ARCHITECTING THE FUTURE OF SOFTWARE ENGINEERING

A National Agenda for Software Engineering Research & Development

Carnegie Mellon University
Software Engineering Institute

Software is vital to our country's global competitiveness, innovation, and national security. It also ensures our modern standard of living and enables continued advances in defense, infrastructure, healthcare, commerce, education, and entertainment. As part of its work as a federally funded research and development center (FFRDC) focused on applied research to improve the practice of software engineering, the Carnegie Mellon University Software Engineering Institute led the community in creating this multi-year research and development vision and roadmap for engineering next-generation software-reliant systems.

Authors and Contributors

Lead Author Team

Anita Carleton, Study Lead
Director, Software Solutions Division (SSD), Carnegie Mellon University (CMU) Software Engineering Institute (SEI)

Mark Klein
Principal Technical Advisor and Principal Researcher, SSD, CMU SEI

John Robert
Deputy Director, SSD, CMU SEI

Erin Harper
Strategic Communications Manager, SSD, CMU SEI

Additional Authors (in alphabetical order)

Rob Cunningham
Vice Chancellor for Research Infrastructure, University of Pittsburgh

Dio De Niz
Technical Director, Assuring Cyber-Physical Systems, SSD, CMU SEI

Ed Desautels
Senior Technical Writer & Content Strategist, CMU SEI

John Foreman
SEI Fellow and Principal Engineer, SSD, CMU SEI

John Goodenough
SEI Fellow, Principal Researcher, Director's Office, CMU SEI

James Herbsleb
Director and Professor, Institute for Software Research, CMU

Charles Holland
Principal Researcher, SSD, CMU SEI

Ipek Ozkaya
Technical Director, Engineering Intelligent Software Systems, SSD, CMU SEI

Doug Schmidt
Cornelius Vanderbilt Professor of Engineering, Vanderbilt University

Forrest Shull
Lead for Defense Software Acquisition Policy Research, SSD, CMU SEI

Advisory Board

Deb Frincke, Chair
Associate Laboratory Director for National Security Sciences, Oak Ridge National Laboratory

Sara Manning Dawson
Chief Technology Officer, Enterprise Security, Microsoft

Jeff Dexter
Senior Director of Flight Software & Cybersecurity, SPACEX

Yolanda Gil
Director of Knowledge Technologies, Information Sciences Institute, University of Southern California

Vint Cerf
Vice President and Chief Internet Evangelist, Google

Penny Compton
Vice President for Software Systems, Cyber, and Operations, Lockheed Martin Space

