knuth in 1984 [32, 33] as a batch (non-interactive) tool to support his then radical new concept of *literate programming*. The idea was to facilitate programmers write literate documentation for their code. WEB combines a sequential documentation file with code that can be presented in any order, thus overcoming the problem that the best explanation of a program is not necessarily written in the same order as the code it explains. The original WEB allowed Pascal programs to be documented in TE X, but many variants of WEB have since been developed that are more flexible in the systems they support.
  WEB documents an entire program, but there are variants such as relit [34] that allow arbitrary parts of programs to be documented, and hence are useful for normal scientific papers that need to explain algorithms, but do not need to show or explain the entire code required for computer execution.
  In contrast to the other tools reviewed here, literate programming is intended to produce high quality publications *about* code, rather than publications just *using* code, inserting output, such as statistics or graphs, generated by running it.
- *Mathematica* is one of the earliest fully interactive notebook tools for computational papers. Notebooks were developed by Theodore Gray in 1988 [35]. Notebooks consist of a collection

[2] The present paper's author's own survey of scientists publishing in the *Journal of Machine Learning* had comparable results [20].

of ‘cells,’ where cells can be labelled as text or as a section heading, but if a cell is labelled as code, it can be run and it will normally generate a new cell following it as its output. The new cell can be numbers, tables, mathematics, a plot, an image, or even more text — the arbitrary output of running code, in fact. Moreover, a notebook itself is a *Mathematica* expression, so it — the paper itself — can be analyzed or manipulated by code in any way. The entire notebook structure and contents can be checked for consistency and correctness in any way the author chooses.

A *Mathematica* notebook can be published directly as a paper, but some code might be distracting for a publication. Typically the author therefore optionally hides some or all code cells that are irrelevant to the narrative of the paper, but which nonetheless were required to generate the results presented.

The user manual for *Mathematica* itself was written as a *Mathematica* notebook, which ensured all its examples actually worked — and probably helped ensure the correctness of the *Mathematica* code used behind the scenes (see Section 6.3). The book is now the largest example of software documentation in existence: in its latest edition it runs to over 10 000 pages.

- Jupyter was developed by Fernando Pérez and Brian Granger [36], and takes a similar approach to *Mathematica*, but Jupyter is open-source and not closely integrated with any particular programming language, as *Mathematica* is. Jupyter can be installed on a local computer or run over the web.
- Jupyter is both an authoring tool and a framework on which to build other tools: thus Google’s colaboratory is built on top of Jupyter, using it as a foundation but making stylistic changes, including providing free computational resources.
- Jupyter is very popular and widely-known. Many extensive examples of using Jupyter notebooks (and other good practice, such as using repositories) to support large scale science projects can be found at the Gravitational Wave Open Science Center at www.gw-openscience.org.
- knitr [37] is a powerful culmination of a variety of tools, Pweave, Sweave and ideas from literate programming. Knitr combines a markdown document with R code, and is a more powerful approach than the analogous, and perhaps more familiar, HTML+Javascript example shown above to motivate this section.

Thimbleby [38] is a 1999 example of a peer-reviewed paper (about user interface design) written as a *Mathematica* notebook, which makes the point that a distinctive feature of its methodology is that the *Mathematica* notebook creates a fully inspectable and replicable process. The notebook is available on the author’s web site; it can be checked by others, or easily extended or repurposed to support new research — and it still works 24 years later.[3]

There are many tools to make using code more convenient and more reproducible, as this section briefly reviewed, but unfortunately they are rarely used or used haphazardly, as the next section shows.

2.3. RAP: reproducible analytical pipelines

Writing a paper typically starts with a word processor (such as Microsoft Word or LaTeX), sketching an outline, writing boiler-plate text (such as the authors’ names and standard section headings), and then gradually building up the evidence base (including citing the literature) that the paper relies on. This workflow will be concurrent with many other activities — grant writing, writing up lab books, negotiating authorship, protecting IP, workshops, finding publication outlets, and so on.

Table 4 illustrates the core pipeline of how experiments and data are used to provide information on which analysis and calculations are based, the results of which are then collected and edited into a paper.

For clarity, the schematic pipeline in table 4 omits many steps in the creative scientific workflow. Furthermore, each step is iterated and modified as the research progresses, and, indeed, as referees require revision. The point is that in typical scientific practice each step in the table is largely or entirely manual, typically selecting and copying output from the previous phase, and then pasting and editing the results into the next. The pipeline of data $\rightarrow \cdots \rightarrow$ paper is then iterated by hand as the various components are refined and improved until the authors (and funders, editors, and referees) are happy with the final paper.

As problems are found in a paper, the data, calculations and code are debugged, refactored and refined. The workflow is rarely systematic, and even less likely to be documented — after all, the atomic steps seem to be innocuous copy and paste actions. The final paper and the ideas it embodies are what matters.

The insight of the reproducible analytic pipelines (RAP) proponents is that every time any step in the pipeline is performed it could have been automated [40–42]. If automated, it could then be repeated reliably — unlike a manual cut and paste which is potentially different and certainly error-prone every time it is performed. If automated, any part of the workflow can be reliably repeated if any experimental data, literature or other knowledge changes; the paper’s analysis will brought up to date with ease. In particular, any other researcher, whether part of the authorship team or a later reader of the paper, can reproduce the paper and its results reliably provided that the RAP workflow is made available. Table 5 provides a brief summary of RAP principles.

For example, if the paper in question is a systematic review, it could be kept current by automatically re-running the programmed atomic actions that it was built with. Indeed, this ability is one of the original motivations of RAP, so Government agencies could easily generate up to date reports on request without having to repeat all the manual work, and risk making procedural errors doing so. Furthermore, every time a publication is re-derived or updated, the RAP pipeline itself is reviewed and improved, so the quality of the reproduced work improves — unlike in a non-RAP workflow where new errors are potentially introduced every time a work is revisited.

RAP not only helps develop reproducible science, and improve the quality of the science as the authors debug and refine their methodology, it also provides a precise audit trail that can be used to protect against fraud, as discussed in [27]. RAP can perform checks much faster and more efficiently than conventional *post hoc* investigations.

[3] *Mathematica* is an example of a proprietary system: using it requires a paid-for license, which is a limitation on reproducibility; worse, in the long-run the system owners may go out of business and the code would potentially be unusable at any price. However, these limitations are also limitations that impact free software: versions may become obsolete, and the community may move on and stop maintaining old systems. The impact on reproducibility is much the same as with proprietary systems. The common solution is to code in whichever language is chosen in as portable a way as possible, to chose a system that uses a well-defined notation in an open representation (such as XML or ASCII), so that if the worst happens the old code, or at least the key parts of it, can be translated to run on a new system.

Table 4. A simplified schematic of the publication pipeline. For clarity, the pipeline has been linearized; in general, there will be repetitive cyclic iteration and refinement. The RAP and RAP+ approaches encode the normally manual steps in the pipeline workflow so that they can be run automatically, and hence reproduce the results that underpin the final paper. The encoded RAP+ algorithms can be shared with other scientists, scrutinized, simplified and optimized, and themselves turned into publishable objects — they are scientific instruments, just like thermometers or DNA sequencers. Additional schematics are provided in section 10.f in the Supplemental Material.

Data sources	→	Models and analysis	→	Select results for write up	→	Submit for publication
Experiments		Hand calculations	}			
Standard data		Packages		Copy & paste		
Search engines		SPSS etc		and edit data		Final paper
Literature		Graphics packages		(text, images, graphs, etc)		
Sensors		Specially-written code,		into paper		
		C, Python, R, Mathematica, etc				
⋮		⋮				

Table 5. A minimum standard of RAP, based on the UK Statistics Authority summary [39].

1	**Peer-review is used to ensure the workflow followed is reproducible and to identify improvements**
2	**No or minimal manual interference; for example copy-paste, point-click and drag-drop steps replaced using computer code that can be inspected by others**
3	**Open-source programming languages so that processes do not rely on proprietary software and can be reproduced by others**
4	**Version control software, such as Git, to guarantee an audit trail of changes made to code**
5	**Publication of code, whenever possible, on code hosting platforms such as GitHub to improve transparency**
6	**Well-commented code and embedded documentation to ensure the workflow can be understood and used by others**
7	**Embedding of existing quality assurance practices in code, following guidance set by recognized organizations**

Adopting RAP principles is not necessarily about incorporating all of the above: implementing just some of these principles will generate valuable improvements.

RAP embodies Donald Knuth's comment, also quoted after this paper's abstract,

> 'Science is what we understand well enough to explain to a computer.'
> from the foreword to $A = B$ [1]

The corollary is that if we are doing arbitrary cut and paste that has not been programmed into a computer, then we are not doing good science; we are certainly not explaining what we are doing to the computer. Science is in principle an algorithmic process, and therefore, as Knuth says, if we understand well enough what we are doing in science, we can explain it as code, specifically in a RAP, for a computer to automatically run and rerun.

2.3.1. Potential code

Code is usually thought of as text written in a programming language, such as Python or C, but this ignores special cases of what can be called *potential code*: processes that could be presented in code, but have not been and therefore are invisible. Such potential code cannot be reasoned about as rigorously as they would be had they been expressed explicitly in code. The loss of scrutiny, loss of the ability to reason rigorously, the loss of the ability to review potential code are problems that RAP tries to address. Potential code includes:

1) Critical algorithmic steps may never be codified. Writing a paper may involve creating an image and copying it into a word processed document. Indeed, every time the image is modified, the process must be repeated. In an important sense, the author is executing a computational process, effectively using code that has never been written down. There is a process here, but it is informal and may be run differently each time. However, it could have been coded and reproduced precisely every time it was needed. Moreover, if the process had been coded explicitly, it could be reasoned about, critiqued and improved.
2) Compiling and running programs (for instance to generate results) is also a computational process that may not be recorded in code. The author, more explicitly than creating and using images, runs programs to get results — but the process of running the programs may not be recorded. Typically, if the author notices that the processes are repetitious, they may develop a shell script to codify the repetitive process.
3) Processes in writing a paper, configuring software and generating data may be codified, but the author discovers that their codified processes do not generate quite what they want. Rather than debugging the code, it is tempting to manually edit the final results. The author knows they could have coded things correctly, but this seems too tedious — especially if the edits required seem minor.
4) Another case of potential code arises when a paper has been completed, but referees or the publishers require some minor fixes. These minor fixes are easier to implement in the paper as simple textual corrections, rather than revisiting and updating the explicit code that informed or generated the paper (and updating the repositories and so forth). The RAP approach would require the entire workflow to be revised, not just the final presentation in the paper.

In all cases, a computational process is involved that could have been explicitly coded, but defining the general case as program code seemed harder than an *ad hoc* implicit process. The problem for reproducibility is that the final science depends on this potential code as much as on any explicit code, but it is nowhere recorded, and therefore reconstructing the science will be unreliable.

2.3.2. RAP as research

RAP itself is an object of research. For example, in reproducing the results of a paper published over a decade previously, [43] shows that RAP workflows — which had not been considered or followed at the time — can be semi-automatically reconstructed (along with software dependencies) using suitable tools, thus making the original paper and the experiment it depended on fully reproducible. Furthermore, the newly derived and now explicit pipelines were specified as pure functions, meaning that the workflow was fully-defined using nothing but explicit functions and explicit parameters, arguably a prerequisite for rigorous, deterministic reproducibility.

While this reproduction of an old paper obtained important insights, deriving a RAP workflow after the fact cannot benefit the original scientific process. Instead, insights from this *post hoc* reproduction methodology [43] can be used as an insightful basis to help advance approaches to RAP; that is, doing RAP explicitly (in, for example, the ways that [43] explores) while the science is being developed would provide powerful and constructive insights *during* the scientific process, rather than later in hindsight.

3. THE STATISTICS/CODE ANALOGY

The central role of computational methods in science may be fruitfully compared to using statistics, an established scientific tool.

Poor statistics is much easier to do than good statistics, and there are many examples of science being let down by naïvely planned and poorly implemented statistics. Often scientists do not realize the limitations of their own statistical skills, particularly when developing new experiments, so careful scientists generally work closely with professional statisticians.

In good science, all statistics, methods and results are reported very carefully and in precise detail [44–46], generally following strict journal or disciplinary guidelines. A statistical claim in a paper might be summarized as follows:

> ‘Random intercept linear mixed model suggesting significant time by intervention-arm interaction effect. [...] Bonferroni adjusted estimated mean difference between intervention-arms at 8-weeks 2.52 (95% CI 0.78, 4.27, $p = 0.000\,9$). Between group effect size $d = 0.55$ (95% CI 0.32, 0.77).’ [47]

This standard wording formally summarizes confidence intervals, *p* levels, and so on, to present various statistical results so the paper's claims can be seen to be complete, easy to interpret, and easy to scrutinize. It is a *lingua franca*. It may look technical, but it is written in the standard, widely accepted form for summarizing statistics — it is a clear, rigorous, and readily interpreted way to express uncertainty in results. Moreover, behind any such brief paragraph is a substantial, rigorous, and appropriate statistical analysis.

Scientists write like this and conferences and journals require it because statistical claims need to be properly accountable and documented in a clear way. The journal *Science*, for example, in its many explicit and quite technical statistics requirements requires

> ‘Adjustments made to alpha levels (e.g., Bonferroni correction) or other procedure used to account for multiple testing (e.g., false discovery rate control) should be reported.’ [46]

Spiegelhalter [48] says statistical information needs to be accessible, intelligible, assessable and usable; he also suggests probing questions to help assess statistical quality (see Supplemental Material section 11). Results should not be uncritically accepted just because they are claimed. The skill and effort required to do statistics so it can be communicated clearly and correctly, as above, is not to be taken for granted; in fact, there is widespread concern about the poor quality of statistics in science [49, 50]. While it is assumed that statistics should be peer-reviewed, and that review will often lead to improvement, critical papers like [49, 50] show that reviewers and editors are often failing to pick up on poor statistics.

Scientists accept that statistics is a distinct, professional science, itself subject of research and continual improvement. Among other implications of the depth and progress of the field of statistics, undergraduate statistics options for general scientists are recognized as insufficient training for rigorous work in science — their main value, arguably, is to help scientists to understand the value of collaborating with specialist statisticians. Collaboration with statisticians is particularly important when new types of work are undertaken, where the statistical pitfalls have not already been well-explored.

