

Navigating the development challenges in creating complex data systems

Received: 11 July 2022

Accepted: 25 April 2023

Published online: 1 June 2023



Sören Dittmer ^{1,2,20}✉, Michael Roberts ^{1,3,20}✉, Julian Gilbey ¹,
Ander Biguri ¹, AIX-COVNET Collaboration*, Jacobus Preller ⁴,
James H. F. Rudd ³, John A. D. Aston⁵ & Carola-Bibiane Schönlieb¹

Data science systems (DSSs) are a fundamental tool in many areas of research and are now being developed by people with a myriad of backgrounds. This is coupled with a crisis in the reproducibility of such DSSs, despite the wide availability of powerful tools for data science and machine learning over the past decade. We believe that perverse incentives and a lack of widespread software engineering skills are among the many causes of this crisis and analyse why software engineering and building large complex systems is, in general, hard. Based on these insights, we identify how software engineering addresses those difficulties and how one might apply and generalize software engineering methods to make DSSs more fit for purpose. We advocate two key development philosophies: one should incrementally grow—not plan then build—DSSs, and one should use two types of feedback loop during development—one that tests the code's correctness and another that evaluates the code's efficacy.

Machine learning (ML) is in a reproducibility crisis^{1–3}. We argue that a primary driver is poor code quality, which has two root causes: poor incentives to produce good code and a widespread lack of software engineering skills. This crisis also suggests that data science systems (DSS) can, and will, fail silently if no continual verification infrastructure exists throughout their development^{4–6}.

Modern DSSs are extremely complex systems with many components, many themselves intrinsically susceptible to minor changes in the data or underlying code. Three key communities are involved in developing successful DSSs: researchers, professional software engineers and their incentivisers. The last is the community that encourages those developing DSSs to produce useful results, for example, research supervisors, employers, funding agencies and journal editors. These communities can work separately or jointly, but all are trained and rewarded differently.

Although researchers designing and implementing DSSs in academia and industry will be aware of the particular complexities of data in their own domain, they usually lack formal training in software

development methodologies. Therefore, the DSS is often planned and then built solely to satisfy the objectives and incentives of the researcher. Software engineers, however, are specifically trained to develop complex systems and incentivized to make them flexible, organized and maintainable.

Gall's law⁷ states that complex systems cannot be built—they can only be grown—and software development methodologies such as Agile implicitly acknowledge that one cannot plan complex systems. Rather, one has to evolve and grow them (following Gall's law⁷). A critical requirement in software development is the development of a minimum viable product. For data pipelines, this is often called a steel thread⁸, bootstrapping a stable path that one can gradually extend to build a more complete pipeline. Effective utilization of feedback loops and repeated testing allows one to assess the code's correctness without deviating from a working codebase.

The current crisis is a natural consequence of an environment in which data scientists develop complex DSSs without growing them methodically or establishing a careful and continual assessment of

¹Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, UK. ²ZeTeM, University of Bremen, Bremen, Germany.

³Department of Medicine, University of Cambridge, Cambridge, UK. ⁴Addenbrooke's Hospital, Cambridge University Hospitals NHS Trust, Cambridge, UK.

⁵Department of Pure Mathematics and Mathematical Statistics, University of Cambridge, Cambridge, UK. ²⁰These authors contributed equally: Sören Dittmer, Michael Roberts. *A list of authors and their affiliations appears at the end of the paper. ✉e-mail: sd870@cam.ac.uk; mr808@cam.ac.uk

their code's correctness. Although incorrect code can be computationally reproducible—that is, rerunning the code produces identical results—for replicability and general reproducibility using independent implementations, correctness is crucial. Even more crucially, reusability—standing on the shoulders of giants—demands correctness. Indeed, without it, every downstream task inherits the lack of correctness and reproducibility.

Owing to the pervasiveness of ML, the community of people developing DSSs is now vast and diverse. Therefore, in this Perspective, we identify specific challenges for those developing DSSs and their incentivisers and recommend corrective actions to improve DSS reusability and reproducibility. We also present a development philosophy, formalizing and adapting software development methodologies for DSS researchers.

