Improving science that uses code

Harold Thimbleby

See Change Fellow in Digital Health, Swansea University
Email: 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+)

Received 30 April 2022; revised 8 June 2022, 3 January 2023, 31 January 2023 & 12 May 2023

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

CONTENTS

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 COVID-19 pandemic. A pilot survey implies that such problems are ubiquitous and by no means limited to epidemiology: see tables 1 & 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.a): 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 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

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

TABLE 1. Overview of peer-reviewed paper sample, broken down further in Table 2. Survey methodology and data is 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.a.)

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

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.

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.b.

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].

[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.

2.a. 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 9 : 000 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 & 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, 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 to 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 URL `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.d.

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 (URL `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 under-estimates.[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.b. 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

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

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.a**, 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 TeX, 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 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.c). 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 URL `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.c. 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.

[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.

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

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.

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.

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

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.

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.c.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.c.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.b), which creates a vicious cycle of ignoring code, leading to ignoring the critical — and non-trivial — role of correct code in science.

3.a. 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 (figure 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 $x = 4$.

[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.

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 figure 1.

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.a) 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

[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.

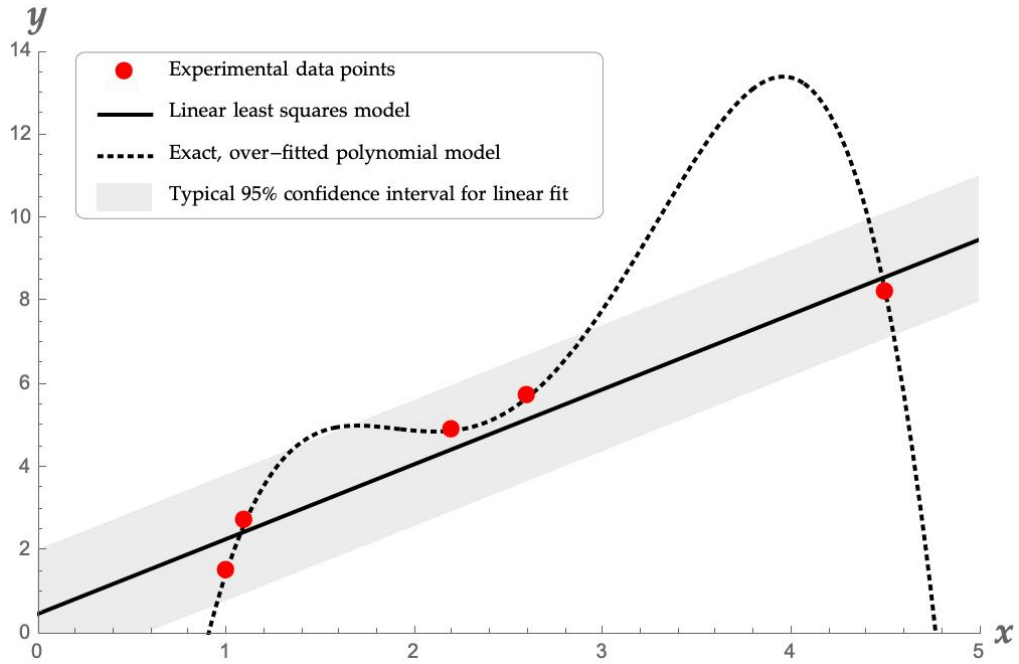


FIGURE 1. Much computational science is concerned with finding plausible multi-dimensional models that fit models to data with the aim of extrapolating or predicting new results from them. Shown here is notional sample of experimental 2D data (the dots), a linear least squares regression, and an exact polynomial model. The over-fitted polynomial model fits the sample *exactly*, but since the experimental data is 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.

improved science.

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.a. 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.b. 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.a), 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.)

4.c. 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 $N = 2\,000$. 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 $2,000$ or as $2\,000.0$, or if a number containing the digits 2000, like 12000, 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.d. 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 num-

bers 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 contains 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 naïïve 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.e. 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.a, 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.b 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.a. 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.

[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.)

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.a); 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.a), rather than to objectively elucidate the science.

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.b. 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 summarise 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

[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.

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 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

[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., 92], supporting the concerns expressed in the present paper. n-source, available at URL `github.com/mrc-ide/covid-sim` version (19 July 2021).

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.c. 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 1,000 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.d. The lop-sided emphasis on data

Data has 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 is 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

[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 URL `datadryad.org` curates raw, unprocessed data. At the time of writing, Dryad excludes code; however, it uses a separate organization, Zenodo URL `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.d).

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.

There is a lop-sided emphasis on data in science. In fact, data is 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 is routinely formatted in simple or standard ways, but code, in contrast, is 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.a. 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 judgements, 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.f). 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.b. 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.b: 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.b).

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.c. 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.b): 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.

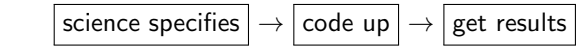
The more we view the paper as a proactive scientific laboratory environment, the more we gain from the RAP+ perspective, and the more science gains from improved, conceptually broadened, reliable, and reproducible science. The more we will also want to engage mature Software Engineering (and computational 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.d. 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.a & 4.c) 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:



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.e. 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.c) 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 on-going 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.f. 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.g. 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.h. 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.a) 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.b).

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.

SUPPORTING INFORMATION

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.

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, is available online at URL `github.com/haroldthimbleby/improving-science`

The raw data is 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].

Competing interests The author declares no competing interests.

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. URL `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. DOI `10.1038/d41586-023-00191-1`

[5] Sommerville, I. (2015) *Software Engineering*, Pearson, 10th ed.

[6] Knight, J. (2012) *Fundamentals of Dependable Computing for Software Engineers*, CRC Press.

[7] Wing, J. M. (2008) "Computational thinking and thinking about computing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **366**(1881):3717–3725. DOI `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.* (16 March 2020) *Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand.* URL `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. DOI `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. DOI `10.1038/nature04795`

[12] Baker, M. (2016) "1,500 scientists lift the lid on reproducibility," *Nature*, **533**(7604):452–454. DOI `10.1038/533452a`

[13] Rougier, N. P. *et al.* (2017) "Sustainable computational science: the ReScience initiative," *PeerJ Computer Science*, **3**(e142). DOI `10.7717/peerj-cs.142`

[14] Chang, H. (2007) *Inventing Temperature: Measurement and Scientific Progress*, Oxford Studies in the Philosophy of Science.

[15] von Hippel, M. (2022) "Crucial computer program for particle physics at risk of obsolescence," *Quanta Magazine*. URL `www.quantamagazine.org/crucial-computer-program-for-particle-physics-at-risk-of-obsolescence-20221201`

[16] Bemer, R. W. (1958) "Techniques department: Policy statement," *Communications of the ACM*, **1**(1):5–7.

[17] Hoare, C. A. R. (2007) "The ideal of program correctness: Third Computer Journal lecture," *The Computer Journal*, **50**(3):254–260. DOI `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 `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**(60). DOI `10.1038/s41597-022-01143-6`

[20] Thimbleby, H. (9 May, 2004) "Give your computer's IQ a boost — *Journal of Machine Learning Research*," *Times Higher Education Supplement*. URL `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," *Molecular Brain*, **13**(24):1–6. DOI `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," *Journal of Clinical Epidemiology*. DOI `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," *Nature Human Behaviour*, **1**(1):0021. DOI 10.1038/s41562-016-0021

[24] Smith, A. M. *et al.* (2018) "Journal of open source software (JOSS): Design and first-year review," *PeerJ Computer Science*, **4**(e147). DOI 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**(6242):1422–1425. DOI 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"*. URL 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**(c7452). DOI 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**(5):e5738. DOI 10.1371/journal.pone.0005738

[29] Machina, H. K. and Wild, D. J. (2013) "Electronic laboratory notebooks progress and challenges in implementation," *Journal of Laboratory Automation*, **18**(4):264–268. DOI 0.1177/2211068213484471

[30] Perkel, J. M. (6 May 2021) "Reactive, reproducible, collaborative: Computational notebooks evolve," *Nature*, **593**:156–157. DOI 10.1038/d41586-021-01174-w

[31] Akhlaghi, M., Infante-Sainz, R., Roukema, B. F., Khellat, M., Valls-Gabaud, D. and Baena-Gallé, R. (May 2021) "Toward long-term and archivable reproducibility," *Computing in Science & Engineering*, **23**(3):82–91. URL https://maneage.org, DOI 10.1109/mcse.2021.3072860

[32] Knuth, D. E. (1984) "Literate programming," *The Computer Journal*, **27**(2):97–111. DOI 10.1093/comjnl/27.2.97

[33] ——— (1992) *Literate programming*, vol. **27** of CSLI Lecture Notes, 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," *Science of Computer Programming*, **156**:45–67. URL GitHub.com/haroldthimbleby/relit, DOI 10.1016/j.scico.2017.12.010

[35] Gray, T. W. and Wolfram, S. (26 March 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," *Computing in Science & Engineering*, **23**(2):7–14. DOI 10.1109/MCSE.2021.3059263

[37] Xie, Y. (2015) *Dynamic Documents with R and knitr*, CRC, second ed.

[38] Thimbleby, H. (1999) "Specification-led design for interface simulation, collecting use-data, interactive help, writing manuals, analysis, comparing alternative designs, etc," *Personal Technologies*, **4**(2):241–254. URL harold.thimbleby.net/ansim, DOI 10.1007/BF01885563

[39] Office for National Statistics (2022) *Using reproducible analytical pipelines (RAP) to improve statistics*. URL https://code.statisticsauthority.gov.uk/case-studies/using-reproducible-analytical-pipelines-rap-to-improve-statistics/

[40] Upson, M. (2017) *Reproducible analytical pipelines*. URL 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. URL www.gov.uk/government/publications/better-broader-safer-using-health-data-for-research-and-analysis

[42] Ainsworth, R. *et al.* (July 2022) *The Turing Way: A Handbook for Reproducible Data Science*, vol. **v1.0.3**, Zenodo. URL the-turing-way.netlify.app/welcome, DOI 10.5281/zenodo.3233853

[43] Courtès, L. (2020) "[re] storage tradeoffs in a collaborative backup service for mobile devices," *Rescience C*, **6**(1:#6):10. URL https://gitlab.inria.fr/lcourtes-phd/edcc-2006-redone, DOI 10.5281/zenodo.3886739

[44] Glen, S. (2022) *Reporting Statistics APA Style*, Statistics How To. URL 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," *Pharmacological Reports*, **72**(3):481–485. DOI 10.1007/s43440-020-00110-5

[46] (2022) *Science journals: editorial policies*. URL 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," *Nature Digital Medicine*, **3**(85). DOI 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," *Proceedings National Academy of Sciences*, **110**(48):19313–19317. DOI 10.1073/pnas.1313476110

[51] Shneiderman, B. (2016) "Opinion: The dangers of faulty, biased, or malicious algorithms requires independent oversight," *Proceedings National Academy of Sciences*, **113**(48):13538–13540. DOI 10.1073/pnas.1618211113

[52] Friedman, B. and Nissenbaum, H. (1996) "Bias in computer systems," *ACM Transactions on Information Systems*, **14**(3):330–347. DOI 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 nephrology*, **15**:132. DOI 10.1186/1471-2369-15-132

[54] Whitty, C. J. M. (2015) "What makes an academic paper useful for health policy?" *BMC Medicine*, **13**:301. DOI 10.1186/s12916-015-0544-8

[55] Hawkins, D. M. (2004) "The problem of overfitting," *Journal of Chemical Information and Modeling*, **44**:1–12. DOI 10.1021/ci0342472

[56] May, R. M. (1976) "Simple mathematical models with very complicated dynamics," *Nature*, **261**:459–467. DOI 10.1038/261459a0

[57] Dyson, F. (2004) "A meeting with Enrico Fermi," *Nature*, **427**:297. DOI 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 Informatics*, **27**(e100165):1–5. DOI 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 [129]), Leipzig. URL 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*, Open Educational Resources for the Ancient Near East, University of Central Florida. URL 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**(1257). DOI 10.12688/f1000research.26932.2

[63] Page, M. J. *et al.* (2021) "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews," *Systematic Reviews*, **10**(89):1–11. DOI 10.1186/s13643-021-01626-4

[64] Thimbleby, H. (2016) "Human factors and missed solutions to Enigma design weaknesses," *Cryptologia*, **40**(2):177–202. DOI 10.1080/01611194.2015.1028680

[65] Sayburn, A. (2020) "Covid-19: Experts question analysis suggesting half UK population has been infected," *BMJ*, **368**:m1216. DOI 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**(m1328). DOI 10.1136/bmj.m1328

[67] Knuth, D. E. (1998) *The Art of Computer Programming (Seminumerical Algorithms)*, vol. **2**, Addison-Wesley, 3rd ed.

[68] Jackson, D. (2021) *The Essence of Software*, Princeton University Press.

[69] Popper, K. R. (2002) *Conjectures and Refutations: The Growth of Scientific Knowledge*, Routledge, 2nd ed.

[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**(6227):265–270. DOI 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," *Annals of Internal Medicine*, **162**(1):W1–73. DOI 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. DOI 10.1038/d41586-020-01003-6

[74] Ferguson, N. (22 March 2020) *Tweet*. URL twitter.com/neil_ferguson/status/1241835454707699713

[75] Leake, J. (29 March 2020) "Neil Ferguson interview: No 10's infection guru recruits game developers to build coronavirus pandemic model," *The Sunday Times*. URL 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," *Computing in Science & Engineering*, **15**(4):60–63. DOI 10.1109/MCSE.2013.91

[77] Smith, B. (6 May 2020) "SAGE adviser Neil Ferguson quits over coronavirus lockdown breach," *Civil Service World*. URL 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," *Mathematics of Computation*, **27**:927–937. DOI 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," *Communications of the ACM*, **15**(10):873–882. DOI 10.1145/355604.361593

[80] The MISRA Consortium Limited (2020) *MISRA Compliance: 2020 – Achieving compliance with MISRA Coding Guidelines*, MISRA Consortium. URL 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. URL book.realworldhaskell.org

[83] Chawla, D. S. (8 June 2020) "Critiqued coronavirus simulation gets thumbs up from code-checking efforts," *Nature*, **582**:323–324. URL www.nature.com/articles/d41586-020-01685-y

[84] Scheuber, A. and van Elsland, S. L. (1 June 2020) *Codecheck confirms reproducibility of COVID-19 model results*. URL 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*. URL github.com/codecheckers/covid-report9, DOI 10.5281/zenodo.3865491

[86] Hatton, L. and Roberts, A. (1994) "How accurate is scientific software?" *IEEE Transactions on Software Engineering*, **20**(10):785–797. DOI 10.1109/32.328993

[87] Halloran, M. E. *et al.* (2008) "Modeling targeted layered containment of an influenza pandemic in the United States," *Proceedings of the National Academy of Sciences*, **105**(12):4639–4644. URL www.pnas.org/content/105/12/4639, DOI 10.1073/pnas.0706849105

[88] Schnell, S. (2015) "Ten simple rules for a computational biologist's laboratory notebook," *PLoS Computational Biology*, **11**(9):e1004385. DOI 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. URL grants.nih.gov/grants/guide/notice-files/NOT-OD-21-013.html

[90] Kozlov, M. (16 February 2022) "NIH issues a seismic mandate: Share data publicly," *Nature*. URL www.nature.com/articles/d41586-022-00402-1

[91] Ferguson, N. (22 March 2020) *Tweet*. URL twitter.com/neil_ferguson/status/1241835456947519492

[92] Richards, D. and Boudnik, K. (16 May 2020) "Neil Ferguson's Imperial model could be the most devastating software mistake of all time," *The Telegraph*. URL 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," *Nature Scientific Reports*, **9**(2216). DOI 10.1038/s41598-019-38665-w

[94] Thimbleby, H. (2003) "The Directed Chinese Postman problem," *Software — Practice & Experience*, **33**(11):1081–1096. URL harold.thimbleby.net/cpp/index.html, DOI 10.1002/spe.540

[95] Sander, B., Nizam, A., Garrison Jr, L. P., Postma, M. J., Halloran, M. E. and Longini Jr, I. M. (2009) "Economic evaluation of influenza pandemic mitigation strategies in the US using a stochastic microsimulation transmission model," *Value Health*, **12**(2):226–233. DOI 10.1111/j.1524-4733.2008.00437.x

[96] Perkel, J. M. (2022) "How to fix your scientific coding errors," *Nature*, **602**:172–173. DOI 10.1038/d41586-022-00217-0

[97] Guest, O. and Martin, A. E. (2021) "How computational modeling can force theory building in psychological science," *Perspectives on Psychological Science*, **16**(4):789–802. DOI 10.1177/1745691620970585

[98] ACT-R Research Group (2022) *ACT-R*. URL 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 Evidence and Electronic Signature Law Review*, **17**:1–14. DOI 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**(6493):860–868. DOI 10.1126/science.abb5793

[102] Verity *et al.* (2020) "Estimates of the severity of coronavirus disease 2019: A model-based analysis," *Lancet*, **20**(6):669–677. DOI 10.1016/S1473-3099(20)30243-7

[103] Hoare, C. A. R. (1969) "An axiomatic basis for computer programming," *Communications of the ACM*, **12**(10):576–580. DOI 10.1145/363235.363259

[104] Freedman, L. P., Cockburn, I. M. and Simcoe, T. S. (2015) "The economics of reproducibility in preclinical research," *PLOS Biology*, **13**(6):e1002165. DOI 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**(160018):1–9. DOI 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 Computational Biology*, **18**(9):e1010476. DOI 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 Computing Surveys*, **41**(4):1–36. DOI 10.1145/1592434.1592436

[108] ACM (2020) *Code of Ethics and Professional Conduct*, ACM. Accessed 23 April 2020, URL www.acm.org/code-of-ethics

[109] ——— (2020) *Artifact Review and Badging — Current*, vol. **Version 1.1**, ACM. URL 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*. Revised, URL 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.* URL webstore.iec.ch/publication/22273

[112] Hoare, C. A. R. (1996) "How did software get so reliable without proof?" *in Lecture Notes in Computer Science*, vol. **1051**, Springer, pp. 1–17. DOI 10.1007/3-540-60973-3_77

[113] Fomel, S. (2015) "Reproducible research as a community effort: Lessons from the Madagascar Project," *Computing in Science & Engineering*, **17**(1):20–26. DOI 10.1109/MCSE.2014.94

[114] Stol, K.-J. and Fitzgerald, B. (2018) "The ABC of software engineering research," *ACM Transactions on Software Engineering and Methodology*, **27**(3):1–51. DOI 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," *Procedia Computer Science*, **4**:577–578. DOI 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," *Procedia Computer Science*, **9**:439–448. International Conference on Computational Science, ICCS 2012, DOI 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)," *Computational Linguistics*, **22**(2):265–276.