Except in the most trivial of cases, all numbers and graphs, along with the statistics underlying them, will be generated by computer. Indeed, computers are now very widely used, not just to calculate statistics, but to run the models, to do the data sampling and processing, to operate the sensors or surveys that generate the data, and to process it all. Many papers now explore the contribution of AI and ML to their fields. The data — including the databases and bibliographic sources — and code to analyze it is all stored and manipulated on computers. Computers even help with the word processing and typesetting of the research.

In short, computers, data and computer code are central to modern science, not just to the explicitly computational sciences. Some AI work is uncovering biases and ethical issues that were previously unrecognized, so computational sciences are not just routinely contributing to existing science but extending its reach and improving its quality.

However, using any code raises many critical questions: formats, backup, cyber-vulnerability, version control, integrity checking (e.g. managing human error), auditing, debugging and testing, and more. Is the code correct, and is it dependable enough to justify the claims the scientists would like to make? Software code, like statistics, is subject to unintentional bias [51, 52]. All these issues are non-trivial concerns requiring technical expertise to manage well. As with statistics, good answers to such “technical” issues makes the science that relies on them better; conversely, failing to properly address the questions makes the science suspect.

For example, a common oversight in scientific papers is to present a model, such as a set of differential equations, but omit how that model was reliably transformed into the code that generates the results the paper summarizes. The code may have problems that cannot be identified as there is no specification to reference it to, and possibly even no link to the code at all.

Failure to properly document and explain computer code undermines the scientific value of the models and the results they generate, in the same way as failure to properly articulate statistics undermines the value of any scientific claims. Indeed, as few papers use code that is as well-understood and as well-researched as standard statistical procedures (such as Bonferroni corrections), the scientific problems of poorly planned and reported code are widespread. The terms 'invariant,' 'pre-condition,' 'post-condition' are basic technical terms used in reliable coding, yet none of these concepts appear in in any of the code repositories referred to in this paper's survey. Only one project uses assertions, and then only for checking user interface input data rather than the correct operation of the paper's model (see Supplemental Material data summary). These basic coding concepts are simpler than Bonferroni corrections.

We would not believe a statistical claim that was obtained from some *ad hoc* analysis with a new fangled method devised just for one project — instead, we demand statistics that is recognizable, even traditional, so we are assured we understand what has been done and how reliable results were obtained.

An interesting overlap with statistical and Software Engineering sloppiness concerns the many papers that disclose as part of their methodology that they used a particular software package, for example

> 'Data analyses were performed using SAS 9.2 (SAS Institute, Cary, North Carolina, USA).' [53]

but without giving more details. Besides, the authors have not made their code available so it is moot what system it runs on. The problem is that the common practice of declaring using a named system (such as SAS in this case) does not help scrutiny in the least, as such systems can do almost anything. *How* those analyses might have been performed was not discussed, and one assumes it follows that the analyses could therefore not have been properly reviewed for scientific competence during the publication workflow.

A reviewer, if nobody else, needs to actually examine the code used and its documentation to assess whether the analysis presented in the paper is appropriate and sufficiently reliable. Furthermore, if the analysis in this case actually depended on using SAS version 9.2, and not *any* general purpose statistical system, then it is problematic because it is not reproducible as it depends on idiosyncrasies in SAS version 9.2. Of course, an author can disclose the idiosyncratic dependencies; while this seems to be an onerous obligation, conversely it is arguable that if an author is unaware of dependencies, then their science relying on them is equally unreliable.

It is recognized that to make critical claims, models must be run under varying assumptions [54], yet somehow it is overlook edthat the code that implements those models also needs to be carefully tested under varying assumptions to uncover and fix bugs and biases, as well as to uncover unknown dependencies. Indeed, the code may be poorly written (as this paper shows it often is), so the results derived from the code simply may not be reliable.

In normal scientific reporting (outside of teaching and assessing science) details of methodology are routinely glossed. A chemist does not say they cleaned their glassware. One might argue, then, that scientists need not discuss their code in detail because they know how to program and their code is correct. This argument is mistaken. Code is rarely considered a valuable part of the science to which it contributes (section 4.2), which creates a vicious cycle of ignoring code, leading to ignoring the critical — and non-trivial — role of correct code in science.

3.1. The siren call of over-fitting code

Poor code can generate plausible and possibly misleading results from *any* data or theory, including fraudulent science. A temptation is that developing code to get 'good results' becomes more important than the code's overall faithfulness to the real scientific phenomena, theoretical or empirical.

In conventional modeling terms, successful computer programs are often *over-fitted* to phenomena [55]. That is, instead of using code to rigorously challenge, test, and develop our models, we tinker with code adapting it to generate results closer and closer to our prejudices. The code then apparently confirms our science, since we fitted it to our preconceptions but not to the science.

In general, an *over-fitted model* fits a set of data closely, but contains more parameters than can be justified by the science. An over-fitted model fails to reliably predict results beyond the scope of the data it has been fitted to. Over-fitting is a well-known problem, but the point is that when code is used, over-fitting is done unconsciously by programmers adjusting the code — its parameters, its structure, its embedded data, the calculations it performs — that specifies the model. 'A model over-fits if it is more complex than another model that fits equally well' [55], is a criterion that describes almost every program! Programmers without the discipline and experience to manage the unlimited adaptability of code debug, alter, and extend their code to make it do what they think it should do. This becomes a vicious circle as the idea of what the science is becomes driven by the code. Rather than debugging code by improving its fit to the actual science, it gets debugged and extended to fit the expectations.

The problems of over-fitting data may be visualized using a Real→Real function of one variable (Fig. 1). The code that generates an over-fitted curve seems to work very well: in the example shown, the over-fitted curve fits the sampled data exactly; indeed, the code used here will fit any new data exactly as well. But the code has a negligible ability to predict new data or to describe theories of the data, which is the point of modeling. The fact that over-fitted code seems to work well is deceptive.

Looking specifically at the data plotted in Figure 1 if, for the sake of argument, we assume the error in the data is normally distributed then the values the over-fitted code generates outside the range of the sample are improbable. For example, the basic linear model predicts $\hat{y}(0) = 0.5$, versus the extreme value predicted by the over-fitting code, $\hat{y}(0) = -41.8$, even lies far outside the plot region shown in the figure.[4] Similar problems happen with interpolation rather than extrapolation, for instance around $x = 4$.

While over-fitting data is a well-known problem, the point for this paper is that *code* itself can easily be over-fitted. Code can of course be over-fitted in more complex ways than can be illustrated with elementary polynomials, as here. Code over-fitting is much harder to recognize because there may be no simple graph plot, like Figure 1, to highlight the problems. Furthermore, almost all code is far more intricate than the two trivial polynomials used to illustrated in Figure 1.

[4] The bounds of the confidence interval illustrated in Figure 1 depend on assumptions about the distribution of the data; in this example, we are *assuming*, perhaps because we know something more about the experiment, or thanks to Occam's Razor that a linear function is more likely than a high order polynomial.

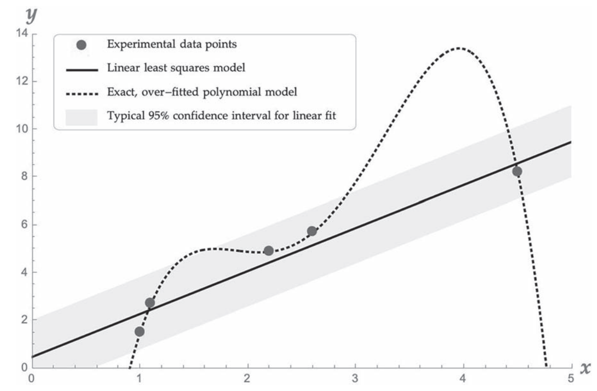

Figure 1. Much computational science is concerned with finding plausible multi-dimensional models that fit models to data with the aim of extrapolating or predicting new results from them. Shown here is notional sample of experimental 2D data (the dots), a linear least-squares regression, and an exact polynomial model. The over-fitted polynomial model fits the sample *exactly*, but since the experimental data are presumably subject to random error (indicated by the confidence interval, itself estimated), the linear model would generally be considered a better description of the experimental data.

Unfortunately, code in published science is often over-fitted, and over-fitted in a way that is very hard to scrutinize. For example, in epidemiology (which is considered in Section 5.1) it is routine to use very complicated, large dynamical models parameterized with numerous social, cultural, health, demographic and geographical data. The parameterization is mixed between data files, data written explicitly into the code, and with conditionals and other structuring in the code to cover special cases. Indeed, many of the programs in the survey used comments to inactivate code, presumably indicating an unfinished tinkering approach to code development.[5]

Furthermore, scientific support code is rarely documented well enough to know what it should have been doing, which should be answered by a specification. With no clear specification and documentation, the code can be arbitrarily hacked to get any convenient results, since no particular specification for it has been defined that it should adhere to. Thus we risk doing and promoting substandard science because we — the scientists and the publication process — are not managing the unlimited adaptability and complexity of code that science has come to rely on. This is over-fitting of the worst kind — in conventional over-fitting one can at least hope to see that the fitting is over-parameterized for the data, but in code over-fitting the code and specification are not visible, therefore not adequately scrutinized, and — worse — the 'data' the code over-fits includes the entire conceptual contribution of the paper.

Reference [56] shows that even trivial code (in the case cited, implementing simple difference equations) with very few parameters can have very complex results, and reference [57] is a historically significant paper pointing out how the problems of over-fitting has improved science.

[5] Of the 10 papers in the pilot survey that reported use of code repositories (covering 182 thousand lines of code—so this is not a trivial amount of programming effort), one provided an empty repository with no code at all (effectively commenting out all their code!), and seven repositories explicitly commented out chunks of workable code. The two remaining non-trivial repositories with no commented-out code consisted of straightforward, short code files with few comments of any sort.

4. THE CONVENTIONAL ROLE OF CODE

Models map theory and parameters to describe phenomena, typically to make predictions, or to test and refine the theory supporting the models. With the possible exception of theoretical research, all but the simplest models require computers to use; indeed even theoretical mathematics is now routinely performed by computer.

Whereas the mathematical form of a model may be concise and readily explained, even a basic computational representation of a model can easily run to thousands of lines of code, and its parameters — its data — may also be extensive. The chance that a thousand lines of code is error free is negligible, and therefore good practice demands that checks and constraints should be applied to improve its reliability. How to do this is the concern of Software Engineering.

While scientific research may rely on relatively easily-scrutinized mathematical models, or models that seem in principle easy to mathematize, the models that are run on computers to obtain the results published are sometimes not disclosed, and even when they are they are long, complex, inscrutable and (as our survey shows) lack adequate documentation. Therefore the models are very likely to be unreliable *in principle*.

If code is not well-documented, this is not only a problem for reviewers and scientists reading the research to understand the intention of the code, but it also causes problems for the original researchers themselves: how can they understand their historical thinking well enough (say, just a few weeks or months later) to maintain it correctly if it has not been clearly documented? As a scientist pursues a research career building on their previous work, how can they be certain their work is reliable, and not merely converging to their prejudices? Without proper documentation, including a reasoned case to assure that the approach taken is appropriate [58], how do researchers, let alone reviewers, know exactly what they are doing?

Without substantial documentation it is impossible to scrutinize code properly. Consider just the single line '`y = k*exp(x)`' where there can be *no* concept of its correctness *unless* there is also an explicitly stated relation between the code and the mathematical specifications. What does it mean? What does `k` mean — is it a basic constant or the result of some previous complex calculation? Does the code mean what was intended? What are the assumptions on `k`, `x` and `y`, and do they hold invariantly? Moreover, as code generally consists of thousands of such lines, with numerous inter-dependencies, plus calling on many complex libraries of support code, it is inevitable that the *collective* meaning will be unknown. A good programer would (in the example here) at least check that `k` and `x` are in range and that `k*exp` was behaving as expected (e.g. in case of under- or overflow).

Without explicit links to the relevant models (typically mathematics), it is impossible to reason whether any code is correct, and in turn it is impossible to scientifically scrutinize results obtained from using the code. Not providing code and documentation, providing partial code, or providing code without the associated reasoning is analogous to claiming 'statistical results are significant' without any discussion of the relevant methods and statistical details that justify making such a claim. If such an unjustified pseudo-statistical claim was made in a scientific paper, a reviewer would be justified in asking whether a competent experiment had even been performed. It would be generous to ask the author to provide the missing details so the paper could be better reviewed on resubmission.

Some authors assert that the purpose of code is to provide insight into models, rather than precise (generally numerical) analyses summarizing data or properties of the data [59]. In reality, if code is inadequate, any so-called 'insights' will be potentially flawed, and flawed in unknown ways. Indeed, none of the papers sampled (see Supplemental Material section 12) claimed their papers were using code for insight; all papers claimed, explicitly or implicitly, that their code outputs were integral to their peer-reviewed results.

Clearly, like statistics, coding can be done poorly and reported poorly, or it can be done well and reported well — and any mix between the extremes. The question is whether it matters, *when* it matters, and, if so, when it does, *what* can be done to *appropriately* help improve the quality of code (and discussions about the code) in scientific work?

4.1. The deceptive simplicity of code

It is a misconception that programming is easy and even children can do it [60]. More correctly, toy programming is easy, but mature programming is very difficult.

An analogy helps justify this key point. Building houses is very easy — indeed, many of us have built toy Lego houses. Obviously, though, a Lego house is not a *real* house. It is not large enough or strong enough for safe human habitation! This point is obvious because we can see Lego houses, and everyone is familiar with the limitations of building-block play. Its real-world engineering limitations are too obvious to need stating.

In contrast to Lego, computer programs are generally invisible, and therefore the engineering problems within them are also made invisible. The 'programming is easy' cliché is deceptive — programming appears easy *because* professional standards of building software are ignored, because people cannot see the reasons why they are needed, and because — like Lego — toy programs can look inspiring but be unreliable, difficult to use, even dangerous.