The core problems

Data × code = complexity²

Until relatively recently, statisticians had a monopoly on data analysis. They were, and are, highly trained to appreciate the intricate relationships and biases in data and use relatively simple methods (in the best sense of the word) to analyse and fit models to that data. Data collection was often done under their guidance to ensure biases were understood, documented and mitigated. Nowadays, data are ubiquitous and heralded as the 'new oil'. However, real-world datasets often resemble more of an oil spill, containing a plethora of unknown (and often unknowable) biases⁹.

DSS developers must be tolerant of the complexities in the data and code, along with any due to their interaction. Although software engineering masters the complexities of code, combining code with data stacks complexity on top of complexity. Thus, constructing a DSS can resemble balancing a stick on top of a stick. Consequently, without sufficient statistical and software engineering skills, the development of a DSS tends to lead to the following implications:

Big Data ⇒ Messy Data ⇒ Big Code
⇒ Messy Code ⇒ Incorrect Conclusions

A Cambrian explosion in data and codebases

The radical increase in the scale of data and the wide availability of ML tools have also led to an equally radical paradigm shift in their use. From the data and codebase perspectives, this paradigm shift resembles a Cambrian explosion in quantity and intrinsic complexity. Hence, DSS researchers are consumers of many more software tools than classical statisticians.

As a user of many tools, focusing on how to interface with them (rather than gaining a deep understanding of their internal workings) becomes an uncomfortable necessity. The underlying software must therefore be trustworthy. One has to assume it is almost bug-free, with any remaining bugs being insignificant.

The shift makes expressing and structuring an analysis plan in code the bedrock for all data science projects. However, software engineering is a challenging discipline, and building on vast unfamiliar codebases often leads to unexpected consequences.

Why is the problem challenging?

This section will discuss some substantial challenges, both technical and human-driven, that data scientists face when developing correct and effective DSSs.

Challenge 1: DSS researchers lacking software engineering and software development skills

Most data scientists only learn to write small codebases, whereas software development focuses on creating interconnected modules and components, each of which is an isolated component of a much larger

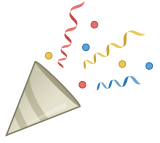
	Not correct	Correct
Not efficacious	You do not know whether your idea is bad. Try to achieve correctness, it might give you efficacy too.	You need a new idea.
Efficacious	You do not know whether your idea works. Try to make the system correct or analyse why your system is effective.	

Fig. 1 | Consequences of a code that is or is not correct and is or is not efficacious. In summary, a successful DSS results only from a code that is both correct and efficacious.

codebase. Code is the interface to many data science tools, and software engineering is the discipline of organizing interfaces methodically. In this Perspective, we define software engineering as the discipline of managing the complexity of code and data, with interfaces as one of its primary tools¹⁰. Although many software development practices focus on enterprise software and do not trivially apply to all components of DSSs, it is our belief that software development methodologies should play a more prominent role in data science projects.

Challenge 2: correctness and efficacy

A DSS must work correctly, that is, do what you think it does. It also must be efficacious, that is, produce relevant and usable predictions. Without software engineering, following earlier arguments, this tends to lead to the following implications:

Multiple Experiments ⇒
Messy Code ⇒ Incorrect Conclusions

So, why do we need correctness and efficacy for a trustworthy highly performing model? First, although a published and executable codebase can provide computational reproducibility, repeatability requires correctness. Second, although an incorrect DSS can be efficacious owing to a lucky bug, it is uninterpretable and hard to modify. Without correctness, it is impossible to understand, interpret or trust the outputs of a DSS or conclusions based on it (Fig. 1).

Challenge 3: perverse incentives for researchers

Software engineers are rewarded for creating highly performing, well-documented and reusable codebases, while industrial data scientists are rewarded based on a DSS's usefulness to a business. Research data scientists work within a very different incentive structure. They are incentivized to use the outputs of their DSS to write novel papers to further their field, apply for grants and to enhance their reputation and career prospects. For researchers, this creates a temporal conflict. In the short term, publishing papers quickly and giving less attention to the reusability of the codebase is rewarding. However, in the long term, reusable DSSs increase the probability that the associated paper will become influential and well-cited. This perverse incentivization may even discourage producing and publishing code comprehensible to a broad audience to avoid getting 'scooped' by competitors.