[121] Falessi, D. and Kruchten, P. (2015) "Five reasons for including technical debt in the software engineering curriculum," *in Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW'15, ACM, pp. 28:1–28:4. DOI 10.1145/2797433.2797462

[122] Heineman, Jr., B. W. (2013) "The JP Morgan "Whale" report and the ghosts of the financial crisis," *Harvard Business Review*. URL 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**(m4181). DOI 10.1136/bmj.m4181

[125] Wadman, M. (2021) "Could too much time between doses drive the coronavirus to outwit vaccines?" *Science*. DOI 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," *Frontiers in Neuroinformatics*, **11**(69). DOI 10.3389/fninf.2017.00069

[127] House of Commons Science and Technology Committee (16 December 2020) *The UK response to covid-19: Use of scientific advice*. URL committees.parliament.uk/publications/4165/documents/41300/default

[128] Thimbleby, H. (29 April 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]). URL 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," *Computing in Science & Engineering*, **11**(2):7–11. DOI 10.1109/MCSE.2009.30

Supplemental Material

Improving science that uses code

Harold Thimbleby, harold@thimbleby.net

June 6, 2023

Section, figure, table, citation, and pagination numbering continue from the main paper

Contents

8 Further issues for Software Engineering Boards (SEBs)

Software Engineering Boards, henceforth SEBs, will be used to help and assure that critical code, including epidemic modeling, is of high standard, to provide assurance for scientific papers, Government public health and other policies, etc, that the code used is of appropriate quality for its intended uses.

Further details of the SEB proposal is in the main paper. Here we raise further issues for SEBs (additional to those covered in the main paper's introduction to SEBs), potential limitations and possible responses that can be addressed over time:

1. Until there are national qualifications, nobody — certainly nobody without professional training in software — really knows just how bad (or good) they are at Software Engineering.

2. When code is taken seriously, concerns may be raised on programmers' contributions to research, intellectual property rights, and co-authoring [130]. Software engineering is a hard, creative discipline, and getting epidemiological (and other scientific) models to work is generally a significant challenge, on a par with the setting up and exploring the mathematical models themselves. Often Software Engineers will need to explore boundary cases of models, and this typically involves hard technical mathematics [100]. Often the Software Engineers will be solving entirely new problems and contributing to the research. How this is handled needs exploring. How Software Engineers are appropriately credited and cited for their contributions also needs exploring.

3. SEBs require policies on professional issues such as membership, transparency, and accountability.

4. There should be a clear separation between the SEB members' activities as part of the Board, and their other activities, including professional advice, code development, or training (which is likely to be in demand from the same people who require formal approvals from the SEBs).

5. Professional Engineering Bodies have a central role to play in professionalism, ranging from education and accreditation to providing professional structures and policies for SEBs. For example, should and if so how should the programming skills taught to computational scientists (epidemiologists, computational biologists, economists, computational chemists, ...) be accredited?

6. In the main paper, SEBs are viewed as a constructive contribution to good science, specifically helping improve the quality of epidemiological modeling. More generally, SEBs will have wider roles, for instance in overseeing software subject to medical device regulation [60].

7. SEBs may fruitfully collaborate with other engineering disciplines to share and develop best practice. For example, engineers in other domains (e.g., civil engineers) routinely sign off projects, yet, on the other hand, they often overlook the quality of Software Engineering their projects implicitly rely on — for the same reasons as the scientific work discussed in this paper overlooks the dependence on quality software.

8. Clearly, at least while this paper's concepts are tested and mature, SEBs will need to collaborate closely with research organizations, journals, and funding agencies in order to develop incremental developments to policies and processes that will be most effective, and which can be introduced most productively over time to the scientific community at large. Funding agencies may wish to support such strategic work, as they have previously funded one-off projects such as [131].

There are other ideas to help make SEBs work, but it is clear they are part of the solution. We must not let perfection be the enemy of the good. SEBs don't need to be perfect on day one, but they do need to get going in some shape or form to start making their vital contribution.

8.a Relationships of SEBs to Ethics Boards

1. Although SEBs may start with a checklist approach, like Ethics Boards generally do, it cannot be assumed that people approaching SEBs know enough about Software Engineering to perform adequate software assessments when there is any risk (as there is in public policy, medical apps, and so on). SEBs may also provide mentoring and training.

2. Unlike Ethics Boards, which provide hands-off oversight, SEBs should provide professional advice, perhaps providing training or actually helping hands-on develop appropriately reliable software. During a pandemic SEBs would be very willing to do this, but in the long run it is not sustainable as voluntary labour, so all research, particularly medical research, should include support for professional Software Engineering.

3. Ethics Boards typically require researchers to fill in forms and provide details, which is a feasible approach as researchers know if they are doing experiments on children, for instance, so the forms are relatively easy to fill in (if often quite tedious). On the other hand, few healthcare and medical researchers understand software and programming, so they are *not* able to fill in useful software forms on their own. SEBs need to know how well engineered the software really is, not how good its developers *think* it is. As typical programs are enormous, SEBs are either going to need resources to evaluate programs, or they will need to supervise independent bodies that can do it for them.

4. SEBs should have a two-way collaboration with Ethics Boards.

   - SEBs have to deal with ethical concerns, and how they may be implemented in code. One of the papers [192] in the survey (discussed later in this Supplemental Material) is a case in point, as is the growing cross-fertilization between AI and ethics [e.g., 132].
   - Ethics Boards also have to deal with software, and it is clear that they often fail to do this effectively. The case of the retraction of a peer reviewed articles for *The Lancet* [133, 134, 135] and the *Journal of Vascular Surgery* [136, 137, 138], discussed in section 12.a.2, are cases in point.

5. Like some Ethics Boards, SEBs might become, or be perceived as becoming, onerous and heavy handed — as if the Board is not interested in ethics but only in following a bureaucratic pathway. It seems essential, then, that SEBs have (and perhaps are chaired by) experienced, practicing, professional Software Engineers to avoid this problem.

8.b SEBs are necessary but not sufficient

The main paper provides evidence and argues that SEBs (or equivalent) are necessary to help improve the quality of science, specifically science relying, explicitly or implicitly, on tools or methods based in software.

SEBs address the problems identified at the laboratory end of doing science; they do not address the processes of review, editorial control, and action based on claimed results. As shown in the review of 32 papers, only some journals have code policies, and the policies are not enforced. In other words, improving the professionalization of Software Engineering has to proceed from doing science, which the paper covers, to the downstream issues of review and publication. SEBs may work with journals, funding agencies and even international standards agencies to improve broader awareness of professional Software Engineering, but this is a topic the present paper has not addressed. It needs doing.

9 Software engineering best practice

This Supplemental Material provides more explanations and justification for following standard Software Engineering practices that support reliable modeling, reliable research, and, most generally, reliable science.

The reader is referred to standard textbooks for more information [e.g., 5,6], as well as to specialized texts that are more specifically addressed to Software Engineering in science [e.g., 131]. Written and maintained by a team of experts, a substantial and wide-ranging reference is the Software Engineering Body of Knowledge (SWEBOK) [139], recognized as International Standards Organization Technical Report 19759.

The Turing Institute has an excellent open resource [42], though it emphasizes RAP for handling data and authoring papers rather than for programming reliably.

The book *Why Programs Fail* [140] is a very good practical guide to developing better code, and will be found very accessible. Humphrey [141] outlines a thorough discipline for anyone wanting to become a good programmer. Improvement is such an important activity, Humphrey has also published a book to persuade managers of the benefits [142]. Further suggestions for background reading can be found throughout this section.

Software Engineering includes the following topics:

9.a Requirements

It is not always necessary to program well if the code to be produced is for fun, experimenting, or for demonstrations. On the other hand, if code is intended for life-critical applications, then it is worth putting more engineering effort into it. The first step of Software Engineering, then, is to assess the requirements, specifically the reliability requirements of the code that is going to be produced.

In practice, requirements and expectations change. Early experimental code, developed informally, may well be built on later to support models intended to inform public policy, for instance. Unfortunately, prototypes may impress project leaders who then want to rush into production software because, it seems, “it

obviously works." Fortunately, best practice Software Engineering can be adopted at any stage, particularly by using *reverse engineering*. In reverse engineering, one carefully works out (generally partly automatically) what has already been implemented. This specification, carefully reviewed, is then used as the basis for a more rigorous Software Engineering process that implements a more reliable version of the system.

9.b Formal methods

In the physical world, to do something as simple as design and build a barbecue, you would need to use elementary mathematics to calculate how many bricks to buy. To build something more substantial, such as block of flats, you would need to use structural engineering (with certified structural engineers) to ensure the building was safe. Although programming lends itself to mathematical analysis, it is surprising that few programmers use explicit mathematics at all in the design and implementation of software.

The type and use of mathematics used in Software Engineering is called **formal methods**. Not using formal methods ensures the resulting code is unsafe and unreliable. Of particular relevance to scientific modeling: there must be an explicit use of formal methods to ensure mathematical models (such as differential equations) are correctly implemented in code (and to understand the any limitations of doing so).

It is important to be clear that formal methods is a spectrum, from doing it as rigorously and comprehensively as possible using state of the art methods, to applying a basic "formal methods mindset" that any competent programmer could do.

- An interesting paper explores a formal approach to coding differential equations [143], but it makes it clear that their approach is beyond most programmers. Such formal methods require sophisticated knowledge of logic [107], as well as practical knowledge of using appropriate formal methods tools (Alloy, HOL, PVS, SPARK [81], and *many* others). Using the right tools is essential for reliable programming, because the tools do quickly and reliably what, done by hand, would be slow and error-prone. Standard tools cover verification, static analysis of code, version control, documentation, and so on — this paper explains why some of these activities are essential for reliable programming below.

- At the other end of the spectrum, a formal methods mindset means being clear about basic mathematical properties of code, such as by using assertions, proving invariants in loops, and so on. This approach should be feasible — and perhaps required — for all scientific programmers.

Crucially, computer tools are available to catch common human errors that we are all prone to. Many tools are designed to avoid common human errors arising *in the first place*; notably, the MISRA C toolset simply stops the developer using the most error-prone features of normal C, and hence improves the quality of programming with little effort [80].

Many programming languages and programming environments have integrated features that support formal methods. For example, Hoare's triples [103] (and formal thinking based on similar ideas) are readily supported by assertions, as either provided explicitly in a programming language or through a simple API. In particular, assertions readily support contracts, an important rigorous way of programming: assertions allow the program, the programming language, or tools (as the case may be) to automatically (and hence rigorously) check essential details of the program.

Hoare's original 1969 paper [103] is very strongly recommended because it is a classic paper that has stood the test of time; in the 1960s it was leading research, but now it can be read as an excellent introduction, given how the field of Software Engineering has advanced and become more specialized and sophisticated over the decades since. Hoare is also a very good writer.

Formal methods have the huge advantage that they "think differently" and therefore help uncover design problems and bugs that can be found in no other way. Because formal methods are logical, mathematical theories (safety properties, and so forth) can be expressed and checked (often automatically); this provides a very high degree of insight into a program's details, and hence supports fault tolerance (e.g., redundancy). Ultimately, formal methods provides good reasons to believe the quality of the final code — that it does what it is supposed to do. Unfortunately, because formal methods are mathematical, few programmers have experience of using them. Fortunately tools are widely available to help use formal methods very effectively.

9.c Defensive programming

Defensive programming is based on a range of methods, including error checking, independent calculation (using multiple implementations written by independent programmers), assertions, regression testing, etc. Notoriously, what are often unconsciously dismissed as trivial concerns frequently lead to the hardest to diagnose errors, such as buggy handling of "well-known, trivial" things like numbers [144]. The great advantage

of defensive programming is that it detects, and may be able to recover from, bugs that have been missed earlier in the development process (such as typos in the code). Defensive programming requires professional training to be used effectively, for example it is not widely known that some choices of programming language make defensive programming unnecessarily hard [145].

A special case of defensive programming appropriate for pandemic modeling is mixing methods. Do not rely on one programming method, but mix methods (e.g., different numerical methods) to use and compare multiple approaches to the modeling.

Interestingly, the only paper reviewed that claimed to do any independent testing [189] failed to include any testing in its data or code repository, so the testing itself — the essential quality assurance of the code — is not open to scrutiny (e.g., the code and the "independent" code are likely to contain common code, data, and common bugs).

9.d Using inappropriate programming languages undermines reliability

Many popular languages are popular because they are easy to use, which is not the same as being reliable to use. The fewer constraints a language imposes, the easier it *seems* to be to program in, but the lack of constraints means the language cannot provide the checks stricter languages do. C, for instance, which is one of the languages widely used for modeling [146, 74], is not a good choice for a reliable programming language — it has many intrinsic weaknesses that are well-known to professionals, but which frequently trap inexperienced programmers. (This is not the place for a review of bad programming languages, for which see [145], but Excel is even worse than C.)

In particular, C is not a portable language, which means C code will work differently on different types of computer and operating system.

SPARK Ada is a popular example of a much more appropriate high integrity programming language to use [81]. SPARK Ada also has the advantage that most Ada programmers are better qualified than most C programmers.

Other high integrity languages include OCaml, F*, and Haskell; reference [82] is an excellent introduction to Haskell, and introduces the wider issues of reliable programming in such languages.

9.e Open source and version control

It is appreciated that the models may change and be adapted as new data and insights become available. Changing models makes it even harder to ensure that they are correct, and thus emphasizes the relevance of the core message this paper: we have to find ways to make computer models more reliable, inspectable, and verifiable. Version control keeps a record of what code was used when, and enables reconstruction of earlier versions of code that has been used. Version control is supported by many tools (such as Git, Subversion, etc).

If version control is not used, one has no idea what the current program actually is. Version control is essential for *reproducibility* [76, 126] (see also section 5.b): it enables efforts to duplicate work to start with the exact version that was used in any published paper, provided that the published paper discloses the version and a URL for the relevant repository. Note that version control should also be used for data and web site data used by code, otherwise the results reported are not replicable.

If results cannot be reproduced, has anything reliable been contributed? When a modeling paper presents results from a model, it is important to reproduce those results without using the same code. Better still, research should be reproduced without sharing libraries or APIs (for example, results from a model using R might be reproduced using *Mathematica* — this is a case of N (where, in this case, $N = 2$) version Programming [86]). Reproducing the same results relying on the same codebase tells you little. The more independent reproductions of results the greater the evidence for belief in the implications.

Clearly, with the transformations a program from avian flu in Thailand [10] to COVID-19 in the United States and in Great Britain [9] taking place over many years, version control would have been very helpful to keep proper track of the changes. Note that professional version control repositories also provide secure off-site back up, ensuring the long-term access to the code and documentation — this would avoid loss of Supplemental Material problems, as occurred in [95].