Tim McBride
President, Zoic Labs

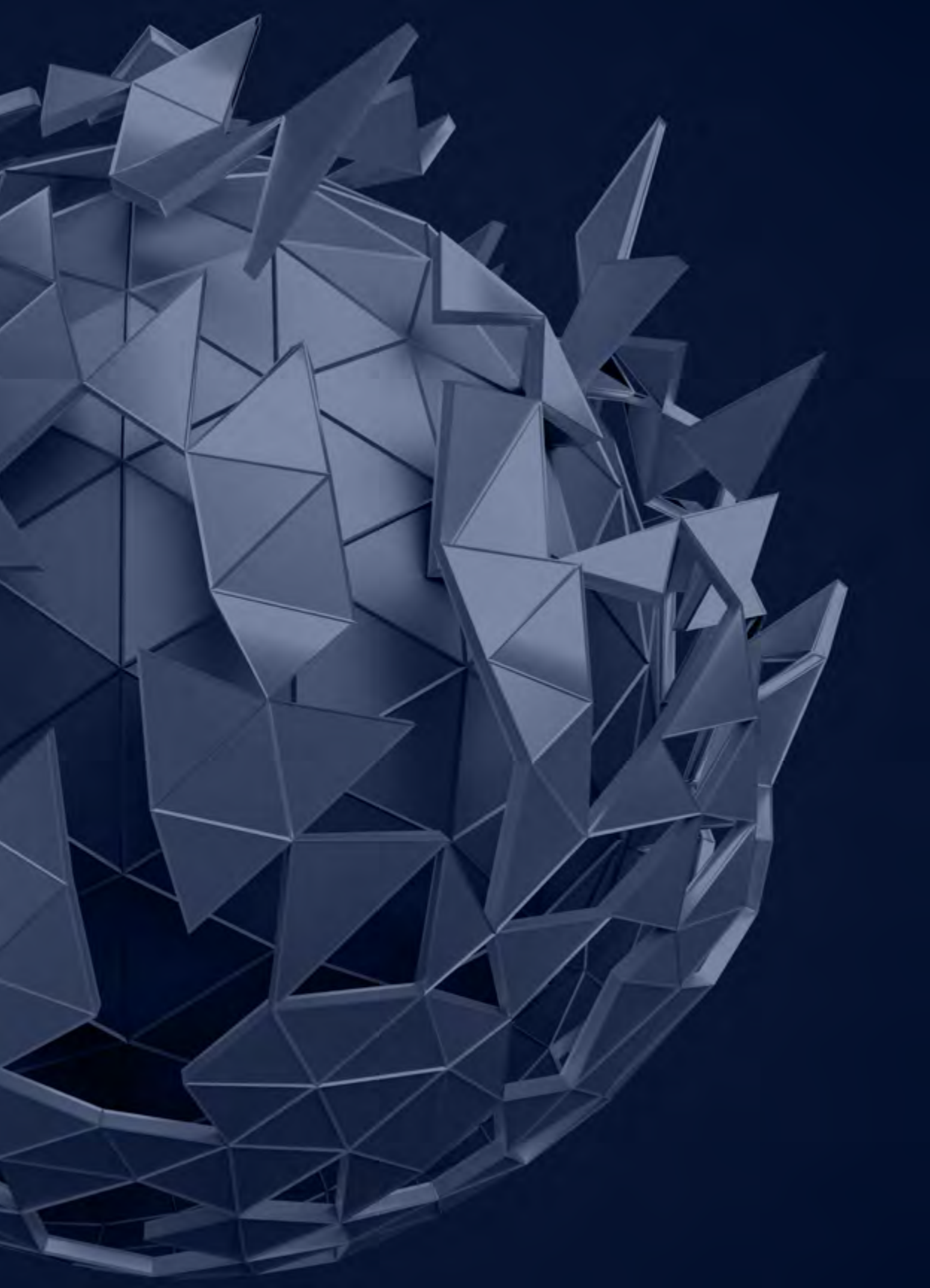
Michael McQuade
Vice President for Research, CMU

Nancy Pendleton
Vice President and Senior Chief Engineer for Mission Systems, Payloads and Sensors, Boeing Defense, Space and Security

Tim Dare
Defense Business Technical Director, Booz Allen Hamilton

William Scherlis
Director Information Innovation Office, Defense Advanced Research Projects Agency (DARPA)

Table of Contents



Foreword: Deb Frincke

Writing a foreword for this report has been both a privilege and a challenge. As the chair of the project's advisory board, I had the opportunity to work with some of the most knowledgeable and passionate individuals I have ever met. The resulting report is important and will be impactful on the future of software engineering. Consequently, it was a privilege to be associated with this work.