If the field's incentive structure and goals are misaligned, the path of least resistance easily wins out. Incentivisers must acknowledge whether their incentives are perverse to their long-term ambitions and take corrective action if so. In particular, journals must be conscious that the need for manuscript novelty can lead to researchers manufacturing complexity in DSSs to increase the likelihood of acceptance.

Challenge 4: short circuits

The wide availability of powerful data analysis and ML tools allows for short circuits, as keen amateurs can quickly develop complex DSSs. This is not to say that using powerful, publicly available tools or short circuits is inherently bad. On the contrary, if every practitioner wrote private versions of common toolkits, this would be a major source of bugs. However, powerful tools reduce the accidental complexity, not the intrinsic complexity of DSSs. Thus, they make the building of complex systems with a high intrinsic complexity easier. This intrinsic complexity is extremely hard to manage, especially as it often hides in subtleties.

Challenge 5: teams versus individual work

Working in a team on a codebase can be extremely powerful. However, without the proper training or organizational structure, it can also produce massive inefficiencies and errors—teams being complex systems themselves. Software engineers are often highly trained in methodologies encouraging effective team working, for example, SCRUM^{11,12}. They also know how to harness the benefits of infrastructure, such as version control, continuous integration pipelines and pair programming. Researchers tend to only possess informal training in these teamwork-enabling tools, and can even be actively discouraged from teamwork to ensure individual contributions are clear.

Challenge 6: bridging the academia–industry gap

Data science projects in industry and academia have many similarities. However, besides the already discussed incentive differences, there are also key differences in the DSS development environment. Owing to a larger software engineering culture, industry embraces the idea that high-quality code is obligatory for maintainable DSSs; academic researchers are usually not incentivized along these lines because of the nature of their short-term projects. In academia, there are many benefits, in particular the freedom to explore completely new ideas, but the incentives promote a strong throwaway mentality towards code, and academia has virtually no feedback loop for code quality. High-quality code is neither a prerequisite for most publications nor used to assess job performance.

Challenge 7: training a DSS is costly

A change in a DSS can require costly and lengthy retraining to check whether or how it changes the outcome. For this reason, seemingly minor fixes, improvements and code cleanups might not happen at all.

Challenge 8: long-term maintenance

Even a small DSS is often sufficiently complex that the number of package dependencies can easily number in the dozens. As complex systems are inherently fragile, a minor change in one of the dependencies can lead to a (potentially silent) failure of the entire DSS. This is one of many reasons why long-term code maintenance is costly or impossible. Although there are many countermeasures to facilitate computation reproducibility, for example, publishing Python/Anaconda environments and test suites, they do not ensure future reusability within a larger DSS.

Summary of the challenges

Many researchers have a systemic lack of awareness that software engineering is integral to modern data science. This results from a lack of formal training and a perverse incentive structure, both of which cause a colossal loss of opportunity to create value. DSSs must be both correct and efficacious. The potential unleashed by the usability of modern data science tools has enabled substantial progress. However, it has also led to the development of many seemingly efficacious but incorrect systems. Industry is inherently better at developing high-quality code as it must integrate with infrastructure, teams and deployment platforms. Academia lacks such guard rails, and code development can be myopic.

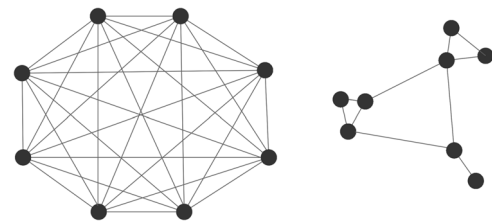


Fig. 2 | Visualization of bad and good software architectures. The graph on the left is a fully connected graph illustrating a bad software architecture. The sparsely, mostly locally, connected graph on the right is a better architecture^{28,29}.

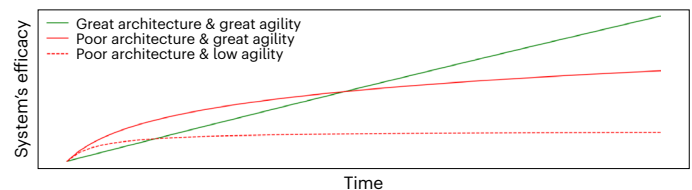


Fig. 3 | Visualization of Agile development. Some software engineering practices require delayed gratification, where one has to sacrifice short-term progress for long-term progress.