Most version control systems would, in addition, enable open source methods so the code could be shared — and reviewed — by a wider community. Open source is not a panacea, however; it raises many trade-offs. Particularly for world-wide concerns like pandemic modeling, it increases diversity in the software developers, and fosters a diverse scientific collaboration. Open source can raise people's standards — some countries [147, 148] are using Excel models to manage COVID-19, and as there are.are serious dependability problems with Excel (illustrated particularly in section 6.g of the main paper), open source projects competently implemented (e.g., avoiding or carefully managing the use of Excel) would help these people enormously.

Open source raises important licensing and management questions to ensure the quality of contributions. A salutary open source case is NPM, where lawyers from a company called Kik triggered Azer Koçulu, that is, a *single* programmer, to remove all his code from a repository. This caused problems to many thousands of JavaScript programmers worldwide who could no longer compile anything — ironically, including Kik itself [149].

Critically in the case of epidemic modeling, open source democratizes the model development and interpretation, and enables properly-informed public debate. Note that many (if not most) successful open source projects have had a closed team of highly dedicated and directly employed developers [113].

9.f Rigorous testing

In poorly-run software development it is very easy to miss bugs, because the flawed thinking that inserted bugs in the code is going to be the same flawed thinking with the same misconceptions that tries to detect them. Rigorous testing includes methods like fault injection. Here, the idea is that if testing finds no bugs, that may be because the testing is not rigorous enough rather than that the program actually has no bugs. Fault injection inserts random bugs, and then testing gives statistical insights into the number of bugs in a program (depending on how many deliberate bugs it successfully finds).

It is very tempting to test code while it is being built, save some or all of the code on a repository, but forget to check that the code has not changed out of recognition of the earlier tests — tests should be saved so that modified code can easily be tested again. For example, if a test reveals a bug, the bug should be fixed *and* the test needs to be re-run to check the fix worked (and did not introduce other bugs previously eliminated).

It is important that code is saved and then downloaded to a clean site, confirmed it is consistent, and a new build made (preferably by an independent tester), which is then re-tested. If this procedure (or equivalent) is not followed, there is no assurance that the code made available with the paper is complete and works reliably.

There are many other important testing methods [5, 6, 86].

9.g Good documentation and record keeping

Documentation covers internal documentation (how code works), developer (how to include it in other programs), configuration (how to configure and compile the code in different environments), external documentation (how the code is used), and help (documentation available while using the program).

For critical projects, such as for pandemic modeling, all documentation (including software) should be formally controlled, typically digitally signed and backed up in secure repositories. One would also expect a structured assurance case to be made, both to help the authors understand and complete their own reasoning and to help reviewers scrutinize it [58].

For purely scientific purposes, perhaps the most important form of documentation is internal documentation: how to understand how and why the code works. This is different from developer documentation, which is how to *use* the code in other programs. For example, code for solving a differential equation needs explaining — what method does it use, what assumptions does it have? In contrast, the developer documentation for differentiation would say things like it solves ordinary differential equations with parameters e for the function f with the independent variable x in the interval $[u, v]$, or whatever, but *how* it solves equations is of little interest to the developer who just needs to use it. How code works — internal documentation — is essential for the epidemiologist, or more generally any scientist. An example of a simple SIR epidemiological model's internal documentation can be found at URL `http://www.harold.thimbleby.net/sir`

There are many tools to help manage documentation (Javadoc, Doxygen, ...). Literate programming is one very effective way of documenting code, and has been used for very large programming projects [32]. Literate programming has also been used directly to help publish clearer and more rigorous papers based on code [34] — a paper that also includes a wider review of the issues.

Documentation should be supplemented by details of algorithms and proofs of correctness (or references to appropriate literature). All the documentation needs to be available to enable others to correctly download, install and correctly use a program — and to enable them, should they wish, to repurpose it reliably for their own work. In addition, documentation requires specifications and, in turn, *their* documentation.

A important role of documentation is to cover configuration: how to get code to work — without configuration, code is generally useless. The most basic is a `README` file, which explains how to get going; more useful approaches to configuration include make files, which are programs that do the configuration automatically.

Without proper record keeping, code becomes almost impossible to maintain if programmers leave the project. Note that computer tools can make record keeping, laboratory books etc, trivial — if they are used.

9.h Usability

Usability is an important consideration: [150, 151] is the program usable by its intended users so they can obtain correct results? Often the programmers developing code know it so well they misjudge how easy it will be for anyone else to use it — this is a very serious problem for the lone programmer (possibly working in another country) supporting a research team. Usability is especially important when programs are to be used by other researchers and by non-programmers, including epidemiologists.

In publishing science, an important class of user includes the scientists and others who will use or replicate the work described. When code used in research is non-trivial, it is essential that the process of successfully downloading code and configuring it to run is made as usable as possible. Typically so-called makefiles are provided, which are shell scripts or apps that run on the target machine, establish its hardware and other features, then automatically configure and compile the code to work on that machine. Makefiles typically also provide demo and test runs and other helpful features. Other approaches to improve usability are zip files, so every relevant file can be conveniently downloaded in one step, and using standard repositories, such as GitHub which allow new forks to be made, and so on.

9.i Reusing quality solutions

Reusing quality code (mathematical functions, database operations, user interface features, connectivity, etc) avoids having to develop it oneself, saves time and avoids the risks of introducing new bugs. The more code that is reused, the more likely many people will have contributed to improving it — for example, reusing a standard database package will provide Atomicity, Consistency, Isolation, and Durability (so-called ACID properties) without any further work (nor even needing to understand what useful guarantees these basic properties ensure).

Note that reusing code assumes the originators of the code followed good Software Engineering practice — particularly including good documentation; equally, if the code being developed building on it follows good Software Engineering practice, it too can be shared and further improved as it gets more exposure. Its quality improves through having scrutiny by the wider community, and in successful cases, leading to consensus on the best methods. Indeed, reuse, scrutiny, and consensus are the foundations of good science.

Anticipating reuse during program development is called *flexibility*, where various programming techniques can greatly enhance the ease and reliability of reuse [152].

A special case of reuse is to use software tools to help with software development. The tools (if appropriately chosen) have been carefully developed and widely tested. Tools enable software developers to avoid or solve complex programming problems (including maintenance) repeatedly and with ease.

9.j Simplicity

When a program doesn't quite do what is wanted, it is tempting to add more features or variables, or to treat the problem as an "exception" and program around it — which inserts more code and, almost certainly, more bugs. This way lies over-fitting, a problem familiar from statistics (and machine learning). Programs can be made over-complex and they can then do anything; an over-complex program may seem correct by accident. Instead, the hallmarks of good science are that of parsimony and simplicity; if a simple program can do what is needed it is more likely to be correct. A simpler program is easier to prove correct, easier to program, and easier to debug. A special case of needing simplicity is when fixing bugs: instead of fixing bugs one at a time, one should be fixing the *reasons* why the bugs have happened. Generally, when bugs are fixed, programmers should determine *why* the bugs occurred, and thence repair the program more strategically.

9.k Compliance with standards

To ensure adherence to best practice and, importantly, to avoid being unaware of relevant methodologies, professional software development projects adopt and adhere to relevant standards, such as ISO/IEC/IEEE 90003:2018 [153]. However, for safety-critical models or models of national policy significance, much stronger standards such as aviation software standards, such as RTCA DO-178C/EUROCAE ED-12C [154], commonly called DO-178C, will be more appropriate. Publications should then cite the standards to which their computer models comply.

Note that medical device regulation, which has its own standards, is lagging behind professional Software Engineering practice, and currently provides no useful guidance for critical software development [60].

9.l Effective multidisciplinary teamwork

As this long list illustrates, Software Engineering is a complex and wide-ranging subject. Software engineering cannot be done effectively by individuals working alone (for instance, code review is impossible for individuals to perform effectively), even without considering the complexities of the domain the code is intended for (in the present case, including pandemic modeling, mathematical modeling, public health policy, etc). Multidisciplinary teamwork is essential.

Modern software is complex, and no one person can have the skills to understand all relevant aspects of all but the most trivial of programs. Furthermore, programming is a cognitively demanding task, and causes loss of situational awareness (that is, cognitive "overload" making one unable to track requirements beyond those thought to be directly related to the specific task in hand). The main solution to both problems is teamwork, to bring fresh insights, different mindsets and skills to the task.

Peer review of code is an essential teamwork practice in reliable program development: [155, 6] it is easy to make programming mistakes that one is unaware of, and an independent peer review process is required to help identify such unnoticed errors.

Almost all software will be used by other people, and user interface design is the field concerned with developing usable and effective software. A fundamental component of user interface design is working with users and user testing: without engaging users, developers are very likely to introduce quirks that make systems less usable (often less safe) than they should be. In short, users have to be brought into the software team too.

9.m Continuous Professional Development (CPD)

As computing technology continues to develop rapidly — especially as new programming tools and systems are introduced — best practice in Software Engineering is also rapidly evolving. Continuous Professional Development (CPD) is essential.

Ironically, the more organized CPD the more likely the content itself will lag behind. There is an argument for two-way links between universities (and other research organizations), research science developers, including enabling developers to undertake part-time research degrees. Research degrees teach not just current best-practice but also how to stay abreast of the relevant technologies and literature as it develops.

The UK's Software Sustainability Institute is one initiative that is making important contributions [156, 157], and its web site will no doubt remain timely and up to date in a way that this paper cannot.

Note that CPD is not just a matter of learning current best practice, but a continual process as best practice itself continually evolves. In Software Engineering, a current (as of 2021) initiative concerns reproducible code artifacts and badging papers to clearly show the approaches they take [109], and this will in due course have a direct impact on Software Engineering standards in other fields.

9.n Security and other factors

Of course, there are many other factors to be considered for the professional development of critical code, such as using appropriate methods to ensure cybersecurity [158, 159], particularly while also being able to up- and download secure updates.

For pandemic modeling specifically, understanding the limitations of numerical methods (in particular, how numerical methods are affected by the choice of programming language and style of programming) is critical.[14] Hamming [100] is considered a classic, but there is a huge choice available.

For reasons of space, the present paper does not discuss the issues raised by AI, nor the many very important, non-trivial social and professional concerns, which have complex implications for Software Engineering practice, such as managing programming teams, data ethics, privacy, legal liability [160], or software as a matter in law, as in disputes over model results or disputes over ownership of code [161].

9.o Software is a human activity

Software is a human activity, and humans are fallible. Even the Software Engineering methodologies to developer better software are themselves human constructs, and are therefore subject to the same fallibilities.

People would generally not make software errors if there were aware they were making errors. Unfortunately programming is a very demanding activity, which causes tunnel vision (also known as loss of

[14] For example, code from one of the surveyed paper [198] uses literal numbers at far too high a precision for the chosen language to be able to represent correctly (conformant implementations use IEEE 754 double precision 64-bit floating point). Such an error typically has an undefined impact on results, and unfortunately is easy to overlook as the program almost certainly ignores the error when running. The error belies misunderstandings in programming which may have wider effects, such as consequences of relying on the precision being higher than it is.

situational awareness). Humans have limited cognitive capacity, and programming (especially programming in a competitive environment, like science) drives programmers to use as much of their cognitive skills for the task in hand. The consequence is programmers focus on "the" problem as it appears in the code, and inevitably become unaware they are not considering wider issues. The correctness, generality, ethics, and usability of a program are therefore often unintentionally sacrificed to making code work at all.

Confirmation bias is a standard Human Factors problem [60], which encourages us to perform tests that show our programs work. Instead, we should be rigorously testing ways in which programs can fail as well. This is exactly the same issue pointed out by Popper [69]: scientists should experiment to find reasons why hypotheses are false, and indeed use simple hypotheses that are testable. Software is really no more than a collection of sophisticated hypotheses, and Computer Science is a science of the artificial [162].

Standard Human Factors mitigations for such problems include team working, with appropriate precautions to manage authority gradients (where the Human Factors oversights of the leader influence the team). Many computerized mitigations are also available — strong typing, code analyzers, formal methods, and so on, as described in this section of the Supplemental Material.

Following the Dunning-Kruger Effect [163, 164], programmers over-estimate their programming skills because they do not have the skills to recognize their lack of knowledge — in the present case, knowledge of basic Software Engineering.

Dunning and Kruger go on to say,

> "People usually choose what they think is the most reasonable and optimal option [...] The failure to recognize that one has performed poorly will instead leave one to assume that one has performed well; as a result, the incompetent will tend to grossly overestimate their skills and abilities. [...] Not only do these people reach erroneous conclusions and make unfortunate choices, but their incompetence robs them of the metacognitive ability to realize it."

Unlike many skills (skating, brain surgery, ...) programming, typical of much engineering, is one where errors can go unnoticed for long periods of time — things seem to work nicely right up to the moment they fail. The worse programmers are, the more trivial bugs they tend to make, but trivial bugs are easy to find so, ironically, being a poor programmer *increases* one's self-assessment because debugging seems very productive. It is easy for poor programmers and their associates to believe they are better than they actually are, fertile ground for the better-than-average bias [163].

It sounds harsh to call programmers incompetent, but challenged with the complexity of programs and the complexity of the domains programs are applied in, we are all incompetent and succumb to the limitations of our cognitive resources, suffering blindspots in our thinking [60]. We *all* make mistakes we are unaware of. If we do not have the benefit of professional qualifications that have assessed us objectively, we generally have a higher opinion of our own competence than is justified. Moreover, if we do not work in a diverse team, nobody will ever point this out, so the potential problems it causes will never be addressed.

Everyone is subject to Human Factors (including the author of the present paper, e.g., as discussed in [165]): for instance, the standard cognitive bias of confirmation bias encourages us to look for bugs when code fails to do what is expected and then debug it to produce better results, but if code generates expected results not to bother to debug it further. This of course tends to make code increasingly conform to prior expectations, whether or not those expectations are scientifically justified. Typically, there was no prior specification of the code, so the code should be right, especially after all the debugging to make it "correct"! Thus coding routinely suffers from HARKing (Hypothesizing After the Results are Known [166]), a methodological trap widely recognized in statistics.

Computers themselves are also a part of the problem. Naïvely modifying a program (as may occur during debugging) typically makes it more complex, more *ad hoc*, and less scrutable. Programs can be written so that it is not possible to determine what they do or how they do it (whether by deliberate obfuscation, as in malware, or accidentally), except by running them, if indeed it is possible to exactly reproduce the necessary context to do so [167]. The point is, introducing bugs should be avoided so far as possible in the first place, and programs should routinely have assertions and other methods to detect those bugs that are introduced (see this paper's Supplemental Material for more discussion of standard programming methodologies).

10 Code, data, and publication

All computer systems are in principle equivalent to Turing Machines, and Turing Machines make no distinction between program and data. It is possible to define Turing Machines that do separate program code and data, but as soon as a Universal Turing Machine is constructed, its data *is* code. Indeed, Universal Turing Machines are a theoretical abstraction of virtual machines, which are used widely in practical computing. Java, for instance, runs in a virtual machine, so any Java program code (and any data it uses) is in

fact merely *all* data to the Java virtual machine. At another extreme, λ-calculus is purely program source code, yet λ-calculus is equivalent to Turing Machine computation. Therefore, even the "pure" programs of λ-calculus also represent data.

These elementary theoretical considerations underly an important practical fact: there is no fundamental difference between code and data, and no distinction that is relevant for scientific publication purposes.

There is no code/data distinction one can imagine that cannot easily, even accidentally, be circumvented. In other words, a journal's data policies and code policies should be the identical — and the conventionally stricter data policies should also apply to code. It is baffling that some journals have code policies that are weaker than their data policies; it is certainly indefensible to have no code policies at all.

Significant cyber-vulnerabilities result from there being no difference between code and data. For example: an email arrives, which brings a paper to work on or other data to a user. The user opens the attachment, perhaps a word processor text document, which is more data. The word processor runs macros in the text document — but now it is code. The macros move data onto the user's disk. The data there then runs as code, and corrupts the user's data across the disk — which includes both data and code stored in files. And so on, spreading around the world when this user emails their work to a colleague. Each step of a computer virus infection crosses over non-existent "boundaries" between data and code [167].

This section's discussion may sound like arcane and irrelevant pedantry, but these issues are at the very foundations of Computer Science.[15] If we ignore or misunderstand these basic things — or overlook them in policies and procedures — bugs and irreproducibility are the inevitable (and confusing) consequence.

The main paper points out that data is often embedded in code using "magic numbers." Let's now explain how.

A simple fragment of program code might say

```
x = 324+sin(theta*pi/180);
```

This is clearly all source code, but the number 324 above is likely to be some sort of relevant data, though it might be a physical constant whose value does not depend at all on *this* experiment. The next hard-coded value mentioned in the calculation is difficult to categorize: is the value of π empirical data or is it part of a standard formula? Some programming languages like *Mathematica* treat π as an exact mathematical constant (e.g., *Mathematica* calculates $\tan \pi/4 = 1$ exactly), but π is *also* definitely an inexact empirical value.[16]

The point is, the distinctions between data, program and even mathematical constants are purely a matter of perspective.