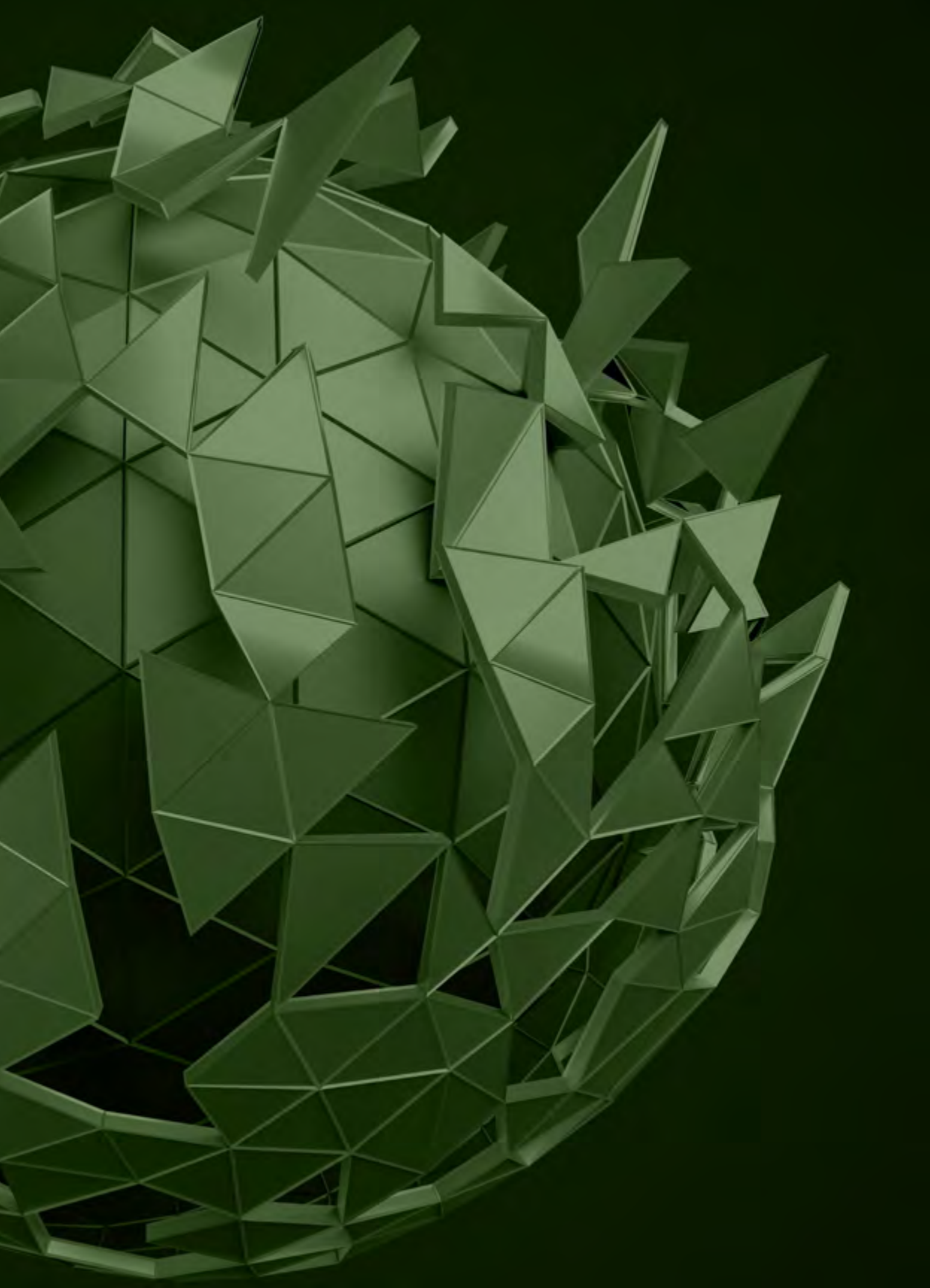
Software, and hence software engineering, problems contributed to the personal challenges I had in writing this foreword because they ate into my scheduled time to write. By chance, I was diverted three times by issues that juxtaposed humans and software-reliant systems. First, I was interrupted by technical challenges arising from ransomware in the context of critical infrastructure protection. Second, I became involved in key practical discussions about how to manage machine learning models that drive important scientific algorithms. And finally, I had to engage in a series of plaintive conversations with my air conditioning repair mechanic because of a software fault that caused my air conditioning to fail during one of the hottest weeks in the year. So while writing, I was actually experiencing the reason that motivated the need for this report: Software inadequacies resulting from inadequate software engineering are truly with us everywhere!