Saying programming is easy is like appreciating a child's Lego building because we are not worried about subsidence, load bearing, electric shock, fire risks, water ingress, or even planning regulations. These are professional engineering issues that Lego builders ignore. Certainly, even real building is much easier and faster when the technical details are ignored, as anyone who has experienced a cowboy builder can attest.

Unlike building houses (the Code of Hammurabi dates from around 1755 bc[6]), programming is a new discipline, and the problems of poor programming are not widely appreciated or embedded in our culture. Professional standards, even when they exist, are not enforced.

Problems for the reliability of science arise when doodling and tweaking software drifts into claiming scientific results that do not have reliable engineering processes or structures underpinning them (let alone the properly developed and documented accessible code) to justify them.

In many countries, there are laws that require all but the very simplest building structures to be formally approved from plans and inspected as they are built, but who writes plans for software, who inspects scientific models while they are being coded? Yet the consequences of building a shoddy garage have negligible impact compared to the consequences of writing poor code that informs national public health policies or climate change interventions.

[6] The Code of Hammurabi says, '(§233) If a builder constructs a house [and] does not make it according to specifications, and a wall then buckles, that builder shall make that wall sound using his own silver.' [61]

4.2. The low status of coding

Since programming appears to be so easy, developing code has a correspondingly low status in scientific practice (and more widely). Developers of code are rarely acknowledged in scientific papers. The implicit reasoning is: if programming is easy, then its intellectual contribution to science is negligible, so it is not even worth citing it or acknowledging the contributors to it. Because it is apparently easy, there is no need to work hard to make it correct. Because of the ease of over-fitting (Section 3.1), code 'works well' with little skill or effort. While such mistaken views prevail, the vicious cycle is that the low status means software development is casualized, which reinforces the low status.

Almost all scientific papers *routinely* describe their experimental method, their data handling, and provide an overview of their analytic (usually statistical) methods. If they are theoretical papers, they will describe their mathematical models and data that is used to run or test their models. However, outside of pure computer science, scientific papers are almost entirely silent on the code they rely on and how it was developed — in particular, how the code might have been protected from bugs, analogously to how appropriate experimental methods were used avoid or control for experimental error.

Since published papers rarely mention their code, new papers contributing to the literature do not write about their code either, so the low status of code persists.

In reaction to this vicious cycle, there is a growing movement to use and cite code correctly [25, 62], because code *is* important, particularly for the reproduction, testing and extension of any scientific work. (Code also needs to be correct, not just cited correctly.) Few journals editorial policies recognize that data and code are in practice indistinguishable (see Supplemental Material). Given that data and code are equally important in science, effectively equivalent and interchangeable, it follows that publishing policies on data handling should also apply at least as strictly to code.

4.3. The critical role of code is often ignored

Because statistics, like code, is so readily susceptible to uncontrolled bias and error, there are many protocols and journal policies that enforce best practice, for example journals often require adherence to PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) [63] for any paper performing a systematic review of the literature. PRISMA is a leading and influential research protocol, and it serves to make the point that the critical role of code is often strategically ignored.

PRISMA is not concerned with the reproducibility of the literature reviewed, nor the reproducibility of systematic reviews themselves. PRISMA not only ignores the role of code, it ignores the Software Engineering principles that assure code that research relies on is reliable and reliably reported.

PRISMA covers the review workflow. For example, it states that authors should report the number of papers they included in their review. Perhaps $N = 2000$. This number will then written be into the review, perhaps in several places. As the authors read and revise their paper and respond to peer-reviewers, it is likely that the number of papers in the survey will change; other numbers and details will certainly change.

The authors now have a maintenance problem that PRISMA does not address: where are the numbers that have changed, and what should they be changed to? Doing a search-and-replace, whether automated or by hand, is fraught with difficulties. What happens if 2000 is used for some other purposes as well? What happens if some of the 2000 values are written as 2000 or as

2000.0, or if a number containing the digits 2000, like 12 000, is changed? What happens if some 2000 are year dates and are changed incorrectly?

Then there are the Human Factors: slips and errors will happen in this workflow anyway [64]. Typos, slips during cut-and-paste, and other errors are common. Similar iterative revision cycles happen with any paper, not just with systematic reviews.

PRISMA, like many such standards, ignores the methodological problems of using code such as the issues raised above.

The irony, then, is that PRISMA says nothing about how to ensure the results of a survey are correctly and reliably presented, despite this being one of PRISMA's explicit motivations. PRISMA explicitly warns about methodological issues, but ignores that poorly managed code raises methodological issues that will undermine the validity of research it supports. PRISMA reinforces the culture that code is trivial and incidental to research.

4.4. Bugs, code and programming

Critiques of data and model assumptions are increasingly common [65, 66] but program code is rarely mentioned. Program code has as great an effect on results as the data; in fact, without code, the data would be uninterpreted and almost useless. Code, however, is harder to scrutinize, which means that errors in code have subtle, often unnoticed effects on results.

Program code contains data. Almost all code contains 'magic numbers' — that is, data masquerading as code (see Supplemental Material, section 10). This common practice ensures that published data is rarely all of the relevant data because it omits the magic numbers embedded in the code. Such issues emphasize the need for repositories to require the inclusion of code so all data, including that embedded in the code, is actually available.

Conversely, much data contain code. Excel spreadsheets are often used to manage code, and almost all Excel spreadsheets contain macros and formulæ — code embedded in the data. JSON, the JavaScript Object Notation, is a structured data language, but as it is part of the JavaScript programming language there is no practical code/data distinction. Indeed, the present paper's data is in a file `data.js` that also contains the JavaScript code to analyze it and generate results (such as Table 2) for presentation in this paper.

Although convenience and convention treat data and code differently, ultimately, data and code are formally equivalent (see Supplemental Material, Section 10) in the important sense that there is no pre-determined boundary between them or formal criteria to distinguish them: different scientific projects can draw their boundaries differently and as they see best fits their work. Indeed, all of the papers reviewed here where code was available include critical data in their code. It therefore makes no sense to have separate rules for data and code — except in the most trivial cases *both* are equally essential for verification, building on and reproduction of work.

Bugs can be understood as discrepancies between what code ought to do and what it actually does. Many bugs cause erroneous results, but bugs may be 'fail safe' by causing a program to crash so no incorrect result can be delivered. Contracts and assertions are essential defensive programming technique that block compilation or block execution with incorrect results; they turn bugs into safe termination, or, better, failure to compile. None of the science surveyed in this paper includes any such basic techniques.

Random numbers are widely used in computational science (and in many of the papers surveyed), for simulation or for randomizing experiments. Misuse of random numbers (e.g. using standard libraries without testing them) is a very common cause of naiïve bugs [67].

If code is not documented it cannot be clear what it is intended to do, so it is not possible to detect and eliminate bugs. Indeed, even with good documentation, *intentional bugs* will remain, that is, code that correctly implements the wrong things [60, 68]. Intentional bugs occur in code that correctly does what was intended, but what was intended was itself faulty (students and inexperienced programmers regularly make intentional bugs). Intentional bugs frequently arise in numerical modeling, where using an inappropriate method can introduce errors that are not bugs in the sense of failing to correctly implement what was wanted, but are bugs in the sense that the wrong numerical method was chosen and inaccurate results are obtained; that is, what was intended was wrong.

4.5. Long-term problems of unreliable code

Scientists explore and extend the boundaries of rigorous knowledge. Put briefly, the purpose of scientific experiments is to vary details to either test and specify the boundaries of theories, or to discover new phenomena that then lead to theory revision.

If poor code, or poorly documented code, is made available with scientific papers, the code is a natural place to start replicating and varying experimental conditions, including both data or code. However, if the starting point is not accurately known, whether due to bugs, obscure code or because of poor documentation, then experimental variations will have an unknown effect. Theory will then be driven by artifacts of the code, not genuine phenomena.

In Section 5.1, below, an example is documented of a research code development process of at least 15 years' duration where the code was admitted to be completely undocumented, leaving details in just one author's head. None of the various related papers describe any controls over the drift of the science, or how independent researchers building on it might have been able to build with confidence rather than merely reproducing the same errors.

Since the code in question was substantial and non-trivial, it is very unlikely that any constructive reproduction occurred outside the original laboratory and mindset; indeed, Section 5.2 describes how 'reproduction' became trivialized because of community pressure to confirm the insights of this particular research.

Trying to constructively refute aspects of this research in the Popperian sense [69] would have been impossible. For example, had the relevant papers published critical code invariants then scientists building on the research could have explored whether those invariants remained valid and, if so, under what assumptions. In fact, invariants are the theories of code, and deserve as high a prominence in published science as the domain theories the code itself is supporting investigating.

5. STATE OF THE ART

5.1. Case study: pandemic modeling

For an excellent review of the extreme pressures under which scientists were working during the COVID-19 pandemic (but nothing about the role of computers!), see [70] — which, while referring to some failed mRNA vaccines, makes an important point on recovering from "failed" science:

> 'Such is the beauty of science: even failed attempts are a step towards more information and progress forward.' [70]

But progress, if any, would be chaotic if it was not possible to scrutinize exactly what science has failed. It undermines progress if science and the code it relies on are not both accessible and adequately defined. It is as misleading as getting the statistics or mathematics wrong.

By focusing on influential science and coding undertaken to inform public health policies during a pandemic emergency, this section now focuses on an area where reliable and high quality science using code open to interdisciplinary scrutiny was very obviously required. Note that pandemic model correctness is a secondary concern here; correctness is not addressed as without informed scrutiny correctness cannot even be assessed.

A review of epidemic modeling [71] says, “we use the words ‘computational modeling’ loosely,” and then, curiously, the review discusses exclusively mathematical modeling, implying that for the authors, and for the peer-reviewers, there is no role for code or computation as such. It appears that the new insights, advances, rigor, and problems that computers bring to research were not considered relevant.

A systematic review [66] of published COVID-19 models for individual diagnosis and prognosis in clinical care, including apps and online tools, noted the common failure to follow standard TRIPOD guidelines [72]. The review [66] itself ignored the mapping from models to their implementation, yet if code is unreliable, the model *cannot* be reliably used, and cannot be reliably interpreted regardless of whether TRIPOD guidelines are followed. Indeed, TRIPOD guidelines ignore code completely.

It should be noted that flowcharts, which the review [66] did consider, are graphical programs intended for human use. Flowcharts, too, should be designed as carefully as code, for exactly the same reasons.

A high-profile 2020 COVID-19 model [9, 73], which influenced UK COVID-19 public health strategies, uses a modified 2005 computer program [10, 11] originally developed for modeling H5N1 in Thailand, when it did not model air travel or other factors required for later western COVID-19 modeling. The 2020 model forms part of a series of papers [9–11] none of which provide details of their code.

A co-author disclosed [74] that the code was thousands of lines long and was undocumented code. As Ferguson, the original code author, noted in an interview,

‘For me the code is not a mess, but it’s all in my head, completely undocumented. Nobody would be able to use it . . . ’ [75]

The admission above is tantamount to saying that the published scientific findings are and need not be reproducible.[7]

The comment was made by a respected, influential world-leading scientist, with many peer-reviewed publications involving computational modeling, with a respectable *h*-index[8] of 93, and at the time ‘one of the top scientists advising the government on its response to the coronavirus crisis’ working in the UK’s Scientific Advisory Group for Emergencies (SAGE) group [77].

Ferguson’s code must be representative of best practice when the stakes were high and reliability was known to be essential; and if not representative of best practice, at least representative of accepted practice both in Ferguson’s team, the field of epidemiology more widely, as well as with members of the high-powered interdisciplinary SAGE group. It is therefore instructive to explore the larger story around this science that uses code.

Lack of reproducibility is problematic, especially as the model code would have required many non-trivial modifications to update it for COVID-19 with its different assumptions; moreover, the code would have had to have been updated very rapidly in response to the urgent COVID-19 crisis.

If Ferguson’s C code had been made available for review, the reviewers would not have known how to evaluate it without the relevant documentation. It is, in fact, hard to imagine how a large undocumented program could have been repeatedly modified and repurposed over fifteen years without becoming incoherent.

If code is undocumented, there would be an understandable temptation to modify it arbitrarily to get desired results (i.e. over-fitting, see Section 3.1); worse, without documentation and proper commenting, it is methodologically impossible to distinguish legitimate attempts at debugging from merely fudging the results. In contrast, if code is properly documented, the documentation defines the original intentions (including, where appropriate, formally using mathematics to do so), and therefore any modifications will need to be justified and explained — or the theory revised.

The programming language C which was used [74] is, like many popular programming languages, not a dependable language; to develop reliable code in C requires professional tools and skills. Some of the code was written in a naïve style (e.g., writing `*(a+i)` instead of `a[i]`, and with obscure numerical goto statements like `if(l == 0) goto S150`), and with C code that was translated simplistically from FORTRAN and Pascal code, from references dating back to the 1970s and 1980s [e.g., 78, 79].

Moreover, C code is not portable, which limits making it available for other scientists to use reliably: C notoriously gets different results with different compilers, libraries, or hardware. In fact, in any area where reliable programming is required in a C-like language, a special dialect such as MISRA C is preferred: MISRA C manages the serious design flaws of C that otherwise make it too unreliable [80]. Alternatively, a high integrity programming language, unrelated to C, such as SPARK Ada [81], or modern languages (many related to the ‘ML family’) like OCaml, F∗, Haskell [82] could be used. These languages have steeper learning curves; however their key benefit is that correct programs are far more likely and are *much* faster to write. (The Supplemental Material discusses these issues further.)

Ferguson, author of the code, says of the criticisms of his code,

‘However, none of the criticisms of the code affects the mathematics or science of the simulation.’ [83]

This claim is implausible.

The original work on theoretical epidemiology may be fine if it does not use any of his code, but if the science is not supported by code that correctly implements the models, then the program’s output cannot be relied on without independent evidence. Over the fifteen plus years the code was in development the science it informs will have developed too, as will the relevant data; it is not clear how they will have remained in alignment.

Typically, models will be developed iteratively as their results are improved to better fit a scientist’s goals — but this, especially when it is done by tinkering, as here — risks making the code arbitrarily fit the goals (that is, over-fitting; see Section 3.1), rather than to objectively elucidate the science.

[7] A constructive discussion of Software Engineering approaches to reproducibility can be found in [76].