Unfortunately, there is data that is extremely easy to overlook (and therefore very hard to manage) because it is embedded in arbitrary ways in code. You may assume that the function `sin`, as used in the calculation example above, is the standard trigonometric function for calculating sines (and because of the π in the expression, you assume `theta` is degrees and `sin` is taking radians as its parameter type) but almost all programming languages allow `sin` to be any function whatsoever. Confusingly, even if it calculates sines, it is generally a different function when the code is run on a different computer producing numbers that are not exactly the same.

It is impossible to tell.

10.a When magic numbers become magic code

Data often controls the flow of code. For example, data summarizing patients may include their gender, but the program processes males and females differently. Then data becomes code.

Arbitrary numbers appearing in code are obviously magic numbers, but code often conceals the magic numbers of data by "programming them away" during the coding process.

For example, the magic number 324 was explicit in the line of code shown above, but if somewhere else the program says

```
if evenQ(324) then A; else B;
```

many programmers would optimize this to `A`, because they know the condition is true because of their assumptions. This now seems to be a more efficient program because it has avoided a test (which a modern complier would have optimized away anyway). Unfortunately, the previously explicit dependency of the code on the magic number 324 has completely disappeared.

[15] Many of the foundational issues were explored thoroughly by Christopher Strachey and others in the 1960s; Strachey's classic lectures are reprinted in an accessible 2000 publication [168]. Being originally a very old paper this classic introduction is much easier to read than many more recent discussions of the foundations of Computer Science.

[16] A record set on 19 August 2021, the most accurate value of π then known was 62,831,853,071,796 digits URL `www.fhgr.ch/en/specialist-areas/applied-future-technologies/davis-centre/pi-challenge`

Obviously this example seems trivial, but it illustrates that programmers do some of their programming while writing code, and many assumptions disappear completely and have no representation in the final code. More complex code will have many facts hard wired into the code — so in fact the code contains data. Code can even read in formulas from data and compile them to perform further calculations, and so on.

This is one reason bugs — effectively incorrect assumptions — are so hard to find, because they have no concrete form in the final program.

10.b When data is code

Many computer programs blur the simplistic code/data distinctions deliberately, to create virtual machines. Data is then run on the virtual machine as program. Many programs provide standard features to do this, such as LISP's and JavaScript's `eval` functions. Henderson's book [169] builds an elegant Pascal program to run *any* LISP program as data, and then shows that the LISP program can run itself running other programs, so it is now its own code and *its* data — despite being purely data to the Pascal program. There are numerous advantages to doing this, including: the Pascal program is not just reading data, but structured data that must conform to the rules of LISP; the LISP running itself runs faster than the original Pascal running LISP, even though the Pascal virtual machine is still doing it in the recursive case; LISP is a much more powerful language than Pascal, so a virtual machine can be used to escape the barriers of a limited implementation such as Pascal. In short, any distinctions between code and data are impossible to maintain.

AI and Machine Learning are further examples of exploiting data as code. Typically a program learns from a training set of data, and then processes future data differently depending on what it has learned. In other words, the original data becomes a model which is now code.

10.c Exploiting code as data for more reliable science

In the present paper, we knowingly built on this blur between data and code, a special case of RAP+. However, what we did was not unusual except in our explicit and rigorous approach to managing and summarizing data reliably in the paper.

The paper and its Supplemental Material are typeset in LaTeX, a popular typesetting language. LaTeX not only has text (as you are reading right now) but it also has code. For example, "LaTeX" was typeset by running the code for a macro called `\LaTeX`, which then calculated how to position the letters as they are wanted. When π was written above, the code that generated what you read actually said `$\pi$` — so is this data that just says π or is it code that tells the computer to change character sets from Latin to Greek, and then uses `\pi` as a program variable name to select a particular glyph from the data about typesetting Greek characters? The distinctions are all a bit moot. In other words, the publication itself is data to a LaTeX program, and within that data it includes further programs. Indeed, LaTeX is run on a virtual machine, in exactly the same way that Henderson's LISP is, and doing so provides the same advantages.

The data for this paper's survey was itself originally written as literal text in LaTeX: it meant that LaTeX could process it to produce a typeset table (as in the Supplemental Material above). As the extent of the data grew, it rapidly became apparent that LaTeX is a poor choice to manage structured data. A simple JavaScript program was written to convert the LaTeX data into JSON (which is much more readable than LaTeX) and also generate CSV files that can be processed in standard office software such as Excel, which some readers may prefer. In fact, examining and comparing the same data in the contrasting formats, this typeset file, in JSON, and in Excel (reading the generated CSV) provided multiple different perspectives of the data that increased redundancy and confidence that the data was correct and correctly handled.

It is important to note that using such techniques is quite routine in science publication, though often pre-existing tools are used to streamline the process (and to ensure that it is more widely understood). The paper [187], for example, in addition to using a typesetting system for publication, also placed its code in a repository using R Markdown [117], a programming environment based on R designed for generating and documenting lab books — almost the polar opposite of LaTeX, which is designed for publication but can be used for programming.

Finally note that what may look like magic numbers used throughout the present paper (such as the 32, as in "32 papers were evaluated") are all in fact named, calculated and placed *in situ* directly from computations performed on the JSON paper's data.

10.d When data is text: Exploiting code for reliable publication

Section 3.a of the main paper looks like part of an ordinary paper, but it (including the figure and calculations) was data generated by a *Mathematica* program.

Most programs are code plus comment, and their data comes from some external source or sources. In *Mathematica*, programs are represented as "notebooks," which can be structured like reports or papers. They have sections, which allow program code, data and program output to be arbitrarily mixed in a single file.

For the purposes of the present paper, a notebook was created with a new type of data, LaTeX text. The LaTeX text can be any mix of text written by the author or material generated by running *Mathematica*. A final step of running the *Mathematica* notebook is to collect all of the LaTeX material, whether written by hand or generated, and save it to a normal text file for LaTeX to typeset.

The idea is very simple, but very effective. The "code" for the paper includes the *Mathematica* notebook that generated section 3.a. In fact, the notebook is written as a self-contained report or paper, as *Mathematica* notebooks generally are, and it thoroughly documents how it works.

For readers of this paper who do not have access to *Mathematica* to run the notebook, a PDF of the notebook is included to show how it works. Note that the method can be applied in any programming language, but *Mathematica* makes the interleaving of paper text and calculations (of arbitrary complexity) very easy — in conventional programming environments the paper text would be separate (e.g., as data) and it would be much harder to keep the text and code in synchronization.

More of such techniques for improving reliability are discussed in section 12.d, where they are applied to accurately and reliably reporting the survey reported in the main paper.

10.e Data and polynomials used in the paper

The data and polynomials used in the main paper's section 3.a, and illustrated in the paper's figure 1, is presented below, as generated in LaTeX by the same *Mathematica* notebook that generated section 3.a. Of course, in the paper itself, the specific data was not necessary to make the point, but if there is any need to replicate it or otherwise scrutinize the arguments in the paper — as there would be for more complex arguments in typical papers — the data used and results are shown below.

The table below is a LaTeX table generated by *Mathematica*, and is exactly the data used in the paper.[17] In a similar way, data and computed results could be presented and made available in other scientific papers.

$x =$	1.00	1.10	2.20	2.60	4.50
$y =$	1.50	2.70	4.90	5.70	8.20

Showing both polynomials with coefficients to 2 decimal places, the linear least squares model to fit this data is:

$$\hat{y} = 0.49 + 1.80x$$

and the Lagrange polynomial model (an exact fit to the 5 data points) is

$$\hat{y} = -41.79 + 85.46x - 56.30x^2 + 15.65x^3 - 1.51x^4$$

These polynomials are the ones shown in the main paper's figure 1.

The name of the relevant *Mathematica* notebook file where all data and code for this section (and for generating the paper's section 3.a and its figure 1) can be found in `programs/over-fitting-section.nb`, which is included in the Git repository for the paper.

10.f Comparing conventional and RAP approaches

The similarities and differences between the conventional copy-and-paste approach to filling in data and diagrams in publications, the improved systematic RAP process, and using notebooks (such as *Mathematica* or Jupyter), are illustrated in the sequence of schematics of figures 2, 3, 4, and 5.

11 The Spiegelhalter trustworthiness questions

David Spiegelhalter is concerned how statistics is often misused and misunderstood. In his *The Art of Statistics* [48] Spiegelhalter brings together his advice for making reliable statistical claims: they need to be accessible, intelligible, assessable, and usable — and the claims need to be properly accountable.

[17] Barring coding or other errors of course, which here we checked against manually by comparing this table typeset in LaTeX against the raw data in the original *Mathematica* data table, but in general might better be done by an automatic round trip — though that would not easily spot LaTeX errors.

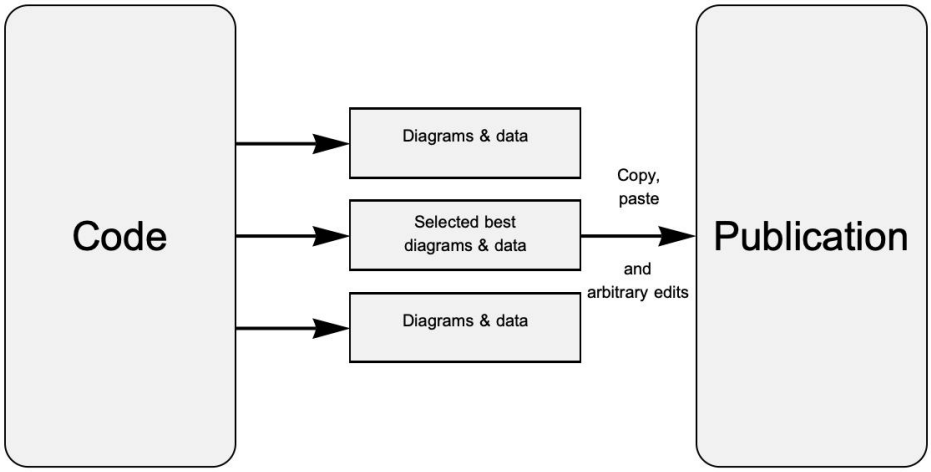


Figure 2: The common basic — error-prone and obsolete — approach to scientific authoring is to use code to help generate analyses and diagrams, then manually copy and paste the selected results into the publication. Note that the publication, including the results, can be edited arbitrarily, and typically the results published will have been edited and modified (if only for typographical purposes) from those actually generated by the code.

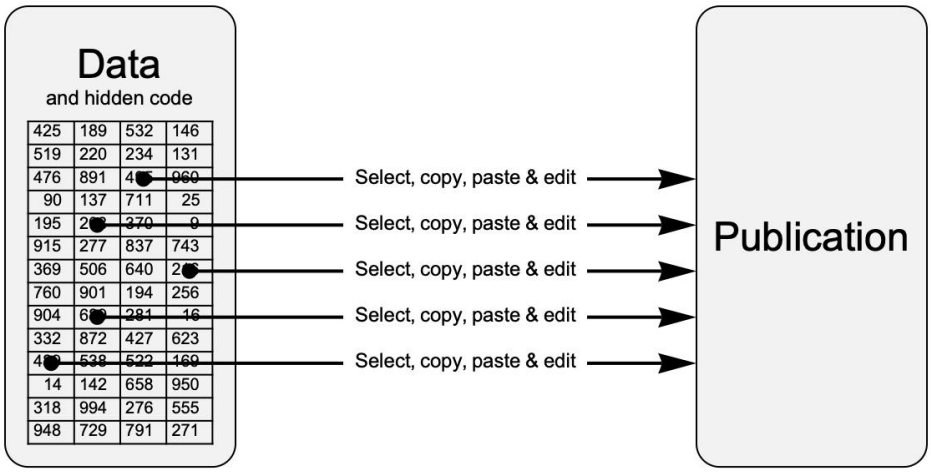


Figure 3: When the code is in a spreadsheet, such as Microsoft Excel, the code is generally hidden from sight. The data copied & pasted into a publication may or may not be calculated from data in other cells (such as column totals). Records are rarely taken of these manual processes, and, anyway, typically it is impossible to be certain exactly what has been copied unless very great care is taken. In consequence, if a spreadsheet is modified, it is haphazard what results are updated and corrected in the publication.

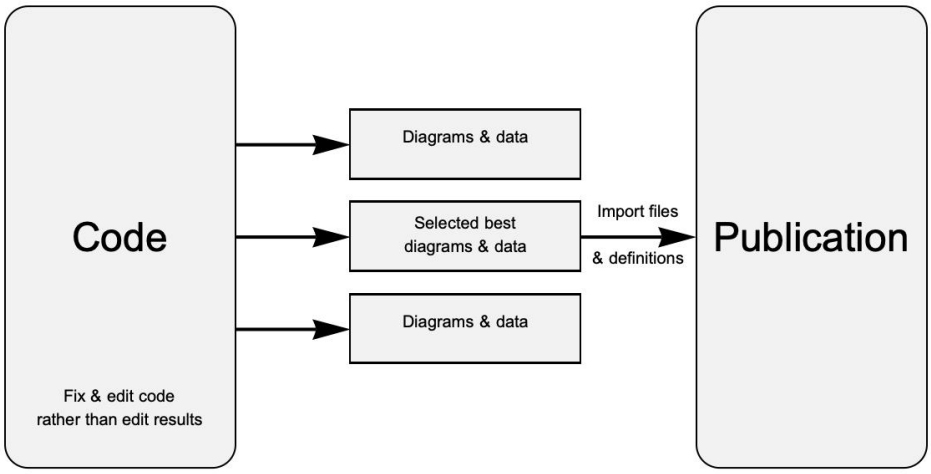


Figure 4: Improving over the normal approach (figure 2), data and diagrams are placed in the publication using a programmed, systematic approach. In the present paper, this was done by generating text files of LaTeX definitions (represented by boxes in the central column of the schematic), hence providing LaTeX names for all of the code-generated values. Since the copying of results into a publication is automated and easy, any improvements to the results (such as correcting errors) are made by improving the code — which therefore contributes to improving all future-generated results too. Not shown, but in well-engineered code, documentation will also be generated, such as by using tools such as JavaDoc or Doxygen.

Spiegelhalter proposes ten questions to ask when confronted with any claim based on statistical evidence. Some of his questions are quite general, and might be applied to any sort of scientific claims, but all have analogous questions that could be addressed to software code or publications relying on code — analogues are suggested in **bold** below.

What might seem like dauntingly technical software issues are no more demanding than the basic statistical issues that are regularly acceded to; failing to ask these questions is as risky as dismissing statistical scrutiny.

11.a How trustworthy are the numbers?

1. *How rigorously has the study been done?* For example, check for 'internal validity,' appropriate design and wording of questions, pre-registration of the protocol, take a representative sample, using randomization, and making a fair comparison with a control group.

   - **How rigorously has the Software Engineering been done? Section 9 in the Supplemental Material provides a list of important issues that must be addressed for any reliable software.**

   - **"Internal validity" assumes that there is evidence the programmers had uncertainty in the code's reliability and checked it. Were different methods used and compared, or was all confidence put into a single implementation? What internal consistency checks does the implementation have? Were invariants and assertions defined and checked?**

2. *What is the statistical uncertainty/confidence in the findings?* Check margins of error, confidence intervals, statistical significance, multiple comparisons, systemic bias.

   - **How are the claims presented that give us confidence in the code that they are based on? Are there discussions of invariants, independent checks for errors, and so on? Again, Supplemental Material section 9 provides further discussion of such issues.**

3. *Is the summary appropriate?* Check appropriate use of averages, variability, relative and absolute risks.