As you read this document, think about how software touches you, and everything around you, and what this implies for the future of software engineering. You will inevitably find that software resilience remains critical, and that software systems have become even more important to our daily lives than ever before. You'll also find that the increasing reliance on societal/global-scale systems highlights even more complexities, such as influence, social manipulation, and other challenges that emerge in these system types. All of this raises the stakes for software engineering.

My hope is that you will find ways to leverage this important report and the insights it contains, and that you will help enact its recommendations. We each have a responsibility to contribute to making software more trustworthy by advocating for investment in advancing the foundations and practice of software engineering.

Deb Frincke, Associate Laboratory Director for National Security Sciences, Oak Ridge National Laboratory

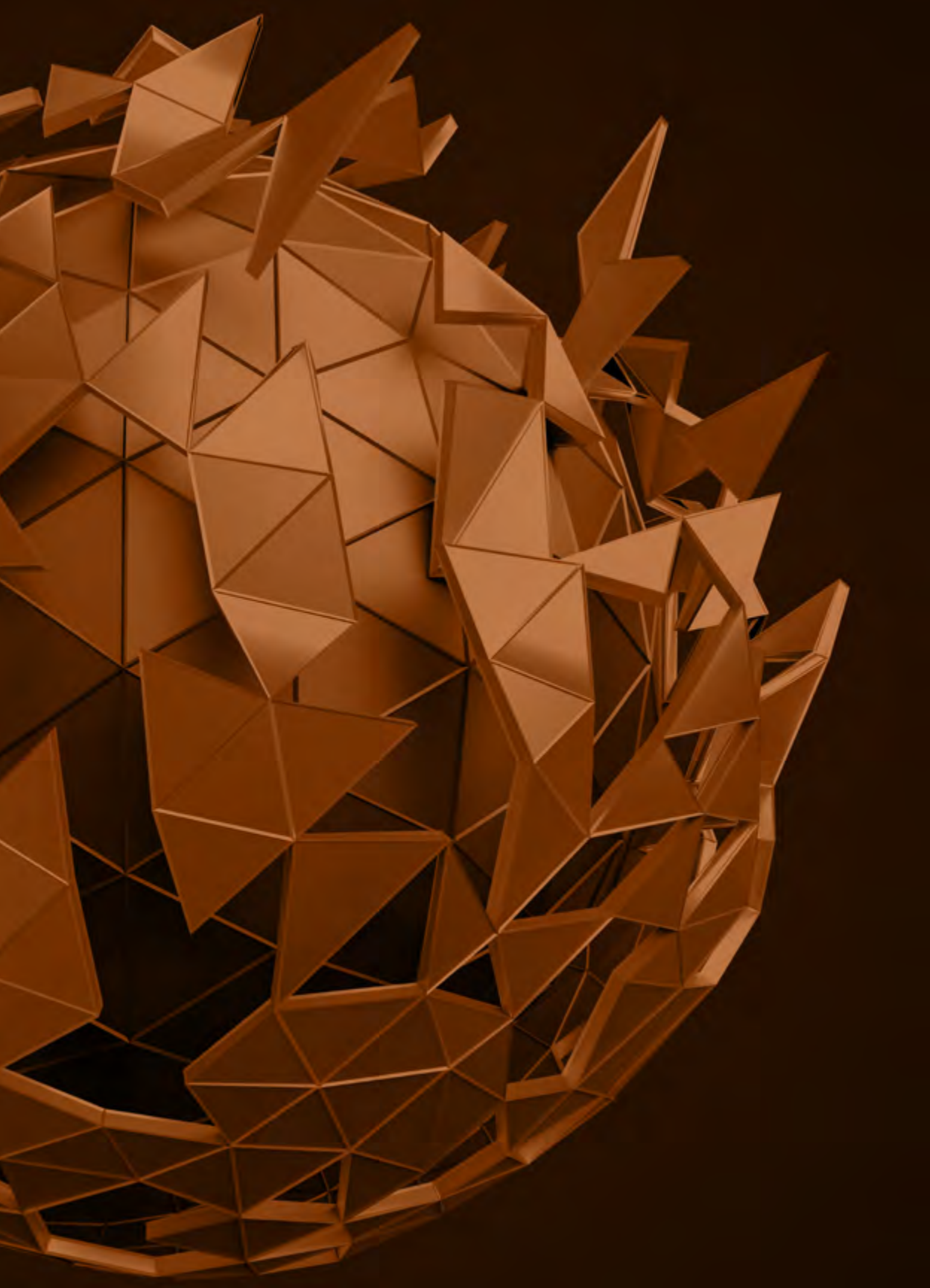
Advisory Board Chair for the National Agenda for Software Engineering Research and Development Study



