Can Software Engineering Harness the Benefits of Advanced AI?

Mary Shaw and Liming Zhu

Artificial intelligence (AI) has allowed us to build systems beyond anything deemed possible earlier. Can we evolve existing techniques in software engineering to meet the needs of AI enabled systems or do we need to build unique and novel tools to do so?

Point: Bringing Engineering to Machine Learning

WE ARE SAID to be in the midst of an "artificial intelligence (AI) revolution" driven by systems that incorporate machine learning (ML) algorithms that draw inferences from very large datasets. Some folks argue that ML systems are so vastly different from conventional software systems that we need a brand new software discipline to cope with them.

On the contrary, software engineering regularly responds to new types of software opportunities by refining our portfolio of established methods. We should do likewise for ML, as we have done for other ideas from AI through the years.

AI has long been a wellspring and incubator for new software ideas. These ideas may initially seem challenging, even impractical, but some of them eventually acquire respectability and enter the mainstream—whereupon their roots in AI are largely forgotten. Our repertoire of software techniques is richer for the influences from AI.

Look at just a few of the ideas that have made their way from AI to mainstream software. AI brought us the view of design as problem solving, often eclipsing the complementary view of design as problem setting. List processing originated in the AI community well over half a century ago, introducing dynamic list structures and functional programming. Search strategies were an AI mainstay years before search became a major industry.

AI researchers were among the first to use exploratory programming in support of their attempts to write programs for particular tasks that were viewed as requiring "intelligence." The production systems they used to implement expert systems were precursors of modern publish–subscribe systems.

What about the current offering from AI to mainstream software systems? The oft-voiced concerns about systems that incorporate ML[1,2,3] involve, first and foremost, the dominant role of data and their curation. In addition, there are concerns about correctness and nondeterminism rooted in opacity, the lack of specifications, correctness, dynamic change, nondeterminism, and abstraction. Real-world software components have always been subject to these concerns, but ML components do so with a vengeance. We need to develop the abstractions that provide intellectual control over systems that incorporate ML components.

AI has regularly brought new ideas and challenges to software engineering.

Digital Object Identifier 10.1109/MS.2022.3203200
Date of current version: 24 October 2022

Software engineering has just as regularly drawn on its repertoire of engineering techniques for innovative responses that bring these ideas to stable practice. We can confidently expect software engineering will do the same for ML.

Large Datasets

ML systems rely on the automated collection, classification, cleansing, versioning, and analysis of very large quantities of data harvested in the wild. Moreover, the datasets are regularly updated, and their models are regularly rederived (relearned). The inferences from these models depend on the quality of the data as well as the quality of the learning algorithms.

However, ordinary database systems also have large datasets, often with rich, highly structured data, that are managed separately from the code and for which data quality is often a major issue. The administration required for ML datasets is different from the administration required for databases, but both cases require system management of data as well as code.

Data quality is vulnerable to many threats, from ill-formed data to well-formed but erroneous data and biased or malicious data; this is a long-standing problem in conventional systems as well. The problem is exacerbated in ML systems by the exploratory nature of model development.

Nondeterminism

ML systems are regarded as unpredictable and unreliable in a number of ways. Their algorithms are opaque, and they can't explain how their outputs are related to their inputs. Moreover, their behavior can change without notice as they are retrained, and hidden dependencies and model interactions add to the unpredictability. However, conventional components are also unexplainable in practice, third-party components also change without notice, and feature interactions have been studied for decades. The apparent nondeterminism of ML components is similar to that found in ordinary complex software systems with ill-specified input, especially cyberphysical systems.

The principles of reliable software allow us to develop reliable systems from nondeterministic components by using techniques such as modularization and architectural firewalls, redundancy, runtime checks and feedback, version control, checkpoints, checkable assertions, prioritization of failure severity, fault tolerance, graceful degradation, stochastic modeling, and so on. These localize or compensate for the vagaries of nondeterministic or unreliable components, provide intermediate sanity checks, define the envelope of allowable performance, allow the analysis of field variability, and indeed use ML techniques to detect anomalies in the system.[4] Even if these techniques do not apply unchanged to ML systems, they offer a robust and well-understood starting point.

We need to develop the abstractions that provide intellectual control over systems that incorporate ML components.

Correctness

ML components are intended to model the phenomena that generated the input data. If we already had precise models of those phenomena, we wouldn't need to use ML. Therefore, the results will necessarily be judged against criteria other than precise specifications. However, this is also true of many systems in the real world, especially sociotechnical systems—they have weak, incorrect, or incomplete specifications.