Using software development methodologies to grow complex systems

Every programmer can write small codebases, but larger ones require software development methodologies to perform correctly and be maintainable¹³. So why is it generally hard to build complex systems from scratch?

Complex systems tend to consist of many highly interconnected components that are susceptible to small perturbations. In the case of a codebase, these perturbations could be simple typos that, with luck, produce a syntax error. On the other hand, a simple typo can subtly alter the outcome in unknown ways, leading to dramatic and unexpected consequences.

Instead, one should use small incremental steps, never deviating far from a working system. Gall's law, stating that complex systems must be grown and not built, should be of great value to data scientists as we argue that growing an n -component system can reduce the worst-case build complexity from $O(n^2)$ to $O(n)$.

Although one can often decompose complex systems into predominantly simple components, their sheer number and interactions quickly produce a complex whole. If one wants to build an n -component system, there are up to $O(n^2)$ interactions, giving $O(n^2)$ potential failure points (assuming that each component works correctly). Software engineering has developed two leading solutions to this ' $O(n^2)$ problem': software architecture and the Agile development methodology^{13,14}.

Software architecture

Well-established code development principles are a critical component of software development. One fundamental principle is the separation of concerns, which splits the software into different components, each handling a single isolated concern and possessing a simple, complexity-hiding interface¹⁰. These components are, in turn, formed by connecting lower-level isolated components. Designing the software architecture in this manner reduces the graph spanned by the different components from a potentially densely connected graph with $O(n^2)$ connections to a sparse graph with far fewer potential failure points. It is also advantageous to have a sense of locality in the code and graph such that components are preferably locally connected. Figure 2 provides a visualization.

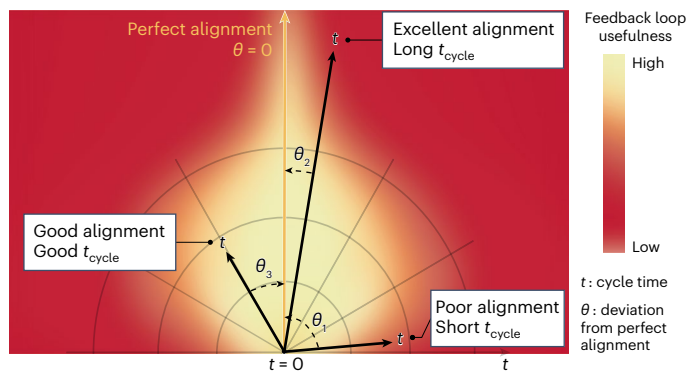


Fig. 4 | Illustration of how the usefulness of feedback loops depends on their alignment θ and cycle time t . For example, we identify (t_1, θ_1) : a integration test that ensures the code can be run without syntactic errors which can take milliseconds; (t_2, θ_2) : retraining a model from scratch on a large database and evaluating it on an independent test set, which can take several hours or days; (t_3, θ_3) : checking for outlier values in the features of the training data, which can take milliseconds.

Agile development

Modern software development tends to follow Agile methodologies, where one grows software incrementally, adding or changing one component at a time, so that there is always a working system. One only has to consider how this new component interacts with the existing n components. This reduces the $O(n^2)$ potential failure points to $O(n)$ possible failure points at each step. In the end, this reduces the ‘build complexity’ from $O(n^2)$ to $O(n)$ when the complex system is grown. Figure 3 provides a visualization.

New habits for DSS development

Now we discuss combining and generalizing these principles into actionable advice for data scientists. Following these suggestions will allow them to write correct and effective code in less time.

Do not build complex systems, but grow them

We have discussed how growing a DSS can reduce the build complexity from $O(n^2)$ to $O(n)$. Planning an entire system and then building it does not work for complex systems. We must grow DSSs to keep the complexity on the order of at most $O(n)$ at each incremental stage, ideally—enabled by good software architecture—of $O(1)$.