   - **If the claims are exploratory, weaker standards of coding can be used; if the claims are a basis for critical decisions, then there should be evidence of using appropriate**

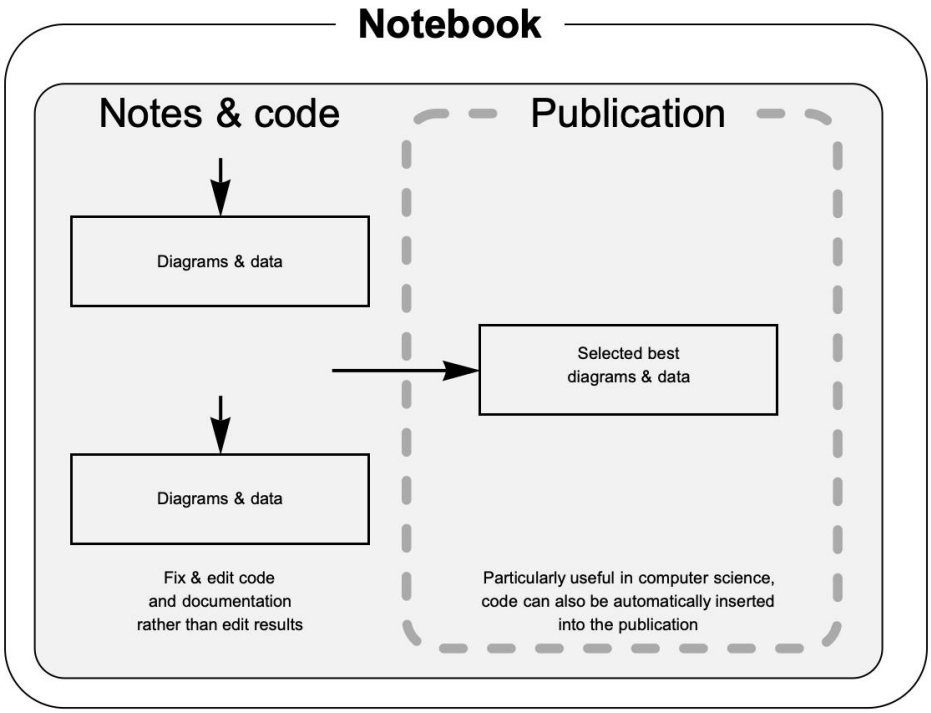


Figure 5: In a notebook system, such as *Mathematica* or Jupyter, a single document — the notebook — integrates the publication, the code, any notes, as well as all the results. The code generates results and images that are automatically (and reliably) inserted in place into the notebook, usually right after the code that generates them. Some parts of the notebook are marked or selected to be printed as the publication, thus allowing most if not all code to be hidden from the final publication. In well-engineered code, the notebook will directly contain full documentation. Note that in most systems, notebooks can import and generate arbitrary data (images, sounds, sensor data, etc).

Software Engineering (such as defensive programming) to provide appropriate confidence in the results claimed.

11.b How trustworthy is the source?

4. *How reliable is the source of the story?* Consider the possibility of a biased source with conflicts of interest, and check publication is independently peer-reviewed. Ask yourself, 'Why does this source want me to hear this story?'

   - **The source of many science stories is the output of running some code. How reliable is this code? What evidence is there that the code was well-engineered so its reliability can be trusted?**

   - **What evidence is there of rigorous (e.g., code review and tool-based) independent methods being used to manage coding bias?**

5. *Is the story being spun?* Be aware of the use of framing, emotional appeal through quoting anecdotes about extreme cases, misleading graphs, exaggerated headlines, big-sounding numbers.

   - **Be wary of AI and ML which may have been trained by chance or specifically (if not deliberately) to get the results described.**

6. *What am I not being told?* This is perhaps the most important question of all. Think about cherry-picked results, missing information that would conflict with the story, and lack of independent comment.

   - **Cherry picking with code is often unconscious and is very common: when running code produces the "cherries" for a paper it is tempting to stop testing the code, and just assume it is running correctly. So, what evidence is there that the code was rigorously developed and cherry picking avoided?**

11.c How trustworthy is the interpretation?

7. *How does the claim fit with what else is known?* Consider the context, appropriate comparators, including historical data, and what other studies have shown, ideally in a meta-analysis.

   ▶ **Is there any discussion of the code and how does it compare with other peer-reviewed publications using code used for similar purposes?**

8. *What's the claimed explanation for what has been seen?* Vital issues are correlation v. causation, regression to the mean, inappropriate claim that a non-significant result means 'no effect,' confounding attribution, prosecutor's fallacy.

   ▶ **These are all good statistical questions. The Software Engineering analogy is: are the claims backed up by a sufficiently detailed discussion of the algorithms and Software Engineering that justify the appropriateness of the chosen software implementation? The Supplemental Material list in section 9 provides examples of expected explanations for the trustworthiness of running some code.**

9. *How relevant to the story is the audience?* Think about generalizability, whether the people being studied are special case, has there been an extrapolation from mice to people.

   ▶ **Generalizability is equivalent to is the code available, easy to understand and use for more general purposes — including further work and checking the reproducibility of the claims being made?**

10. *Is the claimed effect important?* Check whether the magnitude of the effect is practically significant, and be especially wary of claims of 'increased risk.'

12 A pilot survey of computational science

The main paper was motivated by selected epidemiological papers and their problematic computational models (which are discussed in the paper). Although concerning in their own right, especially for informing national public health policies during a pandemic, the wider question is: are the problems illustrated by these case studies typical of science more broadly? We undertook, then, some selected studies of papers in a variety of fields, then undertook a randomized, stratified pilot survey covering several leading peer reviewed journals. The point was not to establish the frequency of problems, so much as to sign whether the problematic case study was exceptional or typical. It is typical.

The pilot study itself follows the RAP methodology. The data and code for this paper's pilot survey (and all other analysis used in the paper and in this Supplemental Material) are available on GitHub. All raw data is converted into LaTeX so that the analysis can be typeset directly in the paper; this Supplemental Material also contains a complete tabular presentation of the data.

Moreover, this paper itself follows the more general RAP+ methodology. For example, as standard practice, a Unix makefile is used to make it easy to analyze and generate all data, typeset the paper, and more. Table 6 shows the options provided.

The RAP+ approach cannot enforce the truth of such as summary, but it (and the opportunity to read and review it easily) very significantly increases the chances that the summary is correct and up to date. For example, if refactoring leads to an option being deleted, then it will also disappear *with no further work* from the table above. Also, since the summary was proof-read at the same time as proof-reading this Supplemental Material, following RAP+ also increases the chances that any errors or functionality omissions or issues in the makefile have been detected and corrected.

12.a Selected journal case studies

There are numerous case studies to be made from journals and their relation to code and data. Here, we select just two.

12.a.1 *The Lancet*

The journal *The Lancet* published and then subsequently retracted a paper on using hydroxychloroquine as a treatment for COVID [170]. The paper was found to rely on fraudulent data [133, 134]. *The Lancet* subsequently tightened its data policies [135], for instance to require that more than one author must have directly accessed and verified the data reported in the manuscript. Curiously, the original (now retracted) paper declares

`make all`	Analyze the data, then typeset the main PDF files (`paper-seb-main.pdf` and `paper-seb-supplementary-material.pdf`).
⋮	
`make data`	Analyze the data, and generate all the data files, the Unix scripts, the CSV, and LaTeX files (including the LaTeX summary of this makefile), etc. This `make` option runs `node programs/data.js`, downloads the Git repositories used in the pilot survey, and then analyzes them. Note that downloading all the repositories in a reasonable time needs decent internet bandwidth.
⋮	
`make git-prep`	What's on Git that we've lost, or stuff we have got locally but probably don't want on Git, so you can delete it or move it out the way or whatever.
⋮	
`make really-tidyup`	More thorough than `make tidyup` — remove *all* files that can be recreated. (This will mean next time you run LaTeX you will have to ignore errors as the .aux files are re-created.)

Table 6: Conforming to the RAP+ methodology, the abbreviated summary above was generated automatically, by using `make data`. A full list of `make` options is generated by `make` or `make help` when run on the Unix command line.

> "... all authors participated in critical revision of the manuscript for important intellectual content. MRM and ANP supervised the study. All authors approved the final manuscript and were responsible for the decision to submit for publication."

which seems to suggest that several original authors of the paper would have been happy to make the new declarations — and, of course, if there is fraud (as was established in this case) it seems likely that authors who make the new declarations of accessing and verifying data are unlikely to make reliable declarations.

The Lancet still has no code publication policy, and for more than one author to have "direct access" to the data they are very likely to access the data through the same code. If the code is faulty or fraudulent, an additional author's confirmation of the data is insufficient, and there is at least as much reason for code to be fraudulent (not least because code is much harder to scrutinize than data). Code needs more than one author to check it, and ideally reviewers independent of the authors so they do not share the same assumptions and systems (for instance shared libraries, let alone potential collusion in fraud).

12.a.2 *Journal of Vascular Surgery*

In 2020 the *Journal of Vascular Surgery* published a research paper [136], which had to be retracted on ethical grounds [137, 138]: it was a naïve study and the editorial process was unaware of digital norms. Notably, the paper fails to provide access to its anonymized data (with or without qualification), and fails to define the data anonymization algorithm, and also fails to even mention the code that it developed and used to perform its study. The journal's data policy is itself very weak (the authors "should consider" including a footnote to offer limited access to the data) and, despite basic statistics policies, it has no policy at all for code (see section 12.c.1). Ironically, the retracted article [136] is still online (as of August 2020) with no reference to any editorial statement to the effect that it has been retracted, despite this being trivial — and necessary — to achieve in the widely-accessed online medium.

Medical research often aims to establish a formula to define a clinical parameter (such as body mass index, BMI) or to specify an optimal drug dose or other intervention for treatment. These formulas, for which there is conventional clinical evidence, are often used as the basis for computer code that provides advice or even directly controls interventions. Unfortunately a simple formula as may be published in a medical paper is *never* sufficient to specify code to implement it safely. For example, clinical papers do not need to evaluate or manage user error when operating apps, and therefore the statistical results of the research will be idealistic compared to the outcomes using an app under real conditions — which is what the clinical research is supposedly for. A widespread bug (and its fix) that is often overlooked is discussed in [144]; the paper includes an example of a popular clinical calculator (based on published clinical research) that calculated nonsense, and potentially dangerous, results. The paper [171] summarizes evidence that such bugs, ignored by the clinical research literature, are commonplace in medical systems and devices.

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

Table 7: Overview of the peer-reviewed paper sample. (Convenient copy of main paper's table 1.)

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 8: Summary of survey results. (Convenient copy of main paper's table 2.)

12.b Pilot paper sample

A sample of 32 recent papers covering a broad range of science were sampled from the leading journals *Lancet Digital Health* ($N = 6$), *Nature Digital Medicine* ($N = 12$) and *Royal Society Open Science* ($N = 14$).

The two journals *Nature Digital Medicine* and *Lancet Digital Health* were selected as leading specialist science journals in an area where correctness of scientific modeling has safety-critical implications, and *Royal Society Open Science* was selected as a leading general science journal. All papers sampled are Open Access, although for some papers some or all of the associated data has no or restricted access, in some cases despite the relevant journal policies on code. Table 7 is an overview of the sample.

Papers were selected from the journals' July 2020 then new online listings where the paper's title implied that code had been used in the research. Commentary, correspondence, and editorials were excluded. The sample is intended to be large enough and objective enough to avoid the selection bias in the papers that motivated the current paper (the sample excludes the motivating papers discussed above as they were not published in the sampled journals), so that the sample may be considered to fairly represent what the editorial and the broader peer review community in leading journals considers to be good practice for computationally-based science. The selection criterion selected papers where the title implies the authors themselves considered code to be a significant component of the scientific contribution, and, indeed, all sampled papers relied on and assumed the quality of code used in their research.

Github repository and paper citation	PDF paper Number of pages	Repository code & data Number of files	Code kLOC	Data bytes
`AI-CDSS-Cardiovascular-Silo` [184]	6	206	143	64 Mb
`blast-ct` [205]	8	54	3	238 Mb
`covid-sim` [9]	20	229	25	734 Mb
`lactModel` [195]	13	20	2	165 kb
`LRM` [197]	22	125	8	2 Mb
`manifold-ga` [203]	7	11	1	—
`MetricSelectionFramework` [176]	17	44	4	236 kb
`PENet` [180]	9	117	8	4 Mb
`philter-ucsf` [183]	8	1 : 987	13	32 Mb
`PostoperativeOutcomes_RiskNet` [182]	10	1	—	—
`SiameseChange` [186]	9	5	1	1 kb
Average ($N = 11$)	12	254	19	98 Mb

Citation numbers > 129 can be found in the Supplemental Material
Repository clones downloaded and automatically summarized 25 May 2023

Table 9: Sizes of repositories, with approximate sizes of code (in kLOC) and data for all available GitHub repositories reviewed in the survey, plus `covid-sim` [9] for comparison. Sizes are approximate because in all repositories code and data are conceptually interchangeable (an issue explained in the Supplemental Material), so choices were made in the survey to avoid double-counting. Many repositories rely on downloading additional code and data, which is not counted in the table as the additional required material is not in the repository cited in the paper. At the time of cloning and checking all repositories in May 2023, paper [182] still had nothing in its repository except a single file still saying "... code coming soon ...," despite 47 months having already elapsed since the paper had claimed the code could be accessed in its repository.

This convenience sample may be considered to be small given the importance of the research questions and relative to the diversity and huge number of scientific papers,[18] but ...

1. the selected journals are leading peer-reviewed scientific journals that set the standards for scientific publishing practice generally (although the sample shows that code policies are not always enforced);

2. as will be clear from the following discussion, there is little variation across the sample, which implies that a larger sample would not have been productively more insightful (this view is consistent with the multi-disciplinary reports in [96], mentioned in section 5.c);

3. the survey is not intended to be a formal, systematic sample of scientific research in general, but is intended to be sufficient to dispel the possibility that the issues described above earlier in this paper are isolated practice unique to a few papers in epidemiology, perhaps an idiosyncrasy of a few authors in a particular field, or perhaps due to an initial chance selection bias (e.g., the Ferguson papers were reviewed above because of Ferguson's public profile and the importance of dependable pandemic research, but they might have just happened to be Software Engineering outliers);

4. the code/data policies of the 3 journals condoned at the time of the sample *and continue to condone* poor practice at the time of writing the present paper (June 2023) — for specific details and further explanation of the problems, see Supplemental Material section 12.c.1;

5. the fact that the specifically identified problems are elementary errors in Software Engineering (see the discussion in section 5.c) suggests more sophisticated analysis is not required;

6. finally, the present paper's LaTeX source, as well as all documented code and data, are available from a repository, which provides a convenient framework for easily refining or developing the research as may be desired (see details at the end of this paper).

The 32 papers surveyed cover a range of specialities, and it is unlikely that non-specialists can properly assess the code from the point of view of the specialism, not least because many of the papers sampled require specialist code libraries (and in the right combinations of versions) to be run that not everyone will

[18] Using Google Scholar it is estimated that over 40,000 papers meeting the title criteria were published in the month of July 2020.

have or be able to install. Code quality was therefore assessed by reading it — due to the paper authors' complex and/or narrative interpretation of data, code, data and hardware/operating system dependencies, no assessment could realistically be made whether the code provided actually reproduced a paper's specific claims. Indeed, if we trust the papers that their code was actually run and provides the results as reported, then running their code (when provided in full) would merely check the paper/code consistency but will not assess the quality or reliability of the code. Indeed, in most scientific papers there are layers of expert scientific work, interpretation and abstraction, lying between the computational models and the report in the paper.

12.c Summary of results

The sample selection criteria necessarily identified scientific research with Software Engineering contributions.

No evidence of verification and validation was seen. There was only one example of very basic Software Engineering methods, namely independent coding, and even then the independent code used for testing was not uploaded to the paper's code repository, so the independent testing is not available for reviewers or readers of the paper.

There was no evidence of any critical assessment of code, suggesting that scientists writing papers take it for granted that their code works as they intend. No competent programmer would take it for granted that their code was correct without following rigorous methods, such as formal methods, regression testing, test driven design, etc (see section 9 in this Supplemental Material).

Much code depended on specific software versions, specific libraries, and substantial manual intervention to compile it. All code (where actually provided) was sufficiently complex that, if it was to be used or scrutinized, required more substantial documentation than was provided.

On the whole, on the basis of the sample evidence, scientists do not make their code *usably* available, and rarely provide adequate documentation (see table 8).

With the one minor exception, no papers reported anything on any Software Engineering methodologies, which is astonishing given the scale of some of the software effort supporting the papers (table 9). The papers themselves, typically only a few published pages, are very brief compared to the substantial code they rely on (see table 9).

With the one exception, none of the papers used any specific Software Engineering methods, such as open source [113] or other standard methodologies provided in this Supplemental Material, to help manage their processes and help improve quality. Although software stability [172] is a relatively new concept, understood as methodologies, such as portability, to provide long-term value of software, it is curious that none of the papers made any attempt at stability (however understood) despite the irony that all the papers were published in archival journals.[19]

Nature Digital Medicine and *Royal Society Open Science* have clear data and code policies (see Supplemental Material section 12.c.1), but actual publishing practice falls short: 11 out of the 26 papers (42%) published in them and sampled in the survey manifestly breach their code policies. In contrast, *Lancet Digital Health*, despite substantial data policies, has no code policy at all to breach. The implication is that the fields, and the editorial expertise of leading journals, misunderstand and dismiss code policies — they (or their editors and reviewers) are technically unable to assess them. This lack of expertise is consistent with the limited awareness of Software Engineering best practice that is manifest in the published papers (and resources) themselves.

Code repositories were used by 10 papers (31%), though one paper in the survey claimed to have code on GitHub but there was no code in the repository, only the comment "Code coming soon..." (checked at the time of doing the review, then double-checked as detailed in the references in the Supplemental Material, as well as most recently on 25 May 2023 while checking table 9): in other words, the repository had never been used and the code could never have been looked at, let alone reviewed.[20] This is a pity because GitHub provides help and targeted warnings and hints like "No description, website, or topics provided [...] no releases published." The lack of code is ironic: the paper concerned [182] has as its title "*Development and validation* of a deep neural network model [...]" (our emphasis), yet it provides no code or development processes for the runnable model it claims to validate, so nobody else (including referees) can check any of the paper's specific claims.