Foreword: The Honorable Heidi Shyu

Software is an essential, if not the central, part of every Department of Defense (DoD) system. Our hardware has become increasingly programmable, and software has become ubiquitous. Therefore, software engineering is a critical enabler for everything that we do in the DoD. To remain competitive, our weapon systems acquisition must migrate away from the linear development and test cycle and evolve into a rapid continuous update and continuous assurance environment. Consequently, this software engineering technology roadmap is a guide for our research and investment strategy that is vital for our national security. As we develop new systems, we must go beyond model-based software engineering to enable us to rapidly develop systems while reducing re-assurance and sustainment costs. In the future, we will need rapid composition of new capabilities that can operate in a highly contested and denied environment. Integrating heterogeneous systems seamlessly and rapidly will enable us to stay ahead of threats. We will need to exploit the promise of artificial intelligence to increase capability not only in our fielded systems but also in our development systems. This research roadmap should serve as the starting point for a sustained effort to improve software engineering. The DoD will continue to look to the Carnegie Mellon University Software Engineering Institute as a leader in improving the state of the art and practice in software engineering.

The Honorable Heidi Shyu, Under Secretary of Defense for Research and Engineering



Acknowledgments

The world runs on software, and software engineering is the means for enabling the capabilities on which we have come to depend. Although advances in software have emerged incrementally and organically from many sectors and enabled commercial advances that were unimaginable twenty years ago, these current, fundamental piece-parts do not add up to the level of capability that future systems will require. Without a focused effort and continual investment and improvement in critical software engineering knowledge, technologies, and foundational software engineering research, next-generation applications may simply not be possible. Consequently, we felt it was imperative to orchestrate the creation of a National Agenda for Software Engineering Study to identify which technologies and areas of research are most critical for enabling future systems. The resulting roadmap is intended to guide the research efforts of the software engineering community. As we developed this roadmap, we asked ourselves, "How do we ensure that future software systems will be safe, predictable, and evolvable?"

With that brief introduction to our study as a backdrop, I would like to acknowledge the principal team of authors: Mark Klein, John Robert, Erin Harper, Rob Cunningham, Dio De Niz, Ed Desautels, John Foreman, John Goodenough, Charlie Holland, Ipek Ozkaya, and Forrest Shull, all from the Carnegie Mellon University Software Engineering Institute (SEI), along with James Herbsleb from the Carnegie Mellon University Institute for Software Research and Douglas Schmidt from Vanderbilt University. I am grateful for the opportunity to collaborate with this fabulous team who worked with passion, creativity, and determination to devise a compelling, thoughtful, and inspiring research roadmap for the future of software engineering.

It is interesting to note that this study was performed entirely during the global COVID-19 pandemic. That means that every part of the study was accomplished in a virtual environment: from designing the study and meeting with our advisory board, to the workshops we held to engage with the software engineering research communities, to working with our distributed team to assemble the study, all of it had to be done in a virtual setting. I want to thank everyone for their commitment to making time for this study and for finding creative ways to overcome the communication barriers and have meaningful conversations that contributed to this important topic.