[8] *h*-index: the largest value of *h* such that at least *h* papers by the author have each been cited at least *h* times. The figure cited for Ferguson was obtained from Google Scholar on 20 January 2022. (Typical *h* values vary by discipline.)

In fact, the Ferguson code, `covid-sim`, is a very large program at 25 kLOC (thousands of lines of code),[9] so it is implausible that the 'mathematics or science' has been correctly implemented in it without error, particularly as there is no discussion of methodologies to code reliably. Ferguson's *reported* science is consequently unlikely to be reliable.

5.2. Concerns with reproducibility

Getting science right, which now, in turn, depends on correct code, is a normal requirement of *reproducibility*.

The code in [9, 73] has been 'reproduced,' as reported in *Nature* [83, 84], but this so-called reproduction merely confirms that the code can be run again and produce comparable results. As Eglen says,

> "Each run generated a tab-delimited file in the output folder. Two R scripts provided by Prof Ferguson were used to summarize these runs into two summary files [. . .] These files were compared against the values generated by Prof Ferguson [. . .] The results were found to be identical. Inserting my results into his Excel spreadsheet generated the same pivot tables. The codecheck found that: 'Small variations (mostly under 5%) in the numbers were observed [. . .]'" [84]

This test would pass provided the runs gave the same answers regardless of whether the answers are correct — it is not a usefully stronger test than just checking that the code compiles. The comparison relied on running (apparently) unchecked R code to summarize the data, which is potentially misleading unless the published results [9] exclusively relied on the same summary code. In general, reproducing code results, even done formally, does not scrutinize the science, as [85] makes clear.

Running code just to obtain results claimed in a paper is a weak test, and anyway one that should be checked routinely during paper preparation and submission. However, in this case the reproduction involved a community effort that also refactored and improved the code, which added value to the code and usefully improved its generality [84]. The reproduction effort was also certified [85], which is the sort of evidence of quality assurance processes that arguably should be required before publication, particularly for critical code such as public health modeling. The publicity of this story and the certificate will certainly raise the profile of the scientific value of independent review of code.

Unfortunately, the terms reproducibility, replicability, and repeatability, have similar meanings in English but have been used in different specific technical ways by different authors. In [83, 84] the reproduction amounted to just re-running the original code. It is certainly essential to establish that a paper's code can be run, as non-working code cannot support any claims in a paper; if the original code runs this confirms a basic level of access for the wider scientific community, and this can be formally certified [85] so it is appreciated by the community. But a more realistic criterion than basic reproduction in this sense is whether an *independently* developed model developed from the same paper(s) specifications produces equivalent results (called *N*-version programming, a standard Software Engineering practice [86]) like public health surely requires as, indeed, Ferguson's own influenza paper [87] argues.

In general, much stronger scrutiny of code than 'reproduction' is required to answer essential questions (numbered below for reference) including:

1) Is the code valid: does it do what the paper claims?[10]
2) Do other scientists, including reviewers and the authors, understand the code?
3) Does the code implement the methods described in the paper?
4) Has the code been over-fitted or tweaked to support specific claims in the paper?
5) Is there a definitive version of code?
6) Is the code controlled and signed?
7) What limitations does the code have?
8) Was the code developed to any standard, and does it comply to that standard?
9) How does the code protect against data, coding, and human error?
10) Was the code tested adequately?
11) Does the code depend on arbitrary parameters, data, or code to over-fit to obtain the published results?
12) Is the code documented adequately, so we know what it is trying to do, and how?
13) . . . and so forth.

All such questions also apply to specifications, documentation, assurance cases, test procedures, and other essential documents, not just to code. In turn, the levels of scrutiny demanded should be guided by explicit claims in the paper [68] — for example, a pilot study requires weaker assurance than code that is developed concerning nuclear power, driverless vehicles, public health, etc.

The questions in the list above are certainly hard to answer for all but the briefest code, but corresponding levels of quality assurance are demanded for other methodologies [63, 69, 72, 88–90], such as data preparation and statistics to support claims in peer-reviewed science.

Because of the recognized importance of the Ferguson paper, a project started to document its code [91].[11] Documenting code in hindsight, even if done rigorously, may describe what it does, *including* its bugs, but it is unlikely to explain what it was originally intended to have done. As the code is documented, bugs will be found, which will then be fixed (refactoring), and so the belatedly documented code will not be the code that was used in the published models; it will be different.

It is well-known that documenting code helps improve it, so it is surprising to find an undocumented model being used in the first place, since so many years' opportunity to improve the code have been lost. The revised code has now been published, and it too has been heavily criticized [e.g. 92], supporting the concerns expressed in the present paper.

Some papers [e.g. 93] publish models in pseudo-code, a simplified form of programming. Pseudo-code looks deceptively like real code that might be copied to try to reproduce it, but pseudo-code introduces invisible and unknown simplifications. Pseudo-code, properly used, can give a helpful impression of the overall approach of an algorithm, certainly, but pseudo-code alone

[9] Ferguson's `covid-sim` system is composed of 229 files, and uses 734 Mb of data. It is now rewritten from C into C++ with Python, R, sh, YAML/JSON, etc. For more details, see Supplemental Material.

[10] Of course, the underlying science may be wrong to, so it is useful to distinguish *internal validity* and *external validity*. Internal validity occurs if the code does what the paper claims; external validity occurs if the code represents correct science which the paper may have interpreted incorrectly.

[11] It is surprising to find an undocumented model being used in the first place, since so many years' opportunity to improve the code have been lost. The revised code has now been published, and it too has been heavily criticized [e.g. 91], supporting the concerns expressed in the present paper. n-source, available at github.com/mrc-ide/covid-sim version (19 July 2021).

is not a surrogate for code: using it instead of making actual code available is worse than not publishing code at all (see [94]). Pseudo-code is too vague to help anyone scrutinize a model; moreover, pseudo-code may mask over-fitting in code used that is not explicit in the pseudo-code.

An extensive criticism of pseudo-code, and discussion of tools for reliable publication of code can be found elsewhere [34].

The Supplemental Material provides further discussion of reproducibility.

5.3. Beyond pandemic modeling

Epidemiology has a high profile because of the COVID-19 pandemic, but the problems of unreliable code are not limited to COVID-19 modeling papers, which, understandably, were perhaps rushed into publication. But other examples that were not rushed include a 2009 paper reporting a model of H5N1 pandemic mitigation strategies [95], which provides no details of its code. Its supplementary material, which might have provided code, no longer exists.

There are many other areas of computational science that are equally if not more critical, and many will have longer-lasting impact. Climate change modeling is one such example that will have an impact long beyond the COVID-19 pandemic.

A short 2022 summary of typical problems of Software Engineering impacting science appears in *Nature* [96], describing diverse and sometimes persistent problems encountered during research in cognitive neuroscience, psychology, chemistry, nuclear magnetic resonance, mechanical and aerospace engineering, genomics, oceanography, and in migration. The paper [96] makes some misleading comments about the simplicity of Software Engineering, e.g., 'If code cannot be bug-free, it can at least be developed so that any bugs are relatively easy to find.'

Guest and Martin promote the use of computational modeling [97], arguing that through writing code, one debugs scientific thinking. Psychology, their focus, has an interesting relationship with software, as computational models are often used to model cognition and to compare results with human (or animal) experiments [97]. In this field, the computation does not just generate results, but is used to explicitly explore the assumptions and structures of the scientific frameworks from which the models are derived. Computational models can be used to perform experiments that would be unethical on live participants, for instance involving lesioning (damaging) artificial neural networks. It should be noted that such use of cognitive models is controversial — on the one hand, the software allows experiments to be (apparently) precisely specified and reproduced, but on the other hand in their quest for psychological realism the models themselves have become very complex and it is no longer clear what the science is precisely!

For instance, ACT-R, one widely-used theory for simulating and understanding human cognition, has been under development since 1973, and is now a 120 kLOC Common LISP and Python system [98]. Furthermore, any paper using ACT-R would require additional code on top of the basic ACT-R framework.

The psychology paper [97] presents an example computational model from scratch to illustrate a framework of computational science. In fact their example model has no psychological content: a simple numerical test is performed, but the psychology of why the result is counterintuitive — the psychological content — is not modeled. Be that as it may, they develop a mathematical specification and discuss a short Python program they claim implements it.

The Python code is presented without derivation; Software Engineering is ignored. The program listed in the paper certainly runs without obvious problems (ignoring some typographical errors due to the journal's publishers), but ironically the Python does *not* implement the mathematical specification explicitly provided for it, thus undermining the argument of the paper.

One might argue the bug is trivial (the program prints `False` when it should print `b`), but to dismiss such a bug would be comparable to dismissing a statistical error that says p= `False` which would be nonsense — if a program printed that, one would be justified in suspecting the quality of the entire program and its analyses. Inadvertently, it would seem, then, that the paper shows that just writing code does not help debug scientific thinking: instead, code must first be derived in a rigorous way and actually be correct. Otherwise, code based on inadequate Software Engineering will introduce errors into scientific thinking.

Code generally for any field of scientific modeling needs to be carefully documented and explained because all code has tacit assumptions, bugs and cybersecurity vulnerabilities [51, 52, 96] that, if not articulated *and properly managed*, can affect results in unknown ways that may undermine any claims. People reading the code will not know how to obtain results because they do not know exactly what was intended in the first place. The problem is analogous to the problem of failing to elaborate statistical claims properly: failure to do so suggests that the claims may have unknown limitations or flaws.

Even good quality code has, on average, a defect every 100 lines — and such a low rate is only achieved by experienced industrial software developers [99]. World-class software can attain maybe 1 bug per 1000 lines of code. Code developed for experimental research purposes will have higher rates of bugs than professional industrial software, because the code is less well-defined and evolves as the researchers gain new 'insights' into their ideas, unable to distinguish genuine insights from artifacts of bugs. In addition, and perhaps more widely recognized, code — especially but not exclusively mathematical code — is subject to numerical errors [100]. It is therefore inevitable that typical modeling code has many bugs (reference [86] is a slightly-dated but very insightful discussion). Such bugs undermine confidence in model results.

Only if there is access to the *actual* code and data (in the specific version that was used for preparing the paper) does anyone know what the researchers have done and whether that corresponds closely to what they are reporting.

Some COVID-19 papers [e.g., 101] make unfinished, incomplete code available. While some [e.g. 101, 102] make what they call 'documented' code available, they provide no more than superficial comments. This is *not* documentation as properly understood. Such comments do not explain code, explain contracts, nor explain algorithms. Contracts, for instance, originated in work in the 1960s [103], and are now well-established practice in reliable programming.

Even if a computer can run it, badly-written code (as found in *all* the research reviewed in the present paper, and indeed in computer science research, e.g. [20]) is inscrutable. Only if there is access to *adequate* documentation can anyone know what the researchers *intended* to do. Without all three (code, data, adequate documentation), there are dangers that a paper simplifies or exaggerates the results reported, and that omissions, bugs and errors in the code or data, generally unnoticed by the paper's authors and reviewers, will have affected the results they report [34].

5.4. The lop-sided emphasis on data

Data have been at the center of science, certainly since the earliest days of astronomy collecting planetary and other information. Today it is widely recognized that lack of accessible and usable data that has already been collected limits the progress of science.

Low quality data and poor access to data causes reproducibility problems, an increasingly recognized problem — in 2015 it was estimated that $28 billion a year is spent on preclinical research that is not reproducible [104].

Curating data are taken seriously as a part of normal science and peer-reviewed publication. Journal policies widely require appropriate discussion of data, much as they require appropriate discussion of statistics. Journals often require archiving data in standard formats so it can be accessed for reproduction in further scientific work.

There are many current activities to proceduralize and standardize the more effective curation and use of data, such as the FAIR principles (Findable, Accessible, Interoperable and Reusable) for scientific data management and stewardship [105, 106], and in the development of journal and national funder policies. For example, the 2022 update to the US National Institutes of Health data policies [89] is described as a 'seismic mandate' by *Nature* [90] in its attempt to improve reproducibility and open science yet they ignored code.

These cost estimates and initiatives under-play the role of code as a critical component despite its becoming the new laboratory for almost all science. The role of code specifically in modeling is discussed throughout this paper; without bespoke code, proposed models (unless intended to be abstract) cannot make a quantifiable contribution to the literature. Code has additional problems of versions and compatibility beyond those of data, for example suitable compilers to run old code may no longer be available, and programming systems may produce different results when used on different computers.

In general, without proper management of code — for example to record, detect and report version control differences — sharing code may even be counter-productive.[12]

Using structured repositories that provide suggestions for and which encourage good practice (such as Dryad[13] and GitHub), and requiring their use, would be a lever to improve the quality and value of code and documentation in published papers. The evidence (see Supplemental Material) suggests that, generally, some but rarely all develop code that is uploaded to a repository just before submitting the paper in order to 'go through the motions.' In the surveyed papers there is no evidence (before, during, or after the date of the survey sample) that any published code was prepared or maintained *using* repositories. This is consistent with the finished code being uploaded to a repository just for the purposes of satisfying publishing requirements, but not using one earlier probably because they did not understand the benefits of doing so — not using a repository *during* the research process means the author of the paper misses out on the many helpful features of repositories, such as version control, review, actions, and other approaches for automating development, sharing workload, and so on. Using repositories during research means that other people can more easily help review the approach, in much the same way that papers are routinely circulated for peer review before submitting to a formal journal.

[12] The data and code shared with the present paper includes cryptographic checksums; if somebody reproducing the work described here does not obtain the same checksums at least when they start their work, then there are problems that need investigation before relying on the reproducibility of the data.

[13] Dryad datadryad.org curates raw, unprocessed data. At the time of writing, Dryad excludes code; however, it uses a separate organization, Zenodo zenodo.org, to host code and other relevant information. This arbitrary separation is unfortunate as it increases management problems, increases reproducibility problems, limits using RAP, and most seriously limits how scientists can structure their data and code to best suit their research (see Section 4.4).