The sizes of all GitHub repositories are summarized in table 9 (since many papers not using GitHub do not have all code available, non-GitHub code sizes are not easily compared and are not listed).

[19] Reasons the present paper does not directly assess the quality of software in the surveyed papers include: many papers did not provide complete software; it was not possible to find correct versions of all software systems to run the models; also, no papers provided adequate test suites so that correct operation of software could be confirmed objectively.

[20] GitHub records show that it had not been deleted after paper submission.

Overall, there was no evidence that any code had been developed carefully, let alone by using recognized professional Software Engineering methods. In particular, no papers in the survey provide any claims or evidence of effective testing, for instance with evidence that tests were run on clean builds. While it may sound unrealistic to ask for evidence on software quality in a paper written for another field of science, the need is no less than the need for standard levels of rigor in statistics reporting, as discussed in the opening of this paper.

Data repositories (the Dryad Digital Repository, Figshare or similar) were used by 9 papers to provide structured access to their data. Unlike GitHub, which is a general purpose repository, Dryad has scientifically-informed guidelines on handling data, and all papers that used Dryad provided more than just their raw data — they provided a little, sometimes substantial, documentation for their data. At the time of writing, Dryad is not helpful for managing code — its model appears to be founded on the requirement that once published papers must refer to exactly the data they used, so further refinements on the data (or code) are taboo, even with version control.

12.c.1 Current code policies of sampled journals

It is noteworthy that none of the journals sampled permit any reliable style of managing data in published papers, such as described above in sections 12.d.1 and 12.d.3. The main paper, section 6.e, additionally mentions *PLOS ONE* and *IEEE Transactions on Software Engineering*.

For all the surveyed papers that had accessible code, the code included explicit (and relevant) data that was not archived *as* data in the journal repositories.

Note that journal policies relevant to the survey were first accessed on 29 July 2020, close after the period covered by the pilot survey, thus presumably fairly closely reflecting the policies in use for the papers in the survey on the dates when they were each submitted. Unfortunately for the survey, the journals do not make clear what exact policies were applied to each paper when the papers were submitted (on the other hand, the survey shows that policies are not rigorously enforced).

Extract from *Royal Society Open Science* author guidelines

"It is a condition of publication that authors make the primary data, materials (such as statistical tools, protocols, software) and code publicly available. These must be provided at the point of submission for our Editors and reviewers for peer-review, and then made publicly available at acceptance. [...] As a minimum, sufficient information and data are required to allow others to replicate all study findings reported in the article. Data and code should be deposited in a form that will allow maximum reuse. As part of our open data policy, we ask that data and code are hosted in a public, recognized repository, with an open licence (CC0 or CC-BY) clearly visible on the landing page of your dataset."

URL `royalsociety.org/journals/authors/author-guidelines/#data`

> Since first accessed 29 July 2020, the policy has been revised (undated, accessed 2 February 2022) but retains the same principles; full policy now available via a DOI [173]. The policy still retains an emphasis on data accessibility, and continues a lack of awareness that code and data are equivalent and often mixed (see section 10).

Extract from *Nature Digital Medicine* author guidelines

"A condition of publication in a Nature Research journal is that authors are required to make materials, data, code, and associated protocols promptly available to readers without undue qualifications. [...] A condition of publication in a Nature Research journal is that authors are required to make unique materials promptly available to others without undue qualifications."

URL `www.nature.com/nature-research/editorial-policies/reporting-standards#availability-of-data`

> Accessed 29 July 2020; since updated (accessed 2 February 2022) to require [in part] "Upon publication, Nature Portfolio journals consider it best practice to release custom computer code in a way that allows readers to repeat the published results. Code should be deposited in a DOI-minting repository such as Zenodo, Gigantum or Code Ocean and cited in the reference list following the guidelines described here."

***Lancet Digital Health* author guidelines**

Journal has detailed data policies, but no code policy.

URL `marlin-prod.literatumonline.com/pb-assets/Lancet/authors/tldh-info-for-authors.pdf`

> Accessed 29 July 2020. Still no code policy when accessed 2 February 2022.

Extract from *Journal of Vascular Surgery* author guidelines

The *Journal of Vascular Surgery* has detailed data policies, but no code policy. While no *Journal of Vascular Surgery* papers were surveyed (but see section 12.a.2), the following statement on data policies is relevant:

"The authors are required to produce the data on which the manuscript is based for examination by the Editors or their assignees, should they request it. [...] The authors should consider including a footnote in the manuscript indicating their willingness to make the original data available to other investigators through electronic media to permit alternative analysis and/or inclusion in a meta-analysis."
URL `www.editorialmanager.com/jvs/account/JVS_Instructions%20for%20Authors2020.pdf`
| Accessed 29 July 2020. Policy unchanged when accessed 2 February 2022.

12.c.2 Sample assessment and scoring

Assessment flags are highlighted in color to be clearer in the following tables.

Flag	Paper count	Meaning
P_c	26	Journal has a code policy (see section ??)
$P_{c\text{-breach}}$	11	Paper breaches journal code policy (see section ??)
R_c	10	Paper uses a code repository (e.g., GitHub)
$R_{c\text{-empty}}$	1	Code repository contains no code
R_d	9	Paper uses a data repository (e.g., Dryad, Figshare, GitHub)
S_{NONE}	12	No code available at all (note: code is not expected for standard models, systems or statistical methods)
S_p	8	Paper says source code is available in principle
S_+	12	Paper or URL provides source code
$S_{rigorous}$	0	Evidence that source code was developed rigorously
S_{tested}	0	Evidence that source code has been run with a clean build and tested
S_{tools}	0	Evidence of any tool-based development
$S_{open\ source}$	0	Team or open source development
$S_{otherSE}$	1	Other evidence of good practice; see details in summary table
C_0	10	Code only has comments unrelated to code intent (e.g., copyright)
C_1	1	Code only has trivial or obvious comments
C_2	3	Helpful comments explaining code intent, rather than rephrasing the code
C_3	2	Code has substantial, useful comments, and documentation
C_a	1	Code in repository uses assertions
C_c	0	Code in repository uses pre- or post-conditions or similar

Ref	Data	Code
[175]	On request.	"Code is available upon request from the corresponding author" (requested) P_c S_p
[176]	"The datasets used in the current study are available from the corresponding author upon reasonable request and under consideration of the ethical regulations." R_d	Matlab. Documented overview, but only trivial comments. Assertions in code only relate to UI. P_c R_c S_+ C_0 C_a
[177]	"In accordance with Twitter policies of data sharing, data used in the generation of the algorithm for this study will not be made publicly available."	"Due to the sensitive and potentially stigmatizing nature of this tool, code used for algorithm generation or implementation on individual Twitter profiles will not be made publicly available." P_c $P_{c\text{-breach}}$ S_{NONE}
[178]	"The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request."	"This code would be made available upon reasonable request." (requested) P_c S_p
[179]	Nothing available	Nothing available (despite building two voice-based virtual counselors). P_c $P_{c\text{-breach}}$ S_{NONE}
[180]	"The datasets generated and analyzed during the study are not currently publicly available due to HIPAA compliance agreement but are available from the corresponding author on reasonable request."	Poor commenting, no documentation. P_c R_c S_+ C_0

Ref	Data	Code
[181]	"The dataset generated and analyzed for this study will not be made publicly available due to patient privacy and lack of informed consent to allow sharing of patient data outside of the research team."	No code available. P_c $P_{c\text{-breach}}$ S_{NONE}
[182]	"The datasets generated during and/or analyzed during the current study are not publicly available due to institutional restrictions on data sharing and privacy concerns. However, the data are available from the corresponding author on reasonable request."	Empty GitHub repository: "Code coming soon . . . " paper says. P_c $P_{c\text{-breach}}$ R_c $R_{c\text{-empty}}$ S_{NONE}
[183]	"The i2b2 data that support the findings of this study are available from i2b2 but restrictions apply to the availability of these data, which require signed safe usage and research-only. Data from UCSF are not available at this time as they have not been legally certified as being De-Identified, however, this process is underway and the data may be available by the time of publication by contacting the authors. Requesters identity as researchers will need to be confirmed, safe usage guarantees will need to be signed, and other restrictions may apply."	Basic documentation, very little comment. P_c R_c S_+ C_0
[184]	"Not available due to restrictions in the ethical permit, but may be available on request."	Trivial comments, no documentation. P_c R_c S_+ C_0
[185]	"The data that support the findings of this study are available in a deidentified form from Cleveland Clinic, but restrictions apply to the availability of these data, which were used under Cleveland Clinic data policies for the current study, and so are not publicly available."	"We used only free and open-source software" — some of which is unspecified. P_c $P_{c\text{-breach}}$ S_{NONE}
[186]	"The i-ROP cohort study data for ROP is not publicly available due to patient privacy restrictions, though potential collaborators are directed to contact the study investigators . . . "	Not all code on GitHub, minor comments only. P_c R_c S_+ C_0
[187]	Data available on Dryad. R_d	Code and example runs available in R Markdown. P_c S_+ C_3
[188]	Data directly written into program code.	Basic Matlab with routine comments. P_c $P_{c\text{-breach}}$ S_+ C_0
[189]	Data available on Dryad plus publicly available data from the 1000 genomes project. Currently (apparently) for private view. R_d	Code available for private view, though some code available with minor comments. Paper describes using two contrasting methods to help confirm correctness, "As an additional check, I also coded the calculation of D based on a probabilistic approach, using genotype frequencies in each population to calculate the expected frequencies of each possible two-genotype combination (electronic supplementary material, table S1). Essentially identical results were obtained." — but the contrasting method is not available. P_c S_p $S_{otherSE}$ C_2
[190]	Data available on Dryad. R_d	Reasonaby commented code on Dryad, but code is not complete and presumably never checked. P_c S_p C_2
[191]	On request.	R, lightly commented. P_c S_p C_0

Ref	Data	Code
[192]	No data required.	Unrunnable incomplete code fragment. P_c $P_{c\text{-breach}}$ S_p
[193]	Data embedded in PDF.	No code available. P_c $P_{c\text{-breach}}$ S_{NONE}
[194]	Data available on Dryad. R_d	Some comments, some code in Matlab. P_c S_p C_2
[195]	Partial data on Dryad. R_d	Documented R, including model's manual. P_c R_c S_+ C_3
[196]	No data required.	"We constructed a bioeconomic model for an RSSF [restricted fishing effort small-scale fishery] using game theory" for which results are discussed, yet no code is available. P_c $P_{c\text{-breach}}$ S_{NONE}
[197]	Data cited, not all available.	Trivial documentation. P_c R_c S_+ C_0
[198]	On Figshare. R_d	On Figshare, large amount of disorganised and undocumented code. Helpful features to make usable for third parties. P_c S_+ C_0
[199]	Data on Dryad. R_d	No code available. P_c $P_{c\text{-breach}}$ S_{NONE}
[200]	Data on various web sites.	No code available. P_c $P_{c\text{-breach}}$ S_{NONE}
[201]	Data on request.	"The coding used to train the artificial intelligence model are dependent on annotation, infrastructure, and hardware, so cannot be released." (!) Algorithm (not source code) available on request. S_{NONE}
[202]	Data on request.	Python scripts can be requested. S_p
[203]	Unspecified location on large website requiring registration. R_d	Has overall documentation but poorly commented Matlab code on GitHub. R_c S_+ C_0
[204]	Available to researchers who meet criteria for access to confidential data.	Despite the paper being a "deep learning algorithm" the code is not available. S_{NONE}
[205]	Data access conditional on approved study proposal.	Almost completely uncommented Python, but does have a basic setup script. R_c S_+ C_1
[206]	Unspecified locations on several large websites.	Python used and apparently on GitHub, but — an oversight? — no code is available. S_{NONE}

12.d Assessment criteria and methods

A survey sampled of recent papers that were published online in July 2020, accepted for publication after peer review in 3 high-profile, highly competitive leading peer-reviewed journals, namely *Lancet Digital Health* ($N = 6$), *Nature Digital Medicine* ($N = 12$) and *Royal Society Open Science* ($N = 14$). Papers were selected from the journals' July 2020 new online listings where the paper's title implied that code had been used in the research. Commentary, correspondence and editorials were excluded. The sample represents what the editorial and the broader peer review community considers to be good practice.

The selection process will have certainly missed some papers that use code, but the criterion selects papers where the wording of the title indicates that the authors consider code to be a component of the scientific contribution. Indeed, all sampled papers used code in their research. Although there is unavoidable subjectivity in the paper evaluations and uncontrolled bias from using a single evaluator (the author of this paper), it is hoped that using a sample of 32 papers from 3 diverse journals is sufficient to randomize errors so that they largely cancel out, and the overall trends as discussed in this paper are reliable. It should be noted that, except where a paper provides a URL to a code repository, much code was disorganized so possibly not all code was reviewed because it was too hard to find (some emails to authors have not been responded to).

Since almost every scientific paper relies on generic computer code (calculating statistics, plotting graphs, storing and manipulating data, accessing internet resources, etc), the baseline of papers using code was not assessed. Papers whose title indicated their contribution included or relied on bespoke code were selected, and all those clearly relied heavily on their own specifically developed code. Papers that may have relied on bespoke code but whose titles made no such implication were not assessed.

Although the pilot survey is not a systematic review, following Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) [63] it is good practice to disclose details of the reviewers. In the present case, the selected papers were assessed by the author of the present paper. The study was not blinded. This is, of course, a limitation of the study. However, the reviewer is a full professor of computer

science, who has taught and assessed computer software since the 1970s, using moderated and peer-reviewed processes for undergraduate and postgraduate computing degrees, and is well aware assessing code quality has been a lively topic in Software Engineering for decades (there is now international standard ISO/IEC 9126, updated to ISO 25000). The author has written legal documents analyzing software for criminal cases involving faulty software. The author has approximately 528 published papers in computer science.

The evaluations performed for the present paper are at a trivial level where sources of bias should have a negligible effect, particularly given that the overall conclusion is consistent across both the diverse sample and the computational science-based papers cited in the paper that did not form part of the selected sample. As said in the body the paper: *the fact that the specifically identified problems are elementary errors in Software Engineering (see the discussion in* [main paper] *section 5.c) suggests more sophisticated analysis is not required.*

In any case, as stated in the main paper, the full dataset and analysis code is available at

URL `github.com/haroldthimbleby/Software-Enginering-Boards`

and the reviewed papers are (unless retracted) still available online for independent assessment.

There is considerable debate over what good commenting practice is, but this is because comments have many roles — from helping students to get marks in assessments, asserting intellectual rights, reminding the developer of things to do, managing version control, to explaining the code to third parties. Different programming languages also develop "cultures" and tool-based systems that encourage different approaches for comments (examples include R Markdown, Mathematica Notebooks, JavaDoc, Haskell's Haddock, and so on). For scientific code, however, the explanatory role is critical, and this is what was assessed in the present survey. It is notable that no such tool-based approach to code or documentation was used in any code reviewed.

The completeness or executability of code was not assessed, although if code was obviously incomplete this was noted. Whether code runs as claimed is a matter of research integrity, which is beyond the scope of this survey. What is relevant to the study is whether the code is described in sufficient detail that the methods used can be scrutinized. Obviously being able to run the code will help, but clarity in documentation and comments is critical. It is more like "can we see the critical pages from your lab book so we understand what you did?" rather than "can we have a free run of your laboratory, even though we don't understand the details of the science?"

As an informal survey, intended to establish whether the issues in epidemic modeling were more widespread, and given the very poor level of documentation found in scientific code, it was not felt necessary to have independent or blind assessment.

The data was recorded in JSON (JavaScript Object Notation), which is a simple standard data format. A typical entry in the data file looks like this (with long field values truncated for clarity):

```
{
  accessed:  "14 July 2020",
  doubleChecked:  "17 January 2021",
  authors:  "Callahan A, Steinberg E, Fries JA, Gomba ...,
  year:  2020,
  title:  "Estimating the efficacy of symptom-based ...,
  volume:  3,
  number:  95,
  journal:  "Nature Digital Medicine",
  doi:  "10.1038/s41746-020-0300-0",
  dataComment:  "On request.",
  hasCodeInPrinciple:  1,
  codeComment:  "‘‘Code is available upon request from th ...,
  pages:  3
}
```

The data was entered by hand (as JSON terms), after reading and reviewing each paper in the survey. In total there are 36 data fields available for documenting papers, but not all need be used for each paper; for example, the field `hasCodeTested` defaults to `false`, so it need not be set — it is also an error to set it if another field asserts there is no code to evaluate! (A separate JSON data structure maps the data fields to English descriptions, along with default values if they are optional descriptors.)