Planning is still required to ensure that we continually grow our systems towards the desired goal, but it should be highly iterative and alternate with incremental implementation steps. Planning not only orients the iterations but also helps to avoid local optima during the evolution of the DSS. It is valuable to recognize that the future evolution of a complex system is increasingly fuzzy. Planning should follow a multiscale approach with a discount factor on future details. An excellent example is the comparison of SpaceX’s rocket development process against the classical approach^{15–17}. The rocket’s design was grown by testing it over many iteratively adapted instantiations, each being less of a failure than the last. This iterative process, embracing the inevitability of errors, must become a deeply appreciated fact during the development of a DSS¹⁸ and of complex systems in general.

The nature and necessity of feedback loops

The power of feedback loops makes incremental, iterative development incredibly effective. When establishing a feedback loop, it is helpful to consider its two properties: alignment and cycle time (Fig. 4).

For alignment we consider how many assumptions about a given code component are measured by the feedback loop? Aligning the feedback loop with our goal for the code is crucial. If the alignment

is suboptimal, the feedback loop cannot give us confidence in the trustworthiness of the component.

For the cycle time we consider how much time (or cost) is required to get the feedback? Even if it were possible, a 100%-aligned feedback loop would be useless if it took an unreasonable amount of time to run. Ideally, we want a short cycle time to allow for high-frequency feedback.

Writing a test suite is extremely powerful for establishing a feedback loop on code. Each test’s execution provides a feedback signal on a particular aspect of the code. Responding to the signal of a test suite with high alignment and low cycle time (running quickly and with high frequency) establishes a strong feedback loop. Reading the code also provides a feedback signal, whose strength is defined by the code’s readability.

As discussed previously, data scientists should care about both the model’s efficacy and the code’s correctness. We therefore need feedback loops that measure both. We measure the DSS’s efficacy by evaluating our model on a test set. We measure its correctness (or trustworthiness) with a test suite and by making the code as readable as possible.

Software architecture for data science systems

Good software architecture reduces the number of components connected to any incremental addition to the system, reducing the system’s build complexity. Good architecture also dramatically improves the code’s readability and future-proofs the codebase’s flexibility¹⁹. We argue that the crucial architecture concept for DSSs is the idea of horizontal layers, as shown in Fig. 5.

Horizontal software layers are the different components of an analysis pipeline, for example, data loading, preprocessing, model training and evaluation. We can view each layer as a high-level component in the software. Accordingly, intra- opposed to interconnections to other layers should dominate. Each layer is a pocket of complexity, hiding its complexity from the other components in the system.

Feature and model engineering are two of the most important tasks in ML, and we can interpret both as asking questions about the data. The ultimate question we often ask is ‘how well can I predict some labels with given features, particular preprocessing and a specific model?’ Answering this requires coding the whole pipeline to establish efficacy feedback. This lengthy wait is antithetical to the Agile approach.

We thus recommend organizing the pipeline in horizontal layers and building each layer in a minimalistic incomplete fashion so that the basic connections between the layers are established early in the project (Fig. 5). We think this point is crucial to quickly establishing a tight, highly aligned feedback loop. Without this feedback loop, even feature preprocessing becomes a potential ‘fishing expedition’, as you cannot know if it improves the outcome.

Using a simple method like linear regression first is a good idea when fitting a predictive model. It is often claimed that you should do this to avoid overfitting. However, the key benefit is that linear regression is easy to implement and fast to run, enabling you to rapidly establish the first feedback loop with a short cycle time. This steel thread⁸ (minimum viable product in SE) allows for iteration and building complexity while never deviating too far from a working codebase.

Testing for data science systems

Developing a suite of tests (for example, unit, integration and end-to-end) for both the codebase and the datasets, while developing a DSS, provides a comprehensive correctness feedback loop.

Code tests. A good test suite mitigates the fragility that DSSs—as complex systems—inevitably have. For example, a typo can destroy everything without ever being noticed. However, software engineers often base tests on example input–output pairs, but knowing these for non-trivial numerical code may be impossible.