There is a lop-sided emphasis on data in science. In fact, data are useless without code, and code must be used to manipulate and analyze it. Often code is used to extrapolate data, so the code itself effectively generates more data, or the code eliminates outliers so it effectively deletes data. Data are routinely formatted in simple or standard ways, but code, in contrast, are architecture- and version-specific, so — unless properly managed — code goes obsolete faster than data. In short: the integrity of code and its availability to scrutiny is in fact both harder and more important than the usual requirements put on data.

6. RETHINKING SCIENCE THAT USES CODE

Computer programs are the laboratories of modern scientists, and should be used with a comparable level of care that virologists use in their laboratories — lab books and all [88] — and for exactly the same reasons: bugs, whether computer bugs or biological bugs, accidentally cultured in one laboratory can infect research, ideas, and policy worldwide.

Inadequate scientific code can be problematic. Incorrect results might be used for supporting science, modeling pandemics or informing public health policy, informing medical research, or adopted for use in other critical software, such as medical diagnostics, credit checking, or any other impactful use. Professional critical software development, as used in critical industries such as aviation and the nuclear power, is (put briefly) based on *correct by construction* [107], effectively: design it right first time, supported by numerous rigorous techniques, such as Formal Methods, to manage error. Not coincidentally, these are *exactly* the right methods to ensure code is both dependable and scrutable, as is required for supporting reproducibility and quality science more generally. Conversely, not following these practices undermines the rigor of science.

6.1. Software Engineering Boards

Misuse of data, exploiting vulnerable people, and not obtaining informed consent are typical ethical problems. Planned research may be ethically unacceptable in ways the investigators do not anticipate: few people have the objectivity and ethical expertise to make sound ethical judgments, particularly when it comes to assessing their own work. National funders, and others, therefore require Ethics Boards to formally review ethical quality. Medical journals will not publish research that has not undergone independent formal ethical review.

Analogously, and supplementing Ethics Boards, it is argued here that Software Engineering Boards (SEBs) would authorize as well as provide advice to guide the implementation of quality Software Engineering to support research and publication processes. Just as journals require conflicts of interest statements, data availability statements and ethics board clearance, we should move to scientific papers and funded research being required to include formal Software Engineering Board statements. Note that Software Engineers themselves have a code of ethics that applies to their own work [108].

Some journals have policies that code is to be made available (see Supplemental Material), but they should require that code is not just available in principle but *actually works* on the relevant data. The authors should test a clean deployed build of their code and save the results. Presumably a paper's authors must have run their code successfully on *some* data at least once, so preparing the code and data in a way that is reproducible should be a routine and uncontentious part of the rigorous development of code underpinning *any* scientific claims. This requirement is no more unreasonable than requesting good statistics, as discussed earlier. And the solution is the same: that relevant experts — statisticians or Software Engineers — need to be available and engaged with the science. Software Engineering Board statements would be a straight forward way of helping achieve this and showing that it has been done adequately.

There need to be many SEBs to ensure convenient access, potentially at least one per university. Active, professional Software Engineers should be on these SEBs; this is not a job for people who are not qualified and experienced in the area or who are not actively connected with the state of the art. There are many high-quality university computer science departments and software companies (especially those in safety-critical areas like aviation and nuclear power) who would be willing and competent to help.

As appropriate, SEBs might require version control, unit testing, static analysis and other quality control methods. Within the field of Software Engineering itself, publishers are already developing rigorous badging initiatives to indicate the level of formal review of the quality of software [109].

A potential argument against SEBs is that they may become onerous, onerous to run and onerous to comply with their requirements. A more balanced view is that SEBs need their processes to be adaptable and proportionate; indeed, few people consider Ethics Boards to be disproportionately onerous. If software being developed is of low risk, then less stringent engineering is required than if the software could cause frequent and critical outcomes, say in their impact on public health policy for a nation. Hence SEBs processes are likely to follow a risk analysis, perhaps starting with a simple checklist. There are standard ways to do this, such as following IEC 61508:2010 [110, 111] or similar. Following a risk analysis (based on safety assurance cases, controlled documents and so on as appropriate to the domain), the Board would focus scrutiny where it is beneficial without obstructing routine science.

A professional organization, such as the UK Royal Academy of Engineering ideally working in collaboration with other national international bodies such as IFIP, should be asked to develop and support a framework for SEBs. SEBs could be quickly established to provide direct access to mature Software Engineering expertise for both researchers and for journals seeking competent peer-reviewers. In addition, particularly during a pandemic or other disasters, SEBs would provide direct access to their expertise for Governments and public policy organizations. Given the urgency, this paper recommends that *ad hoc* SEBs should be established for this purpose.

SEBs are a new suggestion, providing a supportive, collaborative process. They meet Tony Hoare's comments about the value of rigorous management of procedures [112], and widen them to non-programmer scientists. Methodological suggestions already made in the literature include open-source and specific Software Engineering methodologies to improve reproducibility [76, 113]. Reference [114] provides an conceptual framework. However, there is scope for further research to provide an evidence base to motivate and assess appropriate interventions (such as those proposed in this paper) to help scientists do more rigorous and effective Software Engineering to support their research and publishing.

An analogous proposal to SEBs has been made for Methods Review Boards [115], to help scientists ensure the methods they use are appropriate for addressing their research questions. Methods Review Boards were motivated by an Ethics Board member noticing that often experimental methodologies are inadequate, which will waste time that will not be corrected until the flaws are raised, usually too late, typically during peer-review — creating technical debt (discussed below, in Section 6.6). As with SEBs, the goal of Methods Review Boards is not to gatekeep, but to improve. The paper [115] raises many of the same trade-offs that SEBs also face; indeed one would hope that Methods Review Boards would include Software Engineers or have SEBs as sub-boards or *vice versa*: Software Engineering is now a key methodology of science.

6.2. Extending RAP to RAP+

Code is usually seen as an independent set of files that are used to generate results, typically to be copied into a paper; code is usually seen as a passive part of science. In reality, code is very creative. A paper can itself embed code or become code [e.g. 34, 116], as discussed in Section 2.2: code then becomes a driver for the research. This view supports the generalization of RAP to form RAP+. Essentially, RAP+ is the recognition that coding is not just about programming computers (which results in RAP) but is about applying computational thinking [7, 8] that supports and constructively analyzes any process, in particular the creative scientific processes of doing science and creating archival publications.

Once workflow steps in the pipeline are automated, then there is code to run the steps again. Once there is code, it can be managed in a version control system. A version control system then provides an audit trail for free, as well as many advantages such as being able to backtrack to an earlier version, for instance to review earlier edits. Importantly, code can also perform sanity checks on the process. A very simple example is automatic bibliography systems that check that journal names and DOIs are correct, and so forth. (Bibliographic systems also allow the bibliographic data to be pooled and curated with other scientists, which improves its scope and quality.) But RAP+ goes far beyond bibliographies; there is far more of the scientific workflow that can be partly or fully automated — and with corresponding benefits.

GitHub is a tool that provides *actions* that are named specifications to run analytic pipelines, *workflows* in GitHub's terminology. GitHub happens to specify actions in the language YAML, which, being a textual notation, in turn means that the features of GitHub — open-source, version control, and so on — can be applied to the research workflow as well. Research pipelines can thus be made explicit, documented, shared and, most importantly, critiqued and improved.

The entire scientific process can be supported by automation (especially with its interaction with the world automated by sensors, AI, and robots). There are many ways to do this; for example, *Mathematica* makes the analysis of the data and the calculations and the paper "the same thing" in its integrated notebooks. The many alternatives include R Markdown, an approach based in the open mathematical system R [117]; a system, Lepton [118], which allows a LaTeX document to execute and include arbitrary code, and language-independent notebook systems like Jupyter; and so on (section 2.2).

In all such systems, running the computational paper creates the publication. Indeed, every time the paper is run, the authors are likely to check the results and fix any problems, so an explicit RAP workflow actively helps reduce errors.

Here is the insight: the paper's code, just like the paper itself is text, so the code *itself* can fully form part of the pipeline, made reproducible, and benefit from all the usual RAP benefits. To date, this critical point has been overlooked. Since using RAP to integrate code, rather than just the conventional 'pure' scientific workflow, has not been mentioned previously, we call it RAP+ to make it clear this is a new and important generalization of RAP. RAP+ helps improve code quality, for the same reasons

RAP improves the quality of the science. Improving code quality improves the science and its reproducibility.

Software engineers have many tools for automatic code development (such as Unix's `make`) but the idea that these tools can be used to integrate and help automate code authoring as well as its documentation *and* paper authoring is radical. It means that the *entire* research and development process of the paper *including* all its underlying code can be reproduced, reused, and developed by others. The present paper is an example of RAP+; more details are given in the Supplemental Material.

By definition, RAP+ objectifies how science is done to a standard sufficient to enable a computer to run it. RAP+ therefore enables all of the methodologies of Software Engineering to be brought to bear *on the science itself*. RAP+ means the normally tacit, manual, and undocumented processes of science (especially the coding) become explicit. Code can then be scrutinized, optimized and ensured correct by standard Software Engineering practice; thus RAP+ does not automate science just for easier reproduction, it makes the automation explicit so the doing of science itself can be reasoned about — not just by scientists but supported by sophisticated tools, such as theorem proving and AI. Science will be improved by RAP+.

6.3. The paper as a scientific laboratory

The conventional view of science is that experiments are done *then* written up. However, it is more productive to think of the paper itself as an active laboratory, not just as a record of finished work. The view of the paper as a scientific laboratory is explicit in computational papers (Section 2.2): ideas are written down, and their validity is tested by the sense they make or fail to make; the ideas are then revised — writing is an active experiment to find and develop ideas that are worth saying. Viewing the paper as a laboratory encourages authors to copy and adopt laboratory best practice (such as keeping records, as RAP and RAP+ suggest) into the processes of writing the paper itself; viewing the paper as a laboratory also encourages authors to see writing as a scientifically — not just expressive — act, and not just as the final summary of a period of scientific creativity. In short, seen as a laboratory, the paper is no longer a reactive write-up of finished work, but it is an active part of doing good science. Writing a paper explores the space of scientific possibility as constructively as working in the field or on a lab bench. The computable paper is now the scientific laboratory.

With a computational paper, authors can literally experiment *in* the paper, exploring the effectiveness of ideas and explanations. Furthermore, they can experiment with hypotheses: for example, authors can make a clear claim that at that point in the lifecycle of the paper is but a wish rather than an established theory or fact — the wish enables them to sketch a direction they plan to go in and to explore possible supporting arguments and evidence for it. So they then do the experiments or calculations, including consulting the literature and other scientists, to establish a justification and other details. The evidence they generate or the criticism they receive may not be quite what they expected, so they then revise the claim to be correct, or change it to be a more realistic claim, or they could delete it if it just turns out to be a mess, or they could develop some altogether much better approach to the work as a result of the exploration.

Typically, many ideas in a paper will be linked to conventional experiments. For example a computational paper might calculate the statistical power of an experiment is too low (there is an unacceptable risk of committing Type II errors), so the authors will decide to improve the experiments. The computational paper approach allows such calculations to be made before, during, or after the experiments.

If there is code in the paper, every time it is typeset, the code will be run. Therefore the authors of the paper proof-reading the paper and the results of the code will have opportunities to debug and improve the code. Again, the paper itself is acts like a laboratory, helping the authors refine the science.

Note that systems capable of handling computational papers (including LaTeX) can create conditional documents: for example, there could be a flag `publish`. If `publish` is false, the author can see all their private work and thinking, including all their experimental thinking and workings; but when the author sets the flag `publish` to true, the paper would be typeset for a submitted paper with the detailed workings concealed to ensure a clean and concise presentation. In principle, also submitting to the journal a separate version of the paper with `publish` set to false would make the data and workings visible and thus could satisfy journal code and data requirements. Of course, when there is a large body of data or code, they need not all be an explicit in the computational paper: they would be made available separately in the usual way — the flag `publish` rather than being true/false might instead be a number setting the degree of disclosure.

The more we view the paper as a proactive scientific laboratory environment, the more we gain from the RAP+ perspective, and the more science gains from improved, conceptually broadened, reliable and reproducible science. The more we will also want to engage mature Software Engineering (and computational thinking) ideas too, because the quality and creativity of future science relies on them, after all, Software Engineering is about how to tell a computer how to do something reliably, and, as Knuth said, science is what we understand well enough to explain to a computer, and so Software Engineering can now directly help ensure we are not fooling ourselves about what we understand.

6.4. Action must be interdisciplinary

Code is more than a scientific instrument, more than a thermometer or test tube, as code makes, informs and changes decisions; indeed, code can actually *do* science. Still, code is only a part of science, so relying on SEBs *alone* would continue one of the besetting problems about the role of code in science.

The conventional view is that scientists do the hard work compared to the 'easy' coding work (Sections 4.1 and 4.3) so they just need to tell coders what to do. This is the view expressed by Landauer in his classic book *The Trouble with Computers* [119, 120], where he argues that the trouble with computers, an idea he explores at some length, is that we need to spend more effort in working out what computers should do (here, do the science better) and then *just* tell programmers to do *that*.

On the contrary, competent Software Engineers have insights into the logic, coherence, complexity and computability of what they are asked to do, and how it needs refining or optimizing — or the question changing. In other words, Software Engineers can bring important insights into the science, hence improving or changing the questions and assumptions the science relies on. This insight was widely recognized in the specialist area of numerical computation: 'here is a formula I want you to just code up' … 'but it's ill-conditioned, there is no good answer to that question.' It is not a simple sequential workflow with the expert initiative all on the left:

$$\boxed{\text{science specifies}} \rightarrow \boxed{\text{code up}} \rightarrow \boxed{\text{get results}}$$

but an iterative cycle of mutual collaboration and growing understanding, informed by Software Engineering best practice (via SEBs) *and* science, and implemented and tested-out using papers as laboratories.

In short, the way SEBs work and are used will be crucial to the success of the science they support. Software engineers can help improve the science, so it is not just a matter of asking a SEB whether some coding practices (like documentation) are satisfactory, but whether the SEB has insights into the science itself too. The SEB idea requires interdisciplinary working practices (science plus Software Engineering) with mutual respect for their contributing expertise.

6.5. Methodological statements

Scientific publishers (journals, conferences, workshops, videos, books, etc, and funders) often require an explicit methodology discussion, yet they rarely require the methodology to discuss the computational methodology — which, to the extent that anything relies on code, impacts all the other methodology and results discussions.