A JavaScript program sanity checks the JSON data. The sanity checks found a few errors (e.g., it checks that if there are comments of any sort then there must be some accessible code in order to have any comments; it checks the DOI is accessible, etc), which led to a productive double-checking of all the facts of the original papers — and correcting all the errors. Some papers that had had no code available during the

first assessment subsequently uploaded code by the time of the double-checking.[21] A field `doubleChecked` was added to supplement the original data field `accessed` to track the process of double-checking the data. Further sanity checks then of course checked all `doubleChecked` fields were completed. And so on. Note that since the checking was done automatically in code, whenever any data was modified and whenever the paper was typeset, the entire body of checks were easily rerun.

Since JSON data is effectively JavaScript code, it was convenient to combine the data, the data sanity checking, and the analysis all in a single JavaScript file for more convenient maintenance. Hence, running the data generates the core human-readable information used in this paper.

The JavaScript data+program generates files from the JSON; these files were then included in both the main paper and in this Supplemental Material, so when the paper or Supplemental Material is typeset all tables and specific data items are typeset automatically, consistently and reliably by LaTeX.

For example, the register `\dataN` is set to the value 32, which is the total number of papers assessed in the JSON data, and the macro `\journalBreakdown` is defined directly from the data to be the following text (when typeset in LaTeX):

> *Lancet Digital Health* ($N = 6$), *Nature Digital Medicine* ($N = 12$) and *Royal Society Open Science* ($N = 14$)

— which is the breakdown of the total $N = 32$ by journal name. The *exact* same text was also used in the main paper.

An interesting consequence of this automatic approach is that as the author found themselves starting to write text such as:

```
Code repositories were used by 10 papers ...
```

it motivated extending the JavaScript data processing so that *all* specific quantities mentioned in the paper are traceable directly back to the JSON data. The phrase above is now in fact written in LaTeX in the paper as follows:

```
Code repositories were used by
\plural{\countUsesVersionControlRepository}{paper} ...
```

where `\plural` automatically writes a word ("paper" in this case) in singular or plural form as required.

Each of the 63 variables used in the paper were defined in automatically-written LaTeX header files that declare them and assigns appropriate values. The header files are included in the paper using LaTeX's standard `\input` command. Here is an example of one such automatic definition:

```
\newcount \dataVariableCount
\dataVariableCount = 63
```

so the named value (here, `dataVariableCount`) is then available for the author to use any way they wish when the paper is typeset.

Some of the files generated from the JSON data are Unix shell scripts. For example, details of all the papers with GitHub repositories are automatically collected into a shell script so the repositories can be cloned locally and then measured (as it happens, using `awk` scripts), e.g., to generate table 9 for this Supplemental Material.

The full JavaScript JSON data and processing code (including the makefile) is provided in this paper's repository, as described in the main paper.

12.d.1 Detecting and defending against error

Normally, when we write a number like 10 in a paper, especially longer or more complex numbers, we will later proof read them as "the numbers we intended to write" — as remembering what we meant is easier than reading the details. Unfortunately, a sentence would likely seem to make as much sense when a number has been erroneously typed as, say, 1.0, 9, 11, or 100 — we hardly bother to pay attention because we think we know what we are reading; at least we know what we meant to write. Worse, the more often we proof read a document, the more we remember, so the better we know what we think we said, and the more casual our proof reading becomes. It is very hard to spot all of our own typos.

- The first and last errors above are examples of the very common error of "out by ten" (common partly because the correct number, 10 looks very similar to 1.0, and 10.0 also looks very similar to 100) [60].

[21] Note that double-checking was performed by the same person as the first assessment, though with the benefit of a long gap to bring a degree of independence.

- The middle two errors above are examples of the common error of "out by one," or "fence post errors" frequently made by mixing up counting fences or the posts (there is usually one more post than fence panel) [60].

All the discussion and examples above were generated automatically, and have been checked correct for other correct values than 10. This approach, too, considerably helps defend against common Human Factors errors. For example, if we set `\countUsesVersionControlRepository`=10 to be 2 348, say, then all of the subsequent sentences that mention it will say something unexpected and so have to be more carefully proof-read, significantly reducing confirmation bias. The approach turns a possibly-hard-to-spot *single* error into *multiple* errors spread throughout the paper into different contexts, thus increasing the chances of noticing the error.

It must be emphasized that an automatically-guaranteed number that is supposed to be the same appearing in multiple different contexts is an extremely effective way of defending against common Human Factors errors. As the number is proof read, the different contexts encourage it to be read more carefully, and in different ways.

If any of the numbers used in a paper were safety critical (e.g., lives directly depend on their values) then further checks would have been made to help detect and avoid errors. LaTeX itself makes it very easy to check that numbers fall within reasonable ranges, or to have any other required safety properties. For the present paper, a potential problem is if the paper is mistakenly typeset *before* the latest JSON data has been analyzed; in which case, none of the variables, like `\countUsesVersionControlRepository`, will have been correctly set and their values could be undefined or nonsense (e.g., from a debugging run of `data.js`).

Although the following text only shows the automatic result of checking final values (and not the calculations that led to them), in general all generated variables can easily be sanity checked in LaTeX or in the generating programs like the present paper's `data.js` code:

> **This is automatic confirmation that** `countUsesVersionControlRepository` $= 10$ **so**
>
> $$5 \leq \texttt{countUsesVersionControlRepository} \leq 20$$
>
> **and therefore** `countUsesVersionControlRepository` **falls within the pre-defined sanity limits set for this paper.**

The corresponding error (or success) messages would not normally be printed in a paper like this — they would normally be reported before a LaTeX run, that is before the paper can be distributed and potentially cause confusion. Note that failing a sanity check indicates a problem that needs to be fixed, but passing a sanity check does not prove a paper correct, but the more sanity checks that are passed (and the harsher those checks) the more confidence we can have that the data has been processed correctly. Of course, when formal methods are employed in the software development process, the confidence in correctness can be very high.

12.d.2 Defending against system problems

Code can become obsolete as programming languages develop and compilers are improved. Typically, compilers first warn that code is "deprecated" and then later versions reject the old code. Furthermore, when code is run on different computers, different operating systems, and with different compilers, it is common to obtain different results. Data, too, is subject to the same problems, but data standards and formats are far more stable than code standards, so "data rot" is less of a risk (but no less a problem when it occurs) than "software rot."

Additionally, errors can be the result of human slips, such as accidentally deleting a line of code or a line of data in a spreadsheet. Such corruption errors are hard to detect unless specific steps are taken to ensure the integrity of code and data [124]. Checksums are the simplest way to detect such errors, but during active research more refined techniques might be used in addition, for example checking that the number of rows of data in a spreadsheet monotonically increases. In the present paper, the JSON data is more structured than a spreadsheet matrix, and a number (as it happens, 30) of other consistency checks are imposed on the data.

To protect against version, portability and other problems, the GitHub repository for the present paper includes a check on software versions and a checksum check for all possibly affected files, including the data file. This does not solve the problem, but it ensures anyone developing or reproducing the paper's work will at least be forewarned of potential version or portability problems. The GitHub repository itself can be used to restore files that have been corrupted.

12.d.3 Problems of restrictive journal policies

Automatically generated variables are used throughout the paper and this Supplemental Material. As usual, LaTeX detects any spelling errors in the use of variables, thus helping protect the paper against typos that could otherwise mislead the paper's readers. Conveniently, LaTeX also supports sophisticated calculations itself [174], so the typeset paper can use any variable values in further calculations without going back to modify the data source file (in the present case, `data.js`). In practice this enables the author to avoid copying-and-pasting values from a data source or calculator, and then overlooking keeping them up to date with changes to the data or formula required.

For example, the caption of table 9 in the main paper calculates its "47 months" figure from the generated variables recording the repository date of cloning used to provide the data to construct the table. The number of months will of course be correctly updated if the paper's repository [182] is subsequently checked again:

> At the time of cloning and checking all repositories in May 2023, paper [182] still had nothing in its repository except a single file still saying "...code coming soon...," despite 47 months having already elapsed since the submitted paper had claimed the code could be accessed in its repository.

Of course, the data generation process itself checks that this surprising statement remains valid, and provides a warning if the wording may need revising.

Unfortunately, although using generated variables and analyses from a paper's data is a very simple technique to help make published papers more reliable, some journals and preprint servers (such as *IEEE Transactions on Software Engineering*, *PLOS ONE*, and *arXiv*) do not permit papers to be submitted using LaTeX source code that uses the standard `\input`, `\bibliography`, and other related commands. Typically they also do not support running any data collection or analysis either (which the present paper does when it clones repositories). These policies undermine the drive towards RAP and RAP+.

Another program (`programs/expand.js`) was therefore written to recursively expand included files so the expanded version can be submitted adhering to any such restrictive policy. Of course, the expanded version now contains all variables as fixed constants, so the submitted paper is misleading and useless to other researchers if the data is modified — the effort to ensure all published numbers are automatically correct is defeated. Such restrictive publishing policies undermine reproducibility.

13 Additional references for Supplemental Material

References numbered 1–129 appear in the reference list in the main paper; the following references are exclusively cited in this Supplemental Material.

[130] Vancouver Group (1997) "Uniform requirements for manuscripts submitted to biomedical journals," *JAMA*, **277**(11):927–934. DOI `10.1001/jama.1997.03540350077040`

[131] Stepney, S. *et al.* (2018) *Engineering Simulations as Scientific Instruments: A Pattern Language*, Springer.

[132] Misselhorn, C. (2018) "Artificial morality. Concepts, issues and challenges," *Social Science and Public Policy*, **55**:161–169. DOI `10.1007/s12115-018-0229-y`

[133] Servick, K. and Enserink, M. (2020) "A mysterious company's coronavirus papers in top medical journals may be unraveling," *Science*. DOI `10.1126/science.abd1337`

[134] Servick, K. (2020) "COVID-19 data scandal prompts tweaks to elite journal's review process," *Science*. DOI `10.1126/science.abe8656`

[135] The Editors (2020) "Learning from a retraction," *The Lancet*, **396**:799. DOI `10.1016/S0140-6736(20)31958-9`

[136] Hardouin, S., Cheng, T. W., Mitchell, E. L., Raulli, S. J., Jones, D. W., Siracuse, J. J. and Farber, A. (2020) "Prevalence of unprofessional social media content among young vascular surgeons," *Journal of Vascular Surgery*, **72**(2):667–671. DOI `10.1016/j.jvs.2019.10.069`

[137] Baumann, J. (29 July, 2020) "#MedBikini backlash exposes research ethics boards' digital gaps," *Bloomberg Law*. URL `news.bloomberglaw.com/pharma-and-life-sciences/medbikini-backlash-exposes-research-ethics-boards-digital-gaps`

[138] The Editors (of *Journal of Vascular Surgery*) (2020) "Editors' Statement Regarding "Prevalence of unprofessional social media content among young vascular surgeons"," *FaceBook*. URL `www.facebook.com/TheJVascSurg/photos/a.611986142331744/1381024778761206/?type=3&theater`, and copied to journal home page URL `www.jvascsurg.org` Accessed 30 July 2020.

[139] Bourque, P. and Fairley, R., eds. (2014) *Guide to the Software Engineering Body of Knowledge*, (Version 3.0), IEEE Computer Society. URL `www.swebok.org`

[140] Zeller, A. (2006) *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann.

[141] Humphrey, W. S. (2005) *PSP: A Self-Improvement Process for Software Engineers*, Addison Wesley.

[142] —— (2001) *Winning with Software: An Executive Strategy*, Addison-Wesley Professional.

[143] Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G. and Weis, P. (2014) "Trusting computations: A mechanized proof from partial differential equations to actual program," *Computers and Mathematics with Applications*, **68**:325–352. DOI `10.1016/j.camwa.2014.06.004`

[144] Thimbleby, H. and Cairns, P. (2017) "Interactive numerals," *Royal Society Open Science*, **4**(4):160903. DOI `10.1098/rsos.160903`

[145] Thimbleby, H. (2012) "Heedless programming: Ignoring detectable error is a widespread hazard," *Software — Practice & Experience*, **42**(11):1393–1407. DOI `10.1002/spe.1141`

[146] Chao, D. L., Halloran, M. E., Obenchain, V. J. and Longini Jr, I. M. (2010) "FluTE, a publicly available stochastic influenza epidemic simulation model," *PLOS Computational Biology*, **6**(1):e1000656. Source code available at URL `GitHub.com/dlchao/FluTE`, DOI `10.1371/journal.pcbi.1000656`

[147] Abir, M., Nelson, C., Chan, E. W., Al-Ibrahim, H., Cutter, C., Patel, K. and Bogart, A. (2020) *RAND Critical Care Surge Response Tool: An Excel-Based Model for Helping Hospitals Respond to the COVID-19 Crisis*, RAND Corporation. URL `www.rand.org/pubs/tools/TLA164-1.html`

[148] Alvarez, N. M., Gonzalez-Gonzalez, E. and Trujillo-de Santiago, G. (2020) "Modeling COVID-19 epidemics in an excel spreadsheet: Democratizing the access to first-hand accurate predictions of epidemic outbreaks," *MedRxiv*. Preprint, DOI `10.1101/2020.03.23.20041590`

[149] Schlueter, I. Z. (23 March 2016) *Blog: kik, left-pad, and npm*, NPM Blog. Accessed 10 April 2020, URL `blog.npmjs.org/post/141577284765/kik-left-pad-and-npm`

[150] Shneiderman, B., Plaisant, C., Cohen, M. and *et al* (2016) *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Pearson, 6th ed. URL `www.cs.umd.edu/hcil/DTUI6`

[151] Thimbleby, H. (2007) *Press On: Principles of Interaction Programming*, MIT Press.

[152] Hanson, C. and Sussman, G. J. (2021) *Software design for flexibility: How to avoid programming yourself into a corner*, MIT Press.

[153] ISO/IEC JTC 1/SC 7 Software and systems engineering Committees (2015) *Software engineering — Guidelines for the application of ISO 9001:2015 to computer software*, International Organization for Standardization (ISO). URL `www.iso.org/standard/74348.html`

[154] RTCA Committee SC-205 (2011) *DO-178C — Software Considerations in Airborne Systems and Equipment Certification*, RTCA. URL `my.rtca.org/NC__Product?id=a1B36000001IcmqEAC`

[155] Baum, T., Leßmann, H. and Schneider, K. (2017) "The choice of code review process: A survey on the state of the practice," *in Lecture Notes in Computer Science*, vol. **10611** of Product-Focused Software Process Improvement: 18th International Conference, pp. 111–127. DOI `10.1007/978-3-319-69926-4_9`

[156] Brett, A., Croucher, M., Haines, R., Hettrick, S., Hetherington, J., Stillwell, M. and Wyatt, C. (2017) *State of the Nation Report for Research Software Engineers*, Research Software Engineer Network. URL `zenodo.org/record/495360#.Xyfuyi2ZOCM`, DOI `10.5281/zenodo.495360`

[157] Software Sustainability Institute (2020) *Web site*. Accessed 3 August 2020, URL `software.ac.uk`

[158] Anderson, R. (2020) *Security Engineering*, Wiley, 3rd ed.

[159] Shostack, A. and Zurko, M. E. (2020) "Secure development tools and techniques need more research that will increase their impact and effectiveness in practice," *Communications of the ACM*, **63**(5):39–41. DOI `10.1145/3386908`

[160] Schneier, B. (2018) *Click Here To Kill Everybody — Security and Survival in a Hyper-connected World*, W. W. Norton & Company, Inc, New York, United States.

[161] Mason, S. and Seng, D. (2017) *Electronic Evidence*, Humanities Digital Library, 4th ed. (NB See URL `humanities-digital-library.org/index.php/hdl/catalog/book/electronicevidence` until DOI is resolved), DOI `10.14296/517.9781911507079`

[162] Simon, H. A. and Laird, J. (2019) *Sciences of the Artificial*, MIT Press, 3rd ed.

[163] Kruger, J. and Dunning, D. (1999) "Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments," *Journal of Personality and Social Psychology*, **77**(6):1121–1134. DOI 10.1037/0022-3514.77.6.1121

[164] Nuhfer, E., Fleisher, S., Cogan, C., Wirth, K. and Gaze, E. (2017) "How random noise and a graphical convention subverted behavioral scientists' explanations of self-assessment data: Numeracy underlies better alternatives," *Numeracy*, **10**(1):4. DOI 10.5038/1936-4660.10.1.4

[165] Thimbleby, H. (2016) "Human Factors and missed solutions to Enigma design weaknesses," *Cryptologia*, **40**(2):177–202. DOI 10.1080/01611194.2015.1028680

[166] Kerr, N. L. (1998) "HARKing: Hypothesizing after the results are known," *Personality and Social Psychology Review*, **2**(3):196–217. DOI 10.1207/s15327957pspr0203_4

[167] Thimbleby, H., Anderson, S. O. and Cairns, P. (1999) "A framework for modelling Trojans and computer virus infection," *Computer Journal*, **41**(7):444–458. DOI 10.1093/comjnl/41.7.444

[168] Strachey, C. (2000) "Fundamental concepts in programming languages," *Higher-Order and Symbolic Computation*, **13**:11–49. DOI 10.1023/A:1010000313106

[169] Henderson, P. (1980) *Functional Programming: Application and Implementation*, Prentice-Hall International.