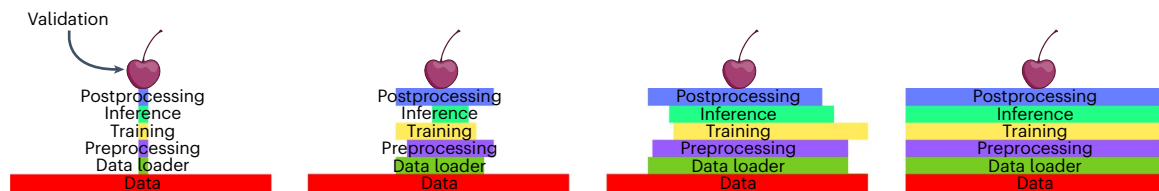


Fig. 5 | When growing a DSS, you must be able to support the cherry on top as early as possible. This ‘steel thread’ is necessary to establish an efficacy feedback loop (the cherry).

Property-based testing can help. Although correct outputs of a numerical function might be hard to know in advance, we often know the mathematical properties the function should satisfy. Property-based testing uses that knowledge, for example, by creating random inputs and checking whether the property holds for them. Libraries, such as `pytest`²⁰ and `hypothesis`²¹, can be utilized for general and property-based testing, respectively.

One critical question is ‘which tests do I have to write to be confident in the correctness of my system?’ We propose that focusing on the functionality the system must provide when deployed is key. This allows for recursive thinking about which system components one must test, and to what degree, to have confidence in the system’s functionality when deployed externally.

DSSs should not just churn data, but should also highlight issues and violated assumptions to the developer. ML code in general, and deep learning code in particular, often fail silently in many unexpected ways⁴. For example, one probably would not notice if early layers of a convolutional neural network’s architecture were buggy and their weights not updated during training.

Data tests. We do not know what the data knows. This is another reason why DSSs can fail silently. Therefore, implementing tests not only for code but also for data is crucial. It is also crucial to plot the data as often as possible. Looking at plots is a feedback loop with high alignment. However, looking at plots is time-consuming; that is, this has a high cycle time. We recommend extracting the relevant information from what you see in the plots and writing tests based on that²².

Within the code, we recommend performing as many checks on assumptions about the data as possible. Although you should do this repeatedly, probably the most crucial point is immediately before the data go into a model⁴. Checking our assumptions on the data is hard, as we are often unaware of them and may forget which ones we made, so it makes sense to hard-code them with tests whenever we notice them. One can also do this with dedicated Python libraries like `pandera`²³. If we expect a variable to be in a particular format, we should write a check that generates an error or warning in case the format is wrong. Tests minimize uncertainties by converting assumptions into certainties.

We ask the data questions by running experiments. Like in a good conversation, you must listen to the answers carefully and adapt your future responses and questions accordingly. That does not mean your question generator algorithm has to be greedy, but it has to be iterative. On the one hand, iterative work unlocks the power of feedback loops, which we need when working with complex/real-world data. On the other hand, this requires agility in how you interact with the data.

Conclusion

Feedback loops are a prerequisite for feature engineering, model development and everything else. Feedback loops allow one to move faster, further, and more confidently. Growing DSSs incrementally harnesses the power of feedback loops.

Correctness and efficacy are different things that require different feedback loops. The most critical feedback loops for correctness are

writing and running a test suite and writing as comprehensible code as possible. The most important point for building a feedback loop for efficacy is establishing it early by growing the entire data pipeline as early and thinly as possible.

We note that (almost) no feedback loop is perfectly aligned; still, they are essential. However, we warn that a subtle problem can arise when iterating on misaligned feedback loops. Overfitting, also known as Goodhart’s law^{24,25}, states that every measure that becomes a target ceases to be a good measure. Overfitting is predominantly a problem for efficacy feedback loops.

As discussed in ref. 26, people and processes optimizing perverse incentives with misaligned feedback can lead them to (un)consciously ‘play the system’. This overfitting, that is, on the validation set, can happen to the entire DSS, not just the model. Although researchers are usually aware of this problem when training a model, they are often unaware of it for the entire DSS. The same countermeasures to model overfitting apply to DSSs, for example, holdout test sets not utilized during the development.

Also, we reemphasize that this is a socio-technical problem. Training of students and early career researchers in these specific issues is essential; see, for example, *The Turing Way*²⁷. Also, despite endeavours like Zenodo or SoftwareX, academia often directs the incentive structures away from creating and publishing high-quality DSSs. We must, therefore, improve the incentives structure alignment in academia.