Many science publishers require explicit statements how the authors have conformed to appropriate methodological standards covering issues such as conflicts of interest, ethics, data access, consent, authoring, funders and other acknowledgements of support, and so forth. Conformance to PRISMA (see Section 4.3) is one such methodological standard. It would be easy for journals and funders to require equivalent types of statements on the quality of code, that is on the quality of its Software Engineering.

Studies of data access statements show that they are unreliable: some authors withdraw papers when journals request access statements [21] (indicating that journals that do not make an explicit request are likely publishing papers that will not provide access), and some authors do not respond to access requests [22]. How we help scientists who do not want to provide data or code access is one problem, but the serious issue for science is how to ensure any statements made are accurate, and any access provided is actually helpful (e.g. well-defined, versioned, etc) to support useful reproduction.

Journals and funders should provide support for hosting data and code (and any other relevant data, such as qualitative data, video, etc), and the review process must check that authors actually provide the material as they claim in the methodological and access statements. Conversely, scientists should be able to access funding to ensure data and code access, as well as funding for ongoing maintenance of the databases, which will typically require funding beyond the end of the normal funding period.

Intellectual Property (IP) is an increasing concern, both for author scientists and their sponsors who want royalties, and for other scientists wishing to freely build on the published science. Particularly concerning access to code, IP is potentially and often is in conflict with scientific openness. Methodological statements should be made concerning any IP associated with code, and to what extent this interferes with open access to code. More routine discussion should include raising known system dependencies, such as operating system, compiler or special hardware dependencies; it is also appropriate to mention standards conformance, such as to IEEE Floating-Point Arithmetic (IEEE 754).

Journal policies could start to explicitly encourage computationally reproducible science using RAP and RAP+ techniques. That is, the research's methodology itself may be a mixture of data and code. As this paper's Supplemental Material shows in section 12.d.3, many journals (e.g. *PLOS ONE* and, ironically, *IEEE Transactions on Software Engineering*) and repositories have policies that make RAP much harder or just counter-productive at the last step.

Methodological statements should be required that make clear what access rights are available for RAP or RAP+ material, as it is much more likely to raise IP issues that normal disclosures. In particular, if the authors plan on publishing a series of papers based on the same methodologies, the RAP/RAP+ access might be provided in a later paper or held under escrow by the journal or funding body.

Journals and funders often require data and code access statements, but as this paper has made clear, code is complex and it is rarely easy to understand and scrutinize even with access to substantial documentation (which is unusual). It is therefore recommended that journals and funders require *assurance arguments* [58], a familiar technique from the safety assurance domain. Assurance arguments provide a concise, high-level argument that the system does what is claimed. Assurance arguments can be more or less detailed, and more or less formal in their approach; editors and referees would have views on the level of detail and formality required for any specific contribution.

Finally, as there is no practical distinction between data and code (see Supplemental Material) and methodology (thanks to RAP), and certainly no distinctions that cannot be circumvented, journal and funder policies of code and data access should be reviewed *and unified* so that the access and methodology statements apply to all information, regardless of arbitrary classification of it as code or data (or documentation, assurance case, etc).

6.6. Training to reduce technical debt

Science has to work for other people in other places at other times, otherwise they cannot be sure they are studying, developing, or correcting the intended ideas reliably, but while working on a research project, the requirements of reproducibility are tempting to postpone or ignore altogether. It seems more expedient to 'just' get on and do the science without regard for the extra effort of ensuring reproducibility. This creates the problem known as *technical debt* [121]: the savings in effort now increase the future cost of reproduction. That is, a debt arises as the authors' savings now create higher costs for scientists later. The authors of a paper may create debt for themselves as shortcuts now increase the effort of retro-fitting reproducibility later. Indeed, what if the *post hoc* rigors of reproducibility expose previously unknown problems where the earlier shortcuts have created a now too-late-to-avoid cost authors with integrity will be obliged to pay?

There is a trade-off to balance when and how much effort to put into reproducibility. The trade-off is comparable to trade-offs in using statistics — the author may realize at a later stage that a 'significant' result depends on designing the experiment appropriately for the intended statistical claims, but now making the claim rigorously requires revising the methodology and probably improving the type of analysis too. What was an easier route to take earlier now causes a challenging and costly revision. However, since mature scientists recognize the importance of correct statistics, statistics is positioned at the forefront of their work rather than delayed and sorted out at greater cost later. Because of its recognized role, statistics is routinely in the undergraduate syllabus, so most scientists consciously make appropriate trade-offs minimizing statistical technical debt.

In contrast to statistics, reproducibility has only recently become an explicit issue, and computational tools to support it are developing rapidly; unlike statistics, very little of best reproducibility practice will be in scientists' background training.

While systems like Jupyter facilitate reproducible science, realizing this may only come after much work has been done. Unfortunately, retro-fitting the science into a reproducibility tool is a steep learning curve — no less than learning statistics from scratch. As with rigorous statistics, if the benefits of reproducibility processes were not realized at the start of a project and they have to be retrofitted, then the reworking of the science will be costly. Worse, Jupyter and similar tools are not 'just' computational notebook systems that are easy to use: they work with a raft of inter-related technologies, such as Binder, Docker, Python, MyST, Sphinx ... as well as field-specific environments like Neurolibre, as well as with (or despite) costly tools the scientists may be relying on, including proprietary systems, high performance computing resources, and subscription services.

Jupyter, and its many alternatives, are effectively lifestyle choices with dauntingly steep learning curves when they are learned on the job during research. The solution, as with statistics, is to push back learning about reproducibility and the benefits of tools to earlier in the scientific career, at the latest to the undergraduate curriculum before reproducibility-related technical debt can arise.

6.7. Benefits beyond science

Science increasingly recognizes the key supporting roles of code and computation, but many fields do not recognize computation as such as a skilled discipline, and therefore they are missing out on the leverage that comes with recognizing computation as a first class player in their activities.

For example, healthcare research is supported by computers and code, yet medical research papers remain in a traditional pre-digital culture and do not refer to code, as if code has no influence in the methodology of the science. Yet clinical practice relies on computer code (e.g. for diagnostics), so inevitably practice must use code unrelated to the code developed in research. In other words, the culture of not discussing and sharing code in research reduces its impact: putting research into practice is a reproducibility question, and if code has been down-played in the research it will be reproduced unreliably. Conversely, the critical issues (including patient safety) that tacitly assume code is more reliable than required for scientific research code do not get evaluated by researchers. The gap is wide. The problems and missed opportunities of under-valued and poorly-managed code are ubiquitous in healthcare [60]. Solutions might well be initiated through SEBs.

Another example is that numerous problems in finance have been precipitated by similar computational cultural naïvety. JP Morgan Chase (JPM) lost over $6 billion in a credit derivatives trade [122], in a costly parody of bad science. As reported in [122], traders did not understand the trades, did not monitor them, doubled down when results were poor, and did not communicate the extent of their losses. They were using manual coding methods; as the report says:

> "[...] the model operated through a series of Excel spreadsheets, which had to be completed manually, **by a process of copying and pasting data from one spreadsheet to another**. [Our emphasis.]
> [...] this individual immediately made certain adjustments to formulas in the spreadsheets he used. These changes, which were not subject to an appropriate vetting process, inadvertently introduced two calculation errors
> [...] after subtracting the old rate from the new rate, the spreadsheet divided by their sum instead of their average, as the modeler had intended."
> etc [123]

Compare the discussion here with figure 3, in a series of figures in the Supplemental Material, which illustrates the problem as it presents in many scientific papers.

The report [122] does not detail how the Excel spreadsheets were specified or coded, seemingly as unaware of Software Engineering as the traders. It was an unconsciously incompetent process.

Reviewers in JPM failed to scrutinize not just the coding, but the trades informed by the code. They passed on optimistic reports. Then there was a merry-go-round of blame: "the information communicated to the Risk Policy Committee ... did not suggest any significant problems ... there was no robust debate with the right facts at the right level about the portfolio risk." UK and US governments are now investigating fraud. Again, lasting solutions might well have been initiated through SEBs or equivalent; involving SEBs might well avoid future financial fiascos.

Without taking the lessons of improving Software Engineering to other fields, including improving and broadening the recognition and career paths for developers, there will continue to be unfortunate and unnecessary disconnects between competent software engineering and actual scientific (or medical or financial, etc) practice. Everything, from healthcare to finance — not just science — will continue to suffer because the critical contributions of dependable code, quality Software Engineering, and competent Computational Thinking are not yet recognized, understood, valued, or required.

6.8. Approaches to further work

Encouraging and informing the improvement of science and, specifically the reproducibility of science that relies on code, were the main aims of this paper. This paper raised problems and suggested some possible solutions: there are solutions, and better ones may yet be found. Although further work is desirable, any contributions can help improve science; not everything needs doing before we start.

Further work should research the efficiency, effectiveness, and quality of the various ideas proposed, such as RAP+ and Software Engineering Boards, and propose and evaluate more ideas.

Further work to extend the reach and scope of the survey with increasing scale, subject coverage, and rigor beyond the exploratory requirements of the present paper might seem worthwhile. If people feel our analysis of the problem is inadequate, better surveys may be appropriate, but recognizing that there is a problem (regardless of arguing over its scale) in practice it is more important to explore what direction to travel in. We should be focusing effort on acknowledging, understanding, assessing, managing and avoiding scientific problems — including poor reproducibility. This requires practical solutions that scientists can adopt, which itself relies on further work to examine rigorously what effective 'practical solutions' might entail.

Being primarily concerned with reproducibility, this paper avoided assessing the correctness of code used in science, not least because that without reproducibility correctness is moot — how code is managed and made available is more relevant than exactly what it does. Indeed, since none of the papers reviewed provided code specifications, it is not obvious what correctness means to practicing scientists. The balance, then, between practical correctness and formal correctness is an important research topic to pursue [112].

The present paper did not assess the correctness of code used in papers (explicitly so in Section 5.1) for several reasons. Code was not documented well-enough (even with high-level discussion in the papers) to know what 'correct' would mean, and no papers performed adequate tests, let alone provided adequate test material for independent verification (see Table 2); moreover, installing the software environments to build and test the systems — different environments for each paper — was too onerous, even when those environments were specified. The implication is that people within the relevant fields, especially referees, should promote standardized software environments to help increase the rate of reproduction and verification of results. As a matter for further research, then, it is important to develop, assess, and promote effective shared online (e.g. cloud) environments, perhaps with discipline-specific solutions, so that development and test environments are standardized, powerful enough and sufficiently accessible.

7. CONCLUSIONS

A pandemic creates unprecedented pressure and exposes problems in scientific methodology. During the COVID-19 crisis, code led epidemiological modeling, implemented track and trace and caused problems [124], modeled mutation pressures against vaccine shortages [125], and more. Code drove public policy. Code had a direct impact on the quality of life.

While this paper was originally motivated by Ferguson's public statements [e.g. 74, 75] about his high-profile COVID-19 pandemic modeling, the evidence reviewed here suggests that scientific coding practice is inadequate in every field, but particularly worrying in the context of the extreme pressures of managing a pandemic in real time. As science becomes more and more reliant on computers, we need to correspondingly improve the quality of code, the quality of code policies, the quality of Software Engineering, and the quality of all scientists' understanding of computation and how to manage its unlimited complexity.

The main challenges to mature computationally-realistic science are:

1) To manage software development to reduce the unnoticed and unknown impacts of bugs and poor programming practices that research and publications rely on. Computer code should be explicit, accessible (well-structured, etc), and adequately documented. Papers should be explicit on their software methodologies, limitations and weaknesses, just as Whitty expressed about the standards of science generally [54]. Professional software methodologies should not be ignored.
2) To use computation to help make scientific workflows and processes explicit, so that they can be reproduced, scrutinized and improved. RAP is an increasingly popular way to help do this, but as this paper points out, RAP can be generalized to RAP+ to help the computational parts of science as well, leading to a virtuous circle.
3) To support and develop the scientific community in the professional use of computation.
4) To find effective ways to promote professional software engineers being recognized and participating fully in scientific research, just as professional statisticians routinely support quality research (see Section 3).

While programming seems easy and is often taken for granted and done casually, programming *well* is very difficult [60]. Science needs coding to be done well.

We know from software research that ordinary programming is very buggy and unreliable. Without adequately specified and documented code and data, research is not open to scrutiny, let alone proper review and its quality is suspect. Some have argued that availability of code and data ensure research is reproducible, but that is naïve criterion: computer programs are easy to run and reproduce results, but being able to reproduce something of low quality does not magically make it more reliable [34, 69, 126] (see Section 5.2).

Software Engineering Boards (SEBs), as proposed in this paper, are an initial, straightforward, constructive and practical way to support and improve code- and computer-based science. If nothing else, the idea of SEBs is something to criticize and improve.

This paper's Supplemental Material summarizes relevant Software Engineering good practice that Software Engineering Boards would draw on, including discussing how and why Software Engineering helps improve code reliability, dependability and quality.

ACKNOWLEDGMENTS

The author is very grateful for comments from: Ross Anderson, Nicholas Beale, Ann Blandford, Paul Cairns, Rod Chapman, José Corréa de Sà, Paul Curzon, Jeremy Gibbons, Richard Harvey, Will Hawkins, Konrad Hinsen, Ben Hocking, Daniel Jackson, Peter Ladkin, Bev Littlewood, Paolo Masci, Stephen Mason, Robert Nachbar, Martin Newby, Patrick Oladimeji, Claudia Pagliari, Simon Robinson, Jonathan Rowanhill, John Rushby, Susan Stepney, Isaac Thimbleby, Prue Thimbleby, Will Thimbleby, Martyn Thomas and Ben Wilson. The author also thanks anonymous referees who also contributed to the quality of this paper.

SUPPLEMENTARY MATERIAL

Supplementary material is available at www.comjnl.oxfordjournals.org. Updates and errata are included in the Supplementary Material, which is kept up to date at github.com/haroldthimbleby/improving-science.

DATA AND CODE ACCESS

The Supplemental Material presents all data in human-readable form, and there is an extended discussion of methodology in section 10. Additionally, all data, code and documentation, including the LaTeX source, are available online at github.com/haroldthimbleby/improving-science.