[170] Mehra, M. R., Desai, S. S., Ruschitzka, F. and Patel, A. N. (2020) "RETRACTED: Hydroxychloroquine or chloroquine with or without a macrolide for treatment of COVID-19: A multinational registry analysis," *The Lancet*, **online**:1–10. DOI 10.1016/S0140-6736(20)31180-6

[171] Zhang, Y., Masci, P., Jones, P. and Thimbleby, H. (2019) "User interface software errors in medical devices," *Biomedical Instrumentation & Technology*, **53**(3):182–194. DOI 10.2345/0899-8205-53.3.182

[172] Salama, M., Bahsoon, R. and Lago, P. (2021) "Stability in software engineering: Survey of the state-of-the-art and research directions," *IEEE Transactions on Software Engineering*, **47**(7):1468–1510. DOI 10.1109/TSE.2019.2925616

[173] Hurst, P. (2022) *Data sharing and mining*, Royal Society. DOI 10.25504/FAIRsharing.dIDAzV

[174] Fuster, R. (2012) "The `calculator` and `calculus` packages: Arithmetic and functional calculations inside LaTeX," *TUGboat*, **33**(3):265–271.

14 References for surveyed papers

[175] A. Callahan, E. Steinberg, J. A. Fries, S. Gombar, B. Patel, C. K. Corbin, and N. H. Shah, "Estimating the efficacy of symptom-based screening for COVID-19," *Nature Digital Medicine*, **3**(95):3pp, 2020. DOI 10.1038/s41746-020-0300-0
Accessed 14 July 2020. Double-checked 17 January 2021.

[176] C. M. Kanzler, M. D. Rinderknecht, A. Schwarz, I. Lamers, C. Gagnon, J. P. O. Held, P. Feys, A. R. Luft, R. Gassert, and O. Lambercy, "A data-driven framework for selecting and validating digital health metrics: use-case in neurological sensorimotor impairments," *Nature Digital Medicine*, **3**(80):17pp, 2020. DOI 10.1038/s41746-020-0286-7 Code URL github.com/ChristophKanzler/MetricSelectionFramework
Accessed 14 July 2020. Double-checked 17 January 2021.

[177] A. Roy, K. Nikolitch, R. McGinn, S. Jinah, W. Klement, and Z. A. Kaminsky, "A machine learning approach predicts future risk to suicidal ideation from social media data," *Nature Digital Medicine*, **3**(78):12pp, 2020. DOI 10.1038/s41746-020-0287-6
Accessed 14 July 2020. Double-checked 17 January 2021.

[178] D. M. Levine, Z. Co, L. P. Newmark, A. R. Groisser, A. J. Holmgren, J. A. Haas, and D. W. Bates, "Design and testing of a mobile health application rating tool.," *Nature Digital Medicine*, **3**(74):7pp, 2020. DOI 10.1038/s41746-020-0268-9
Accessed 14 July 2020. Double-checked 17 January 2021.

[179] T. Kannampallil, J. M. Smyth, S. Jones, P. R. O. Payne, and J. Ma, "Cognitive plausibility in voice-based AI health counselors," *Nature Digital Medicine*, **3**(72):4pp, 2020. DOI 10.1038/s41746-020-0278-7
Accessed 14 July 2020. Double-checked 17 January 2021.

[180] S. Huang, T. Kothari, I. Banerjee, C. Chute, R. L. Ball, N. Borus, A. Huang, B. N. Patel, P. Rajpurkar, J. Irvin, J. Dunnmon, J. Bledsoe, K. Shpanskaya, A. Dhaliwal, R. Zamanian, A. Y. Ng, and M. P. Lungren, "PENet a scalable deep-learning model for automated diagnosis of pulmonary embolism using volumetric CT imaging," *Nature Digital Medicine*, **3**(61):9pp, 2020. DOI 10.1038/s41746-020-0266-y Code URL github.com/marshuang80/PENet
Accessed 14 July 2020. Double-checked 17 January 2021.

[181] S. S. Dhruva, J. S. Ross, J. G. Akar, B. Caldwell, K. Childers, W. Chow, L. Ciaccio, P. Coplan, J. Dong, H. J. Dykhoff, S. Johnston, T. Kellogg, C. Long, P. A. Noseworthy, K. Roberts, A. Saha, A. Yoo, and N. D. Shah, "Aggregating multiple real-world data sources using a patient-centered health-data-sharing platform," *Nature Digital Medicine*, **3**(60):9pp, 2020. DOI 10.1038/s41746-020-0265-z
Accessed 14 July 2020. Double-checked 17 January 2021.

[182] I. S. Hofer, C. Lee, E. Gabel, P. Baldi, and M. Cannesson, "Development and validation of a deep neural network model to predict postoperative mortality, acute kidney injury, and reintubation using a single feature set," *Nature Digital Medicine*, **3**(58):10pp, 2020. DOI 10.1038/s41746-020-0248-0 Code URL github.com/cklee219/PostoperativeOutcomes_RiskNet
Accessed 14 July 2020. Double-checked 17 January 2021.

[183] B. Norgeot, K. Muenzen, T. A. Peterson, X. Fan, B. S. Glicksberg, G. Schenk, E. Rutenberg, B. Oskotsky, M. Sirota, J. Yazdany, G. Schmajuk, D. Ludwig, T. Goldstein, and A. J. Butte, "Protected Health Information filter (Philter): accurately and securely de-identifying free-text clinical notes," *Nature Digital Medicine*, **3**(57):8pp, 2020. DOI 10.1038/s41746-020-0258-y Code URL github.com/BCHSI/philter-ucsf
Accessed 14 July 2020. Double-checked 17 January 2021.

[184] D. Choi, J. J. Park, T. Ali, and S. Lee, "Artificial intelligence for the diagnosis of heart failure," *Nature Digital Medicine*, **3**(54):6pp, 2020. DOI 10.1038/s41746-020-0261-3 Code URL github.com/ubiquitous-computing-lab/AI-CDSS-Cardiovascular-Silo
Accessed 14 July 2020. Double-checked 17 January 2021.

[185] C. B. Hilton, A. Milinovich, C. Felix, N. Vakharia, T. Crone, C. Donovan, A. Proctor, and A. Nazha, "Personalized predictions of patient outcomes during and after hospitalization using artificial intelligence," *Nature Digital Medicine*, **3**(51):8pp, 2020. DOI 10.1038/s41746-020-0249-z
Accessed 14 July 2020. Double-checked 17 January 2021.

[186] M. D. Li, K. Chang, B. Bearce, B. Y. Chang, A. J. Huang, J. P. Campbell, J. M. Brown, P. Singh, K. V. Hoebel, D. Erdoğmuş, S. Ioannidis, W. Palmer, M. F. Chiang, and J. Kalpathy-Cramer, "Siamese neural networks for continuous disease severity evaluation and change detection in medical imaging," *Nature Digital Medicine*, **3**(48):9pp, 2020. DOI 10.1038/s41746-020-0255-1 Code URL github.com/QTIM-Lab/SiameseChange
Accessed 14 July 2020. Double-checked 19 January 2021.

[187] J. I. Hoffman, R. Nagel, V. Litzke, D. A. Wells, and W. Amos, "Genetic analysis of *Boletus edulis* suggests that intra-specific competition may reduce local genetic diversity as a woodland ages," *Royal Society Open Science*, **7**(200419):13pp, 2020. DOI 10.1098/rsos.200419 Code URL datadryad.org/stash/dataset/doi:10.5061/dryad.1g1jwstrw
Accessed 22 July 2020. Double-checked 26 January 2021.

[188] P. Grönquist, P. Panchadcharam, D. Wood, A. Menges, M. Rüggeberg, and F. K. Wittel, "Computational analysis of hygromorphic self-shaping wood gridshell structures," *Royal Society Open Science*, **7**(192210):9pp, 2020. DOI 10.1098/rsos.192210 Code URL royalsocietypublishing.org/doi/suppl/10.1098/rsos.192210
Accessed 22 July 2020. Double-checked 26 January 2021.

[189] W. Amos, "Signals interpreted as archaic introgression appear to be driven primarily by faster evolution in Africa," *Royal Society Open Science*, **7**(191900):9pp, 2020. DOI 10.1098/rsos.191900 Code URL datadryad.org/stash/share/ichHKrWj7hqlznOaR6NQVzITgp4OdlqWvWAgAxyafiQ
Accessed 22 July 2020. Double-checked 26 January 2021.

[190] M. Gordon, D. Viganola, M. Bishop, Y. Chen, A. Dreber, B. Goldfedder, F. Holzmeister, M. Johannesson, Y. Liu, C. Twardy, J. Wang, and T. Pfeiffer, "Are replication rates the same across academic fields? Community forecasts from the DARPA SCORE programme," *Royal Society Open Science*, **7**(200566):7pp, 2020. DOI 10.1098/rsos.200566 Code URL royalsocietypublishing.org/doi/suppl/10.1098/rsos.200566
Accessed 22 July 2020. Double-checked 26 January 2021.

[191] D. Evans, and A. P. Field, "Predictors of mathematical attainment trajectories across the primary-to-secondary education transition: parental factors and the home environment," *Royal Society Open Science*, **7**(200422):20pp, 2020. DOI 10.1098/rsos.200422 Code URL osf.io/a5xsz/?view_only=87ae173f775b40d79d6cd0fdcf6d4a9c
Accessed 22 July 2020. Double-checked 26 January 2021.

[192] N. Beale, H. Battey, A. C. Davison, and R. S. MacKay, "An unethical optimization principle," *Royal Society Open Science*, **7**(200462):11pp, 2020. DOI 10.1098/rsos.200462
Accessed 22 July 2020. Double-checked 26 January 2021.

[193] A. A. Cherevko, T. S. Gologush, I. A. Petrenko, V. V. Ostapenko, and V. A. Panarin, "Modelling of the arteriovenous malformation embolization optimal scenario," *Royal Society Open Science*, **7**(191992):16pp, 2020. DOI 10.1098/rsos.191992
Accessed 22 July 2020. Double-checked 26 January 2021.

[194] A. A. Soczawa-Stronczyk, and M. Bocian, "Gait coordination in overground walking with a virtual reality avatar," *Royal Society Open Science*, **7**(200622):19pp, 2020. DOI 10.1098/rsos.200622 Code URL datadryad.org/stash/dataset/doi:10.5061/dryad.vx0k6djnr
Accessed 22 July 2020. Double-checked 26 January 2021.

[195] S. Duruz, E. Vajana, A. Burren, C. Flury, and S. Joost, "Big dairy data to unravel effects of environmental, physiological and morphological factors on milk production of mountain-pastured Braunvieh cows," *Royal Society Open Science*, **7**(200638):13pp, 2020. DOI 10.1098/rsos.200638 Code URL github.com/SolangeD/lactModel
Accessed 22 July 2020. Double-checked 26 January 2021.

[196] E. Z. D. de Azevedo, D. V. Dantas, and F. G. Daura-Jorge, "Risk tolerance and control perception in a game-theoretic bioeconomic model for small-scale fisheries," *Royal Society Open Science*, **7**(200621):11pp, 2020. DOI 10.1098/rsos.200621
Accessed 22 July 2020. Double-checked 26 January 2021.

[197] A. M. Abdolhosseini-Qomi, S. H. Jafari, A. Taghizadeh, N. Yazdani, M. Asadpour, and M. Rahgozar, "Link prediction in real-world multiplex networks via layer reconstruction method," *Royal Society Open Science*, **7**(191928):22pp, 2020. DOI 10.1098/rsos.191928 Code URL github.com/UT-NSG/LRM
Accessed 22 July 2020. Double-checked 26 January 2021.

[198] J. Webster, and M. Amos, "A Turing test for crowds," *Royal Society Open Science*, **7**(200307):12pp, 2020. DOI 10.1098/rsos.200307 Code URL figshare.com/collections/Supplementary_information_for_Webster_J_and_Amos_M_A_Turing_Test_for_Crowds_/4859118/1
Accessed 22 July 2020. Double-checked 26 January 2021.

[199] Y-l. Zhu, C-J. Wang, F. Gao, Z-x. Xiao, P-l. Zhao, and J-y. Wang, "Calculation on surface energy and electronic properties of CoS_2," *Royal Society Open Science*, **7**(191653):12pp, 2020. DOI 10.1098/rsos.191653
Accessed 22 July 2020. Double-checked 26 January 2021.

[200] B. Yu, C. J. Scott, X. Xue, X. Yue, and X. Dou, "Derivation of global ionospheric Sporadic E critical frequency (f_oEs) data from the amplitude variations in GPS/GNSS radio occultations," *Royal Society Open Science*, **7**(200320):15pp, 2020. DOI 10.1098/rsos.200320
Accessed 22 July 2020. Double-checked 26 January 2021.

[201] K. Joon-myoung, C. Younghoon, J. Ki-Hyun, C. Soohyun, K. Kyung-Hee, B. Seung D, J. Soomin, P. Jinsik, and O. Byung-Hee, "A deep learning algorithm to detect anaemia with ECGs: a retrospective, multicentre study," *Lancet Digital Health*, **2**(7):9pp, 2020. DOI 10.1016/S2589-7500(20)30108-4
Accessed 24 July 2020. Double-checked 26 January 2021.

[202] H. Zhu, C. Cheng, H. Yin, X. Li, P. Zuo, J. Ding, F. Lin, J. Wang, B. Zhou, Y. Li, S. Hu, Y. Xiong, B. Wang, G. Wan, X. Yang, and Y. Yuan, "Automatic multilabel electrocardiogram diagnosis of heart rhythm or conduction abnormalities with deep learning: a cohort study," *Lancet Digital Health*, **2**(7):9pp, 2020. DOI 10.1016/S2589-7500(20)30107-2
Accessed 24 July 2020. Double-checked 26 January 2021.

[203] R. Fung, J. Villar, A. Dashti, L. C. Ismail, E. Staines-Urias, E. O. Ohuma, L. J. Salomon, C. G. Victora, F. C. Barros, A. Lambert, M. Carvalho, A. Jaffer Y, J. A. Noble, M. G. Gravett, M. Purwar, R. Pang, E. Bertino, S. Munim, A. M. Min, R. McGready, S. A. Norris, Z. A. Bhutta, S. H. Kennedy, A. T. Papageorghiou, and A. Ourmazd, "Achieving accurate estimates of fetal gestational age and personalised predictions of fetal growth based on data from an international prospective cohort study: a population-based machine learning study," *Lancet Digital Health*, **2**(7):7pp, 2020. DOI 10.1016/S2589-7500(20)30131-X Code URL github.com/ki-analysis/manifold-ga
Accessed 24 July 2020. Double-checked 26 January 2021.

[204] C. Sabanayagam, D. Xu, D. S. W. Ting, S. Nusinovici, R. Banu, H. Hamzah, C. Lim, Y-C. Tham, C. Y. Cheung, E. S. Tai, X. Y. Wang, J. B. Jonas, C-Y. Cheng, M. L. Lee, W. Hsu, and T. Y. Wong, "A deep learning algorithm to detect chronic kidney disease from retinal photographs in community-based populations," *Lancet Digital Health*, **2**(7):7pp, 2020. DOI 10.1016/S2589-7500(20)30063-7
Accessed 24 July 2020. Double-checked 26 January 2021.

[205] M. Monteiro, V. F. Newcombe, F. Mathieu, K. Adatia, K. Kamnitsas, E. Ferrante, T. Das, D. Whitehouse, D. Rueckert, D. K. Menon, and B. Glocker, "Multiclass semantic segmentation and quantification of traumatic brain injury lesions on head CT using deep learning: an algorithm development and multicentre validation study," *Lancet Digital Health*, **2**(7):8pp, 2020. DOI 10.1016/S2589-7500(20)30085-6 Code URL github.com/biomedia-mira/blast-ct
Accessed 24 July 2020. Double-checked 27 January 2021.

[206] K-L. Liu, T. Wu, P-T. Chen, M. Tsai Y, H. Roth, M-S. Wu, W-C. Liao, and W. Wang, "Deep learning to distinguish pancreatic cancer tissue from non-cancerous pancreatic tissue: a retrospective study with cross-racial external validation," *Lancet Digital Health*, **2**(7):10pp, 2020. DOI 10.1016/S2589-7500(20)30078-9
Accessed 24 July 2020. Double-checked 27 January 2021.