Absent precise specifications, ML systems may be judged on the extent to which they deliver results that reasonably match expectations. However, a similar situation occurs in conventional software, when fitness for task is accepted as the standard of evaluation.

More seriously, developers of ML systems are properly concerned about biased outcomes—that is, models that might accurately capture the data but do not match societal expectations. However, this situation is just what Rittel and Webber[5] described half a century ago when they introduced the concept of "wicked problems."

Appropriate Abstractions

The integration of ML in software systems requires the development of abstractions appropriate to the domain. What are the principal types of components, and do we think of them as algorithms, models, datasets, functions, or something else? What architecture for the system provides intellectual control over data curation, training, and invoking the resulting models, which occur at different times in the operation of the software system? What are appropriate abstractions for handling uncertainty, nondeterminism, and opacity with the right level of detail? Extending our portfolio of abstractions is the bread and butter of software engineering, so this will be a natural evolution.

Hence, as I reasoned in my keynote talk for the Fourth History of Programming Languages conference,[6] I argue here that ML is the latest in a long history of challenges to the established norms of software development. The issues in ML resemble issues the software field has addressed before. Rather than treating ML as a novel challenge signaling an AI revolution that invalidates existing software principles and processes, we should build on software engineering's established approaches to address the distinctive characteristics of ML. The claim that ML is essentially different from other software doesn't stand up.

We should take a cue from Shapiro and Varian's analysis[7] of the information economy: recognize and understand the new technology, but ask how the old principles, perhaps with new parameters, still apply. The unruliness of ML components may be greater than that of more conventional components, but the techniques we've developed for conventional software can be extended to handle them.

Finally, ML will not be the last challenge to software engineering: quantum computing is waiting in the wings to demand new refinements of software engineering's portfolio of models and techniques.

Acknowledgment

Support from the Alan J. Perlis Chair of Computer Science at Carnegie Mellon University allowed me to write the reflections on programming languages for the Fourth History of Programming Languages[6] that provided the basis for this essay. I thank the many colleagues who helped me refine those thoughts.

—*Mary Shaw*

Counterpoint : AI Engineering—The Revolution Is Now

WHY ARE AI systems so difficult to engineer and risky to adopt? I believe this is due to the unique characteristics of AI (both data-driven ML and good old-fashioned AI) presented as new challenges to the traditional software engineering approaches. These new challenges cannot be tackled only with extensions of existing methods. Instead, we need new AI engineering methods.

These new challenges cannot be tackled only with extensions of existing methods.

On a fundamental level, AI solves problems autonomously with seemingly effective solutions that humans may not fully comprehend. AI can even propose new problems to be solved. This paradigm is radically different from the paradigm where humans identify and solve problems via software built by approaches and tools they reasonably understand. Most existing software engineering methods are geared toward the latter paradigm.

Start with requirements. Although the software engineering world is moving from a requirement-driven prescriptive approach to a data-driven explorative approach,[8] the requirements are still being discovered and specified by some input/output expectations by humans. In contrast, the requirements in ML-driven approaches are essentially hidden in the training data. Rather than uncovering the implicit requirements to be confirmed by humans, ML methods only discover a version of the implementation—the learned models. These learned models may seem consistent with the training data but are often inscrutable to humans. Together with the missing explicit requirements, these lead to concerns about value alignment and unintended consequences. Even more concerning, AI could discover that manipulating and changing human requirements is far easier than finding a solution to satisfy them. A classic example is that a recommender engine manipulates humans into liking extreme content, thus making future predictions easier.

Next, look at quality. Traditional software engineering has an impressive array of methods to explicitly build quality into software, verified/validated against an explicit set of nonfunctional requirements and supported by design rationales. This approach is being extended to ethical AI, where we attempt to imbue AI systems with prescriptive human values and ethics. These by-design approaches are not working very well for AI-generated black-box solutions, where you have almost no meaningful places to inject quality control. This is almost inevitable when we hope for the best by merely providing AI training datasets and high-level optimization goals—a hodgepodge of human biases, follies, ingenuity, and underspecified aims.

The unique characteristics of AI systems have manifested in many other ways to confront the existing software engineering methods. System degradation caused by latent data shifts is harder to discover and fix than typical requirements evolution. Reproducibility in continual learning is still a science challenge rather than engineering/data hygiene and data quality. How do we apply quality-by-design approaches and control tactics when AI designs solutions for us that we do not fully comprehend? The big red kill switch is more of a hopeless desperation than a comforting solution.