The raw data are encoded in JSON. JavaScript code checks against 30 possible classes of error, and converts the JSON data into LaTeX, thus making it trivial to typeset results reliably in this paper and the Supplemental Material *directly* from the analysis. In addition, a standard CSV file is generated in case this is convenient, for instance to browse directly in a spreadsheet or to import into other programs.

AUTHOR CONTRIBUTION

Harold Thimbleby is the sole author. An preliminary outline of this paper, containing no supplementary material or data, was submitted to the UK Parliamentary Science and Technology Select Committee's inquiry into UK Science, Research and Technology Capability and Influence in Global Disease Outbreaks, under reference LAS905222, 7 April, 2020. The evidence, which was not peer-reviewed and is only available after an explicit search, briefly summarizes the case for Software Engineering Boards, but without the detailed analysis and case studies of the literature, etc, that are in the present paper. It is available to the public [127, 128].

FUNDING

This work was jointly supported by See Change (M&RA-P), Scotland (an anonymous funder), by the Engineering and Physical Sciences Research Council [grant EP/M022722/1], by the Royal Academy of Engineering through the Engineering X Pandemic Preparedness Programme [grant EXPP2021\1\186] and by Assuring Autonomy International Programme, Assuring Safe AI in Ambulance Service Triage. The funders had no involvement in the research or in this paper.

REFERENCES

1. Petkovsek, M., Wilf, H., and Zeilberger, D. (1996) $A = B$, A K Peters. Ltd. www2.math.upenn.edu/~wilf/AeqB.html
2. Quindlen, A. (2022) *Write for Your Life*. Random House.
3. Abelson, R.P. (1995) *Statistics as Principled Argument*. Lawrence Erlbaum Associates.
4. Editorial (2023) Tools such as ChatGPT threaten transparent science; here are our ground rules for their use. *Nature*, **613**, 612. https://doi.org/10.1038/d41586-023-00191-1.
5. Sommerville, I. (2015) *Software Engineering* (10th edn). Pearson.
6. Knight, J. (2012) *Fundamentals of Dependable Computing for Software Engineers*. CRC Press.
7. Wing, J.M. (2008) Computational thinking and thinking about computing. *Philos. Trans. R. Soc. A: Math., Phys. Eng. Sci.*, **366**, 3717–3725. https://doi.org/10.1098/rsta.2008.0118.
8. McOwen, P.W. and Curzon, P. (2020) *The Power of Computational Thinking*. World Scientific Publishing.
9. Ferguson, N.M. et al. (2020) Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand. www.imperial.ac.uk/media/imperial-college/medicine/sph/ide/gida-fellowships/Imperial-College-COVID19-NPI-modelling-16-03-2020.pdf.
10. Ferguson, N.M., Cummings, D.A.T., Fraser, C., Cajka, J.C., Cooley, P.C. and Burke, D.S. (2005) Strategies for containing an emerging influenza pandemic in Southeast Asia. *Nature*, **437**, 209–214. https://doi.org/10.1038/nature04017.
11. Ferguson, N.M., Cummings, D.A.T., Fraser, C., Cajka, J.C., Cooley, P.C. and Burke, D.S. (2006) Strategies for mitigating an influenza pandemic. *Nature*, **442**, 448–452. https://doi.org/10.1038/nature04795.
12. Baker, M. (2016) 1,500 scientists lift the lid on reproducibility. *Nature*, **533**, 452–454. https://doi.org/10.1038/533452a.
13. Rougier, N.P. et al. (2017) Sustainable computational science: the ReScience initiative. *PeerJ Comput. Sci.*, **3**, e142. https://doi.org/10.7717/peerj-cs.142.
14. Chang, H. (2007) *Inventing Temperature: Measurement and Scientific Progress*. Oxford Studies in the Philosophy of Science.
15. Hippel, M. von (2022) Crucial computer program for particle physics at risk of obsolescence. *Quanta Magazine*. www.quantamagazine.org/crucial-computer-program-for-particle-physics-at-risk-of-obsolescence-20221201.
16. Bemer, R.W. (1958) Techniques department: policy statement. *Commun. ACM*, **1**, 5–7.
17. Hoare, C.A.R. (2007) The ideal of program correctness: third computer journal lecture. *Comput. J.*, **50**, 254–260. https://doi.org/10.1093/comjnl/bxl078.
18. Pimentel, J. F., Murta, L., Braganholo, V., and Freire, J. (2019) A large-scale study about quality and reproducibility of Jupyter notebooks, In*2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 507–517. doi:https://doi.org/10.1109/MSR.2019.00077.
19. Trisovic, A., Lau, M.K., Pasquier, T. and Crosas, M. (2022) A large-scale study on research code quality and execution nature research. *Scientific Data*, **9**. https://doi.org/10.1038/s41597-022-01143-6.
20. Thimbleby, H. (2004) *Give Your Computer's IQ a Boost — Journal of Machine Learning Research*. Times Higher Education Supplement. www.timeshighereducation.co.uk/story.asp?sectioncode=26&storycode=176549.
21. Miyakawa, T. (2022) No raw data, no science: another possible source of the reproducibility crisis. *Mol. Brain*, **13**, 1–6. https://doi.org/10.1186/s13041-020-0552-2.
22. Gabelica, M., Bojčić, R. and Puljak, L. (2020) Many researchers were not compliant with their published data sharing statement: mixed-methods study. *J. Clin. Epidemiol*. https://doi.org/0.1016/j.jclinepi.2022.05.019.
23. Munafò, M.R., Nosek, B.A., Bishop, D.V.M., Button, K.S., Chambers, C.D., Percie du Sert, N., Simonsohn, U., Wagenmakers, E.-J., Ware, J.J. and Ioannidis, J.P.A. (2017) A manifesto for reproducible science. *Nat. Hum. Behav.*, **1**, 0021. https://doi.org/10.1038/s41562-016-0021.
24. Smith, A.M. et al. (2018) Journal of open source software (JOSS): design and first-year review. *PeerJ Comput. Sci.*, **4**, e147. https://doi.org/10.7717/peerj-cs.147.
25. Nosek, B.A. et al. (2015) Promoting an open research culture: author guidelines for journals could help to promote transparency, openness, and reproducibility. *Science*, **348**, 1422–1425. https://doi.org/10.1126/science.aab2374.
26. Alter, G. et al. (2022) *Guidelines for Transparency and Openness Promotion (TOP) in journal policies and practices*. The TOP Guidelines. osf.io/9f6gx/wiki/Guidelines.
27. Godlee, F., Smith, J. and Marcovitch, H. (2011) Wakefield's article linking MMR vaccine and autism was fraudulent. *BMJ*, **342**. https://doi.org/10.1136/bmj.c7452.
28. Fanelli, D. (2009) How many scientists fabricate and falsify research? A systematic review and meta-analysis of survey data. *PloS One*, **4**, e5738. https://doi.org/10.1371/journal.pone.0005738.
29. Machina, H.K. and Wild, D.J. (2013) Electronic laboratory notebooks progress and challenges in implementation. *J. Lab. Autom.*, **18**, 264–268. https://doi.org/0.1177/2211068213484471.
30. Perkel, J.M. (2021) Reactive, reproducible, collaborative: computational notebooks evolve. *Nature*, **593**, 156–157. https://doi.org/10.1038/d41586-021-01174-w.
31. Akhlaghi, M., Infante-Sainz, R., Roukema, B.F., Khellat, M., Valls-Gabaud, D. and Baena-Gallé, R. (2021) Toward long-term and archivable reproducibility. *Comput. Sci. Eng.*, **23**, 82–91. https://doi.org/10.1109/mcse.2021.3072860. https://maneage.org.
32. Knuth, D.E. (1984) Literate programming. *Comput. J.*, **27**, 97–111. https://doi.org/10.1093/comjnl/27.2.97.
33. Knuth, D.E. (1992) Literate programming. In *of CSLI Lecture Notes*, Vol. **27**, Center for the Study of Language and Information Publication, Stanford, CA.
34. Thimbleby, H. and Williams, D. (2018) A tool for publishing reproducible algorithms & a reproducible, elegant algorithm for sequential experiments. *Sci. Comput. Program.*, **156**, 45–67. GitHub.com/haroldthimbleby/relit. https://doi.org/10.1016/j.scico.2017.12.010.

35. Gray, T.W. and Wolfram, S. (2013) Method and system for presenting input expressions and evaluations of the input expressions on a workspace of a computational system. US Patent no. 8,407,580 B2.
36. Granger, B.E. and Pérez, F. (2021) Jupyter: thinking and storytelling with code and data. *Comput. Sci. Eng.*, **23**, 7–14. https://doi.org/10.1109/MCSE.2021.3059263.
37. Xie, Y. (2015) *Dynamic Documents with R and knitr* (2nd edn). CRC.
38. Thimbleby, H. (1999) Specification-led design for interface simulation, collecting use-data, interactive help, writing manuals, analysis, comparing alternative designs, etc. *Personal Technol.*, **4**, 241–254. https://doi.org/10.1007/BF01885563. harold.thimbleby.net/ansim.
39. Office for National Statistics (2022) *Using reproducible analytical pipelines (RAP) to improve statistics*. https://code.statisticsauthority.gov.uk/case-studies/using-reproducible-analytical-pipelines-rap-to-improve-statistics/.
40. Upson, M. (2017) *Reproducible analytical pipelines*. dataingovernment.blog.gov.uk/2017/03/27/reproducible-analytical-pipeline.
41. Goldacre, B. (2022) *Better, broader, safer: using health data for research and analysis*. Department of Health and Social Care. www.gov.uk/government/publications/better-broader-safer-using-health-data-for-research-and-analysis.
42. Ainsworth, R. et al. (2022) *The Turing Way: A Handbook for Reproducible Data Science*, Vol. **v1.0.3**, Zenodo. the-turing-way.netlify.app/welcome, doi:https://doi.org/10.5281/zenodo.3233853.
43. Courtès, L. (2020) [re] storage tradeoffs in a collaborative backup service for mobile devices. *Rescience C*, **6**, 10. https://doi.org/10.5281/zenodo.3886739. https://gitlab.inria.fr/lcourtes-phd/edcc-2006-redone.
44. Glen, S. (2022) Reporting statistics APA style. *Statistics How To*. www.statisticshowto.com/probability-and-statistics/reporting-statistics-apa-style.
45. Cichoń, M. (2020) Reporting statistical methods and outcome of statistical analyses in research articles. *Pharmacol. Rep.*, **72**, 481–485. https://doi.org/10.1007/s43440-020-00110-5.
46. Cichoń, M. (2022) *Science Journals: Editorial Policies*. https://www.science.org/content/page/science-journals-editorial-policies.
47. Richards, D. et al. (2020) A pragmatic randomized waitlist-controlled effectiveness and cost-effectiveness trial of digital interventions for depression and anxiety. *Nat. Digital Med.*, **3**, 85. https://doi.org/10.1038/s41746-020-0293-8.
48. Spiegelhalter, D. (2019) *The Art of Statistics*. Learning from Data, Pelican Books.
49. Cairns, P. (2007) HCI . . . not as it should be: inferential statistics in HCI research, In *BCS-HCI '07: Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI . . . but not as we know it*, Vol. **1**, pp. 195–201.
50. Johnson, V.E. (2013) Revised standards for statistical evidence. *Proc. Natl. Acad. Sci.*, **110**, 19313–19317. https://doi.org/10.1073/pnas.1313476110.
51. Shneiderman, B. (2016) Opinion: the dangers of faulty, biased, or malicious algorithms requires independent oversight. *Proc. Natl. Acad. Sci.*, **113**, 13538–13540. https://doi.org/10.1073/pnas.1618211113.
52. Friedman, B. and Nissenbaum, H. (1996) Bias in computer systems. *ACM Trans. Inform. Syst.*, **14**, 330–347. https://doi.org/10.1145/230538.230561.
53. Laurain, E., Ayav, C., Erpelding, M.-L., Kessler, M., Briançon, S., Brunaud, L. and Frimat, L. (2022) Targets for parathyroid hormone in secondary hyperparathyroidism: is a "one-size-fits-all" approach appropriate? A prospective incident cohort study. *BMC Nephrol.*, **15**, 132. https://doi.org/10.1186/1471-2369-15-132.
54. Whitty, C.J.M. (2015) What makes an academic paper useful for health policy? *BMC Med.*, **13**, 301. https://doi.org/10.1186/s12916-015-0544-8.
55. Hawkins, D.M. (2004) The problem of overfitting. *J. Chem. Inf. Model.*, **44**, 1–12. https://doi.org/10.1021/ci0342472.
56. May, R.M. (1976) Simple mathematical models with very complicated dynamics. *Nature*, **261**, 459–467. https://doi.org/10.1038/261459a0.
57. Dyson, F. (2004) A meeting with Enrico Fermi. *Nature*, **427**, 297. https://doi.org/10.1038/427297a.
58. Habli, I., Alexander, R., Hawkins, R., Sujan, M., McDermid, J., Picardi, C. and Lawton, T. (2020) Enhancing COVID-19 decision making by creating an assurance case for epidemiological models. *BMJ Health Care Inform.*, **27**, 1–5. https://doi.org/10.1136/bmjhci-2020-100165.
59. Kelly, D. and Sanders, R. (2008) *Assessing the quality of scientific software*, first international workshop on software engineering for computational science and engineering (see [128]). *Leipzig.*, citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.526.5076.
60. Thimbleby, H. (2021) *Fix IT: How to See and Solve the Problems of Digital Healthcare*. Oxford University Press.
61. Roth, M.T. (2022) *Laws of Hammurabi*. University of Central Florida, Open Educational Resources for the Ancient Near East. https://stars.library.ucf.edu/cgi/viewcontent.cgi?article=1133&=&context=ancientneareast&=&sei-redir=1&referer=https%253A%252F%252Fscholar.google.com%252Fscholar%253Fhl%253Den%2526as_sdt%253D0%25252C5%2526q%253DIf%252Ba%252Bbuilder%252Bconstructs%252Ba%252Bhouse%252Bhammurabi%2526btnG%253D#search=%22If%20builder%20constructs%20house%20hammurabi%22.
62. Katz, D.S. et al. (2021) Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved] previously titled: "the importance of software citation". *F1000Research*, **9**. https://doi.org/10.12688/f1000research.26932.2.
63. Page, M.J. et al. (2021) The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. *Syst. Rev.*, **10**, 1–11. https://doi.org/10.1186/s13643-021-01626-4.
64. Thimbleby, H. (2016) Human factors and missed solutions to Enigma design weaknesses. *Cryptologia*, **40**, 177–202. https://doi.org/10.1080/01611194.2015.1028680.
65. Sayburn, A. (2020) Covid-19: experts question analysis suggesting half UK population has been infected. *BMJ*, **368**, m1216. https://doi.org/10.1136/bmj.m1216.
66. Wynants, L. et al. (2020) Prediction models for diagnosis and prognosis of covid-19 infection: systematic review and critical appraisal. *BMJ*, **369**. https://doi.org/10.1136/bmj.m1328.
67. Knuth, D.E. (1998) *The Art of Computer Programming (Seminumerical Algorithms,* Vol. **2**, 3rd edn. Addison-Wesley.
68. Jackson, D. (2021) *The Essence of Software*. Princeton University Press.
69. Popper, K.R. (2002) *Conjectures and Refutations: The Growth of Scientific Knowledge* 2nd edn. Routledge.
70. Sridhar, D. (2022) *Preventable: How a Pandemic Changed the World & How to Stop the Next One*. Viking.
71. Heesterbeek, H. et al. (2015) Modeling infectious disease dynamics in the complex landscape of global health. *Science*, **347**, 265–270. https://doi.org/10.1126/science.aaa4339.