Our team has appreciated the opportunity to work with senior thought leaders and luminaries in the field on our advisory board. It's been very helpful to have the breadth and depth of representation from different parts of the community on our board, including representatives from the Department of Defense (DoD), national labs, defense industrial base organizations, tech organizations, and academic leaders in computer science. We've been grateful for their enthusiastic participation and guidance along the way. In the early part of the study, they were instrumental in advising us on the research focus areas and helping us connect with the right people to work with on the study. Going forward, their attention has shifted to helping us think about who needs to know about this study and how we can enact our roadmap. We were most fortunate to have Dr. Deb Frincke as our advisory board chair. She demonstrated amazing leadership and commitment to this study because of her profound understanding of the critical importance of software engineering. The stellar advisory board included Vint Cerf, Penny Compton, Tim Dare, Sara Manning Dawson, Jeff Dexter, Yolanda Gil, Tim McBride, Michael McQuade, Nancy Pendleton, and William Scherlis. They were deeply committed to thinking about the future of software engineering and provided inspiring and thoughtful guidance throughout the study.

Next, our sincere thanks go to the many individuals who took time from their busy schedules to participate in or lead one of our virtual workshops. Work of this kind would be impossible without their willingness to share their experiences and ideas for the benefit of the software engineering community. We specifically want to thank Michele Falce, who provided critical ideas and contributions to enable all of the virtual workshops. We also thank the leaders and facilitators of each the workshops, including the following:

- National Agenda for Software Engineering R&D Workshop: Software Engineering Researcher Edition
  Keith Webster (CMU) and Barbora Batokova
- National Agenda for Software Engineering R&D Workshop: Voice of the Customer Workshop
  Harold Ennulat and Natalie Chronister
- Future Scenarios Workshop: Developing Plausible Alternative Futures
  Keith Webster (CMU)
- National Agenda for Software Engineering R&D Workshop: DoD Senior Leaders Workshop
  John Robert
- Software Engineering Grand Challenges and Future Visions Workshop
  Forrest Shull, Sandeep Neema (Defense Acquisition Research Projects Agency), Sol Greenspan (NSF), Christopher Ré (Stanford AI Lab)

Special thanks also go to the following people who shared their deep knowledge of the field in our expert interviews: Bob Bonneau, Penny Compton, Rob Cunningham, Tim Dare, Dio De Niz, Jeff Dexter, Deb Frincke, Yolanda Gil, John Goodenough, Jim Herbsleb, James Ivers, Grace Lewis, Ruben Martins, Michael McQuade, Ipek Ozkaya, Nancy Pendleton, Dan Plakosh, Bill Scherlis, Doug Schmidt, Mary Shaw, Eileen Wrubel, Hasan Yasar, and Robin Yeman. Jennifer Hykes and Marc Novakouski were also gracious enough to share their imaginative ideas for our section on future scenarios.

We would also like to thank our SEI colleagues in the communication, design, and production teams for their important roles in writing, editing, creating graphics, and web production. We especially thank Cat Zaccardi for her design team leadership and creativity, David Biber for his gorgeous visuals and page design, Donald Kurt Hess for his beautiful graphics, and Mike Duda for his print expertise and alacrity.

And most importantly, all of the authors of this study would like to share their sincere gratitude for the critical support, sponsorship, and contributions of the SEI Director's Office, including Dr. Paul Nielsen, Director and CEO; Mr. Dave Thompson, Deputy Director and Chief Operating Officer; and Dr. Tom Longstaff, Chief Technology Officer. They understood from the very beginning how challenging this activity would be, but also recognized what a critical contribution it would make to advancing the field of software engineering.

And finally, it has been a privilege for us to talk to so many software leaders with industry, academia, and government perspectives. Without exception, the discussions reaffirmed the critical importance of advancing software engineering for national competitiveness and meeting the increasing expectations of software across the globe. Recent news headlines highlight current software engineering limitations and are early indicators that software engineering is unprepared for the even greater challenges ahead. With your help, this roadmap will bring about a new era of multidisciplinary research and new partnerships to prepare us for those challenges and enable ongoing community discussion to advance the discipline of software