Nevertheless, I am not in despair at all. I believe software engineers, perhaps this time with more powerful AI-enabled tools, will find answers. It requires us to think beyond the existing paradigm. The new generation of AI engineering methods should possess attributes that help achieve the following:

- Define outcomes and guardrails in contrast to the traditional approaches of specifying problems, requirements, and specifications. AIs solve problems autonomously for us and sometimes even propose new problems to be solved. We need new methods that engineers can use to engage with diverse users and communities to better select problems, define outcomes, and specify more inclusive goals and guardrails for AI. We can leave the rest to AI but with better-ensured bounds.
- Understand AI-generated solutions rather than set or solve problems. We need new methods and (AI) tools that software engineers can use to interpret, explain, and simplify cryptic AI-based solutions with justified trust. More importantly, these methods should go beyond the algorithmic level to cover end-to-end lifecycles and help governance and communication with nonexpert stakeholders.[9]
- Embed radical observability and monitorability for governance, not just for error detection and diagnosis. With impenetrable solutions and latent/implicit requirements to check against, we need new approaches to observability and monitorability, especially for postdeployment of high-risk AI systems. We need new methods for "meaningful" human intervention rather than humans being symbolic liability sponges. We need new methods to monitor the quality of data input for AI, which is different from traditional rule-based data validation and data quality. Finally, we may have to invent new approaches to studying and controlling mysterious machine behaviors from the outside, just like how social/behavior scientists study humans and societies without a complete understanding of the human brain.

To help invent this new generation of AI engineering methods, AI itself may also help in radical ways. AIs are already writing code, explaining code/models to human engineers, and monitoring other AI systems. AI-generated ethical tradeoffs are assisting humans in clarifying and improving ethical requirements, which stem from our black-box brains. However, even more captivating for us is the realization that these new AI engineering methods will also play a critical role in understanding our own intelligence. Software engineers build software systems no longer just to satisfy human needs but to understand human intelligence. It's a brave new world, and we need brave new methods.

Acknowledgment
I would like to thank Dr. Mark Staples and Dr. Chen Wang from CSIRO's Data61, and Daphne Li for their substantial input into the piece.

—Liming Zhu

Response of Mary Shaw

Software engineering has an established body of knowledge, including concepts, theories, and methods as well as specific implementation techniques. These are concerned with the design and integration of entire software systems as well as the development of individual components of these systems. Software engineering calls for a designerly mindset that selects appropriate structures and methods, recognizing when innovative design is required and when routine, precedented design will suffice. Software engineering's body of knowledge has evolved as new types of systems have emerged—from the original simple ones comprising link-edited single-threaded code components to parallel and distributed systems, interactive systems with sophisticated user interfaces, cloud-based web systems, compositions of services, cyberphysical systems, and so on.

Individual components of these systems can be of many kinds. Most familiar are the code components produced by compiling code in imperative or functional programming languages. However, components can also be databases, constraint sets, dynamic data feeds, output from preprocessors, and otherwise distant from programming languages. The most recent additions to this portfolio are the models produced by ML—sophisticated inference from large datasets. As such, they are clearly candidates for innovative rather than routine engineering.

Different types of components call for different accommodations in integration, especially if their update cycles are automated in a way that does not incorporate changes in their behavior in the analysis of the systems in which they are used. This sort of automated update occurs commonly now, challenging the traditional notion that the manager of the software controls which versions of which components are actually in use; the resulting software might be better described as a "coalition of multisourced components" rather than as a "system." ML calls for innovation in this area as well, for example to handle the large bodies of data, retraining, and model updates.

For many years, AI systems were research systems, with engineering expectations of rapid prototyping and evolution but less commitment to long-term maintenance and correctness for all use cases. They have gradually been becoming components in business systems, but only relatively recently have they had a major presence in systems that conspicuously affect the general public. Now, they face the engineering expectations of production systems, including robustness, correctness (or at least predictability), and long-term support. As AI and ML enter the mainstream, the systems or coalitions in which they are embedded must also.

Liming Zhu and I do fully agree that software systems should be expected to satisfy societal standards for qualities such as fairness and reliability. Although a narrow view of an ML model might focus on internal consistency, it should meet a higher standard of behaving in the public interest before it's released to the public. We also agree on the need for safeguards to keep the results of these systems within the "guardrails."

ABOUT THE AUTHORS

MARY SHAW is the Alan J. Perlis University Professor of Computer Science at Carnegie Mellon University, Pittsburgh, PA 15217 USA. Her research focuses on software engineering, particularly software architecture and the design of systems used by real people. She is a Fellow of ACM, IEEE, and the AAAS. Contact her at mary.shaw@cs.cmu.edu.