72. Moons, K.G., Altman, D.G., Reitsma, J.B., Ioannidis, J.P., Macaskill, P., Steyerberg, E.W., Vickers, A.J., Ransohoff, D.F. and Collins, G.S. (2015) Transparent reporting of a multivariable prediction model for individual prognosis or diagnosis (TRIPOD): explanation and elaboration. *Ann. Intern. Med.*, **162**, W1–W73. https://doi.org/10.7326/M14-0698.
73. Adam, D. (2020) Modelling the pandemic: the simulations driving the world's response to COVID-19. *Nature*, **580**, 316–318. https://doi.org/10.1038/d41586-020-01003-6.
74. Ferguson, N. (2020) *Tweet*. twitter.com/neil_ferguson/status/1241835454707699713.
75. Leake, J. (2020) Neil Ferguson interview: no 10's infection guru recruits game developers to build coronavirus pandemic model. *The Sunday Times*. www.thetimes.co.uk/article/neil-ferguson-interview-no-10s-infection-guru-recruits-game-developers-to-build-coronavirus-pandemic-model-zl5rdtjq5.
76. Hinsen, K. (2013) Software development for reproducible research. *Comput. Sci. Eng.*, **15**, 60–63. https://doi.org/10.1109/MCSE.2013.91.
77. Smith, B. (2020) SAGE adviser Neil Ferguson quits over coronavirus lockdown breach. *Civil Service World*, www.civilserviceworld.com/professions/article/sage-adviser-neil-ferguson-quits-over-coronavirus-lockdown-breach#:~:text=SAGE%20adviser%20Neil%20Ferguson%20quits%20over%20coronavirus%20lockdown%20breach,-Epidemiologist%20says%20he&text=Professor%20Neil%20Ferguson%2C%20one%20of,resigned%20after%20breaching%20lockdown%20rules.
78. Ahrens, J.H. and Dieter, U. (1973) Extensions of Forsythe's method for random sampling from the normal distribution. *Math. Comput.*, **27**, 927–937. https://doi.org/10.1090/S0025-5718-1973-0329190-8.
79. Ahrens, J.H. and Dieter, U. (1972) Computer methods for sampling from the exponential and normal distributions. *Commun. ACM*, **15**, 873–882. https://doi.org/10.1145/355604.361593.
80. The MISRA Consortium Limited (2020) *MISRA Compliance: 2020 – Achieving compliance with MISRA Coding Guidelines*. MISRA Consortium., Norwich, Norfolk, England. www.misra.org.uk.
81. Barnes, J. (2003) *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley.
82. O'Sullivan, B., Stewart, D. and Goerzen, J. (2008) Real World Haskell. *O'Reilly Media*. book.realworldhaskell.org.
83. Chawla, D.S. (2020) Critiqued coronavirus simulation gets thumbs up from code-checking efforts. *Nature*, **582**, 323–324 www.nature.com/articles/d41586-020-01685-y.
84. Scheuber, A., and Elsland, S. L van. (2020) *Codecheck Confirms Reproducibility of COVID-19 Model Results*. Imperial College, London. www.imperial.ac.uk/news/197875/codecheck-confirms-reproducibility-covid-19-model-results
85. Eglen, S. J. (2020) *CODECHECK Certificate 2020–010 for Paper: Report 9: Impact of Non-pharmaceutical Interventions (NPIs) to Reduce COVID-19 Mortality and Healthcare Demand*. github.com/codecheckers/covid-report9, doi: https://doi.org/10.5281/zenodo.3865491.
86. Hatton, L. and Roberts, A. (1994) How accurate is scientific software? *IEEE Trans. Softw. Eng.*, **20**, 785–797. https://doi.org/10.1109/32.328993.
87. Halloran, M.E. et al. (2008) Modeling targeted layered containment of an influenza pandemic in the United States. *Proc. Natl. Acad. Sci.*, **105**, 4639–4644. https://doi.org/10.1073/pnas.0706849105. www.pnas.org/content/105/12/4639.
88. Schnell, S. (2015) Ten simple rules for a computational biologist's laboratory notebook. *PLoS Comput. Biol.*, **11**, e1004385. https://doi.org/10.1371/journal.pcbi.1004385.
89. National Institutes of Health. 2020, effective date January 25,2023. *Final NIH Policy for Data Management and Sharing*, Vol. **NOT-OD-21-013**, Office of The Director, National Institutes of Health. grants.nih.gov/grants/guide/notice-files/NOT-OD-21-013.html.
90. Kozlov, M. (2022) NIH issues a seismic mandate: share data publicly. *Nature*, www.nature.com/articles/d41586-022-00402-1.
91. Ferguson, N. (2020) *Tweet*. twitter.com/neil_ferguson/status/1241835456947519492
92. Richards, D. and Boudnik, K. (2020) Neil Ferguson's Imperial model could be the most devastating software mistake of all time. *The Telegraph.*, www.telegraph.co.uk/technology/2020/05/16/neil-fergusons-imperial-model-could-devastating-software-mistake.
93. Zlojutro, A., Rey, D. and Gardner, L. (2019) A decision-support framework to optimize border control for global outbreak mitigation. *Nat. Sci. Rep.*, **9**. https://doi.org/10.1038/s41598-019-38665-w.
94. Thimbleby, H. (2003) The directed Chinese postman problem. *Software — Practice & Experience*, **33**, 1081–1096. https://doi.org/10.1002/spe.540. harold.thimbleby.net/cpp/index.html.
95. Sander, B., Nizam, A., Garrison, Jr. L.P., Postma, M.J., Halloran, M.E. and Longini, I.M.Jr. (2009) Economic evaluation of influenza pandemic mitigation strategies in the US using a stochastic microsimulation transmission model. *Value Health*, **12**, 226–233. https://doi.org/10.1111/j.1524-4733.2008.00437.x.
96. Perkel, J.M. (2022) How to fix your scientific coding errors. *Nature*, **602**, 172–173. https://doi.org/10.1038/d41586-022-00217-0.
97. Guest, O. and Martin, A.E. (2021) How computational modeling can force theory building in psychological science. *Perspect. Psychol. Sci.*, **16**, 789–802. https://doi.org/10.1177/1745691620970585.
98. ACT-R Research Group (2022) *ACT-R*. act-r.psy.cmu.edu/software.
99. Ladkin, P.B., Littlewood, B., Thimbleby, H. and Thomas, M. (2020) The Law Commission presumption concerning the dependability of computer evidence. *Digital Evid. Electron. Sign. Law Rev.*, **17**, 1–14. https://doi.org/10.14296/deeslr.v17i0.5143.
100. Hamming, R.W. (1987) *Numerical Methods for Scientists and Engineers*. Dover Publications Inc.
101. Kissler, S.M., Tedijanto, C., Goldstein, E., Grad, Y.H. and Lipsitch, M. (2020) Projecting the transmission dynamics of SARS-CoV-2 through the postpandemic period. *Science*, **368**, 860–868. https://doi.org/10.1126/science.abb5793.
102. Verity et al. (2020) Estimates of the severity of coronavirus disease 2019: a model-based analysis. *Lancet*, **20**, 669–677. https://doi.org/10.1016/S1473-3099(20)30243-7.
103. Hoare, C.A.R. (1969) An axiomatic basis for computer programming. *Commun. ACM*, **12**, 576–580. https://doi.org/10.1145/363235.363259.
104. Freedman, L.P., Cockburn, I.M. and Simcoe, T.S. (2015) The economics of reproducibility in preclinical research. *PLoS Biol.*, **13**, e1002165. https://doi.org/10.1371/journal.pbio.1002165.
105. Wilkinson, M.D. et al. (2016) The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, **3**, 1–9. https://doi.org/10.1038/sdata.2016.18.
106. Wood-Charlson, E.M., Crockett, Z., Erdmann, C., Arkin, A.P. and Robinson, C.B. (2022) Ten simple rules for getting and giving credit for data. *PLoS Comput. Biol.*, **18**, e1010476. https://doi.org/10.1371/journal.pcbi.1010476.

107. Woodcock, J.C.P., Larsen, P.G., Bicarregui, J.C. and Fitzgerald, J.S. (2009) Formal methods: practice and experience. *ACM Comput. Surv.*, **41**, 1–36. https://doi.org/10.1145/1592434.1592436.
108. ACM (2020) *Code of Ethics and Professional Conduct*, ACM. Accessed April 23, 2020.www.acm.org/code-of-ethics.
109. ACM (2020) *Artifact Review and Badging — Current*, **Version 1.1**, ACM. www.acm.org/publications/policies/artifact-review-and-badging-current.
110. Redmill, F. (2000) Understanding the use, misuse and abuse of safety integrity levels. In *Lessons in System Safety, Eighth Safety-critical Systems Symposium*. Newcastle University, Revised. homepages.cs.ncl.ac.uk/felix.redmill/publications/1%20SILs.pdf.
111. IEC Technical Committee TC 65 (2010) *IEC 61508:2010 CMV commented version, functional safety of electrical/electronic/programmable electronic safety-related systems*. webstore.iec.ch/publication/22273.
112. Hoare, C. A. R. (1996) How did software get so reliable without proof? *Lecture Notes in Computer Science*, vol. **1051**, Springer, pp. 1–17. doi:https://doi.org/10.1007/3-540-60973-3_77.
113. Fomel, S. (2015) Reproducible research as a community effort: lessons from the Madagascar project. *Comput. Sci. Eng.*, **17**, 20–26. https://doi.org/10.1109/MCSE.2014.94.
114. Stol, K.-J. and Fitzgerald, B. (2018) The ABC of software engineering research. *ACM Trans. Software Eng. Methodol.*, **27**, 1–51. https://doi.org/10.1145/3241743.
115. Lakens, D. (2023) Methods-review boards could avert wasted research. *Nature*, **613**.
116. Gabriela, A. and Capone, R. (2011) Executable paper grand challenge workshop. *Proc. Comput. Sci.*, **4**, 577–578. https://doi.org/10.1016/j.procs.2011.04.060.
117. Xie, Y., Allaire, J.J. and Grolemund, G. (2020) *R Markdown: The Definitive Guide*. Chapman & Hall/CRC.
118. Li-Thiao-Té, S. (2012) Literate program execution for reproducible research and executable papers. *Proc. Comput. Sci.*, **9**, 439–448 . International Conference on Computational Science, ICCS 2012. https://doi.org/10.1016/j.procs.2012.04.047.
119. Landauer, T.K. (1995) *The Trouble with Computers: Usefulness, Usability, and Productivity*. MIT Press.
120. Thimbleby, H. (1996) The trouble with computers: usefulness, usability, and productivity (by Thomas K. Landauer). *Comput. Linguist.*, **22**, 265–276.
121. Falessi, D., and Kruchten, P. (2015) Five reasons for including technical debt in the software engineering curriculum, *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW'15, ACM, pp. 28:1–28:4. https://doi.org/10.1145/2797433.2797462.
122. Heineman, B.W.Jr. (2013) The JP Morgan "whale" report and the ghosts of the financial crisis. *Harv. Bus. Rev.*, https://hbr.org/2013/01/the-jp-morgan-whale-report-and.
123. Cavanagh, M. (ed) (2013) *Report of JPMorgan Chase & Co. Management Task Force Regarding 2012 CIO Losses*. JPMorgan Chase & Co.
124. Thimbleby, H. (2020) The problem isn't Excel, it's unprofessional software engineering. *BMJ*, **371**. https://doi.org/10.1136/bmj.m4181.
125. Wadman, M. (2021) Could too much time between doses drive the coronavirus to outwit vaccines? *Science*. https://doi.org/10.1126/science.abg5655.
126. Benureau, F.C.Y. and Rougier, N.P. (2018) Re-run, repeat, reproduce, reuse, replicate: transforming code into scientific contributions. *Front. Neuroinform.*, **11**. https://doi.org/10.3389/fninf.2017.00069.
127. House of Commons Science and Technology Committee (2020) *The UK response to covid-19: use of scientific advice*. committees.parliament.uk/publications/4165/documents/41300/default.
128. Thimbleby, H. (2020) *Written Evidence Submitted by Harold Thimbleby to The UK response to covid-19: Use of scientific advice, (C190005)*, House of Commons Science and Technology Committee (see [127]). committees.parliament.uk/work/91/default/publications/written-evidence/?SearchTerm=thimbleby.
129. Carver, J.C. (2009) First international workshop on software engineering for computational science & engineering. *Comput. Sci. Eng.*, **11**, 7–11. https://doi.org/10.1109/MCSE.2009.30.