References

- Haibe-Kains, B. et al. Transparency and reproducibility in artificial intelligence. *Nature* **586**, E14–E16 (2020).
- Pineau, J. et al. Improving reproducibility in machine learning research: a report from the neurIPS 2019 reproducibility program. *J. Mach. Learn. Res.* **22**, 7459–7478 (2021).
- Baker, M. 1,500 scientists lift the lid on reproducibility. *Nature* **533**, 452–454 (2016).
- Karpathy, A. *A Recipe for Training Neural Networks*; <https://karpathy.github.io/2019/04/25/recipe/> (2019).
- Aboumatar, H. & Wise, R. A. Notice of retraction. Aboumatar et al. Effect of a program combining transitional care and long-term self-management support on outcomes of hospitalized patients with chronic obstructive pulmonary disease: a randomized clinical trial. *JAMA*. 2018;320(22):2335–2343. *JAMA* **322**, 1417–1418 (2019).
- Bhandari Neupane, J. et al. Characterization of leptazolines A-D, polar oxazolines from the *Cyanobacterium leptolyngbya* sp., reveals a glitch with the ‘Willoughby-Hoye’ scripts for calculating NMR chemical shifts. *Org. Lett.* **21**, 8449–8453 (2019).
- Gall, J. *General Systemantics* (General Systemantics Press, 1975).
- Brabban, P., Case, S., Cutts, S., Diniz, C. & Crawford, L. *Data Pipeline Playbook*; <https://data-pipeline.playbook.ee/> (2021).
- Roberts, M. et al. Common pitfalls and recommendations for using machine learning to detect and prognosticate for COVID-19 using chest radiographs and CT scans. *Nat. Mach. Intell.* **3**, 199–217 (2021).

10. Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**, 1053–1058 (1972).
11. Sutherland, J. & Sutherland, J. V. *Scrum: The Art of Doing Twice the Work in Half the Time* (Currency, 2014).
12. Fowler, M. & Highsmith, J. et al. The Agile manifesto. *Software Dev.* **9**, 28–35 (2001).
13. Farley, D. *Modern Software Engineering: Doing What Works to Build Better Software Faster* (Addison-Wesley, 2021).
14. Bass, L., Clements, P. & Kazman, R. *Software Architecture in Practice* (Addison-Wesley, 2003).
15. Reddy, V. S. The SpaceX effect. *New Space* **6**, 125–134 (2018).
16. Vance, A. & Sanders, F. *Elon Musk* (Harper Collins, 2015).
17. Smith, R. J. Shuttle problems compromise space program: with the shuttle earth-bound, political troubles and cost overruns take off. *Science* **206**, 910–914 (1979).
18. Perkel, J. M. How to fix your scientific coding errors. *Nature* **602**, 172–173 (2022).
19. Lakshmanan, V., Robinson, S. & Munn, M. *Machine Learning Design Patterns* (O'Reilly Media, 2020).
20. Krekel, H. et al. Pytest x.y.; <https://github.com/pytest-dev/pytest> (2004).
21. MacIver, D. R. Hypothesis x.y.; <https://github.com/HypothesisWorks/hypothesis-python> (2016).
22. Baumgartner, P. Ways I Use Testing as a Data Scientist <https://www.peterbaumgartner.com/blog/testing-for-data-science/> (2021).
23. Niels, B. pandera: statistical data validation of pandas dataframes. In *Proc. 19th Python in Science Conference* (eds Agarwal, M. et al.) 116–124 (2020).
24. Goodhart, C. A. in *Monetary Theory and Practice* 91–121 (Springer, 1984).
25. Hoskin, K. in *Accountability: Power, Ethos and the Technologies of Managing* (eds Munro, R. & Mouritsen, J.) 265 (Cengage Learning EMEA, 1996).
26. Muller, J. Z. in *The Tyranny of Metrics* (Princeton Univ. Press, 2019).
27. The Turing Way Community. *The Turing Way: A Handbook for Reproducible, Ethical and Collaborative Research* 1.0.1 (Alan Turing Institute, 2021).
28. Watts, D. J. & Strogatz, S. H. Collective dynamics of 'small-world' networks. *Nature* **393**, 440–442 (1998).
29. Valverde, S. & Solé, R. V. Hierarchical small worlds in software architecture. Preprint at <https://arxiv.org/abs/cond-mat/0307278> (2003).