LIMING ZHU is a research director at CSIRO's Data61 and a conjoint full professor at the University of New South Wales, Sydney NSW 2015, Australia. He is the chairperson of Standards Australia's blockchain and distributed ledger committee and on AI trustworthiness related ISO committees. Contact him at liming.zhu@data61.csiro.au.

My view, though, is that these problems arise in conventional systems as well.

So I say again: AI and ML bring new kinds of components to software systems, and these are still unfamiliar and challenging. However, software engineering has a rich set of concepts and theories that are broader than the specific solutions offered to familiar systems. These provide a robust starting point for innovation that addresses the new opportunities that AI brings. Why start from scratch, seeking brand new engineering methods for AI components, when we can innovate from a base of established experience in evolving existing software engineering knowledge?

Response of Liming Zhu

Prof. Shaw's insightful piece made some excellent arguments. I agree that extending existing approaches plays an important role in tackling some of the challenges posed by AI. However, the fact that some challenges shared between non-AI and AI software were identified decades ago or that AI has inspired many software engineering approaches should not distract us from the reality that we are still grappling with these challenges today. The fact that they have not been solved adds weight to the imperative for new thinking. The challenge, perhaps, should not be framed as only a mere matter of the degree of difficulty or new kinds of components (be they data, cloud, mobile, or AI), which certainly lends itself to extending existing approaches for some valuable results.

I still think the challenge should be seen from the role that software engineers play, hitherto primarily in setting and solving problems. AI systems can propose problems to solve and solve problems for us, often with a seemingly good solution beyond our full comprehension. Methods for understanding inscrutable AI systems are akin more to science than engineering, while noting that the underlying nature of AI systems may turn out to be more complex than that of laws of physics and comparable to that of biological and social systems. All of these are bound to change the way we engineer AI systems and extend our new engineering prowess to the realm of science and vice versa.

References

1. A. Horneman, A. Melinger, and I. Ozkaya, "AI engineering: 11 foundational practices," Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA, White Paper, 2019. Accessed: Dec. 4, 2021. [Online]. Available: https://resources.sei.cmu.edu/asset_files/WhitePaper/2019_019_001_634648.pdf
2. C. Kästner and E. Kang, "Teaching software engineering for AI-enabled systems," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., Softw. Eng. Educ. Training (ICSE-SEET 20)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 45–48, doi: 10.1145/3377814.3381714.
3. I. Ozkaya, "What is really different in engineering AI-enabled systems?" *IEEE Softw.*, vol. 37, no. 4, pp. 3–6, Jul./Aug. 2020, doi: 10.1109/MS.2020.2993662.
4. M. R. Lyu, *Handbook of Software Reliability Engineering*. Piscataway, NJ, USA: IEEE Press, 1996. [Online]. Available: http://www.cse.cuhk.edu.hk/~lyu/book/reliability/index.html
5. H. W. J. Rittel and M. M. Webber, "Dilemmas in a general theory of planning," *Policy Sci.*, vol. 4, no. 2, pp. 155–169, 1973, doi: 10.1007/BF01405730.
6. M. Shaw, "Myths and mythconceptions: What does it mean to be a programming language, anyhow?" *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, p. 44, 2022, doi: 10.1145/3480947.
7. C. Shapiro and H. R. Varian, *Information Rules: A Strategic Guide to the Network Economy*. Boston, MA, USA: Harvard Business School Press, 1999.
8. J. Bosch, and H. Holmström Olsson, and I. Crnkovic, "Engineering AI Systems: A Research Agenda," in *Artificial Intelligence Paradigms for Smart Cyber-Physical Systems*, A. K. Luhach and A. Elçi, Eds. Hershey, PA, USA: IGI Global, 2021, pp. 1–19.
9. Q. Lu, L. Zhu, X. Xu, J. Whittle, and Z. Xing, "Towards a roadmap on software engineering for responsible AI," in *Proc. 1st Int. Conf. AI Eng. – Softw. Eng. AI (CAIN)*, 2022, pp. 101–112.

IT Professional
TECHNOLOGY SOLUTIONS FOR THE ENTERPRISE

CALL FOR ARTICLES

IT Professional seeks original submissions on technology solutions for the enterprise. Topics include

- emerging technologies,
- cloud computing,
- Web 2.0 and services,
- cybersecurity,
- mobile computing,
- green IT,
- RFID,
- social software,
- data management and mining,
- systems integration,
- communication networks,
- datacenter operations,
- IT asset management, and
- health information technology.

We welcome articles accompanied by web-based demos. For more information, see our author guidelines at www.computer.org/itpro/author.htm.

WWW.COMPUTER.ORG/ITPRO

IEEE COMPUTER SOCIETY IEEE

Digital Object Identifier 10.1109/MS.2022.3210318