Acknowledgements

We are grateful to the EU/EFPIA Innovative Medicines Initiative project DRAGON (101005122; S.D. and M.R., AIX-COVNET, C.-B.S.), Trinity Challenge BloodCounts! project (M.R., J.G. and C.-B.S.), EPSRC Cambridge Mathematics of Information in Healthcare Hub EP/T017961/1 (M.R., J.H.F.R., J.A.D.A. and C.-B.S.), Cantab Capital Institute for the Mathematics of Information (C.-B.S.), the European Research Council for Horizon 2020 grant no. 777826 (C.-B.S.), the Alan Turing Institute (C.-B.S.), the Wellcome Trust (J.H.F.R.), Cancer Research UK Cambridge Centre (C9685/A25177; C.-B.S.), the British Heart Foundation (J.H.F.R.), NIHR Cambridge Biomedical Research Centre (J.H.F.R.), HEFCE (J.H.F.R.), Leverhulme Trust project on 'Breaking the non-convexity barrier' (C.-B.S.), the Philip Leverhulme Prize (C.-B.S.), EPSRC grants EP/S026045/1 and EP/T003553/1 (C.-B.S.) and the Wellcome Innovator Award RG98755 (C.-B.S.). We are also grateful to Intel for financial support, I. Selby for creative input, and J.-C. Lohmann, S. Griffith, J. Tang and F. Zhang for comments and discussions.

Competing interests

The authors declare no competing interests.

Additional information

Correspondence should be addressed to Sören Dittmer or Michael Roberts.

Peer review information *Nature Machine Intelligence* thanks Ben MacArthur and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

© Springer Nature Limited 2023

AIX-COVNET Collaboration

Michael Roberts^{1,3,20}, Sören Dittmer^{1,2,20}, Ian Selby⁶, Anna Breger^{1,7}, Matthew Thorpe⁸, Julian Gilbey¹, Jonathan R. Weir-McCall^{6,9}, Effrossyni Gkrania-Klotsas⁴, Anna Korhonen¹⁰, Emily Jefferson¹¹, Georg Langs¹², Guang Yang¹³, Helmut Prosch¹², Jacobus Preller⁴, Jan Stanczuk¹, Jing Tang¹⁴, Judith Babar⁴, Lorena Escudero Sánchez⁶, Philip Teare¹⁵, Mishal Patel^{15,16}, Marcel Wassin¹⁷, Markus Holzer¹⁷, Nicholas Walton¹⁸, Pietro Lió¹⁹, Tolou Shadbahr¹⁴, James H. F. Rudd³, John A. D. Aston⁵, Evis Sala⁶ & Carola-Bibiane Schönlieb¹

⁶Department of Radiology, University of Cambridge, Cambridge, UK. ⁷Faculty of Mathematics, University of Vienna, Vienna, Austria. ⁸Department of Mathematics, University of Manchester, Manchester, UK. ⁹Royal Papworth Hospital, Cambridge, UK. ¹⁰Language Technology Laboratory, University of Cambridge, Cambridge, UK. ¹¹Population Health and Genomics, School of Medicine, University of Dundee, Dundee, UK. ¹²Department of Biomedical Imaging and Image-guided Therapy, Medical University of Vienna, Vienna, Austria. ¹³National Heart and Lung Institute, Imperial College London, London, UK. ¹⁴Research Program in Systems Oncology, Faculty of Medicine, University of Helsinki, Helsinki, Finland. ¹⁵Data Science & Artificial Intelligence, AstraZeneca, Cambridge, UK. ¹⁶Clinical Pharmacology & Safety Sciences, AstraZeneca, Cambridge, UK. ¹⁷contextflow GmbH, Wien, Austria. ¹⁸Institute of Astronomy, University of Cambridge, Cambridge, UK. ¹⁹Department of Computer Science and Technology, University of Cambridge, Cambridge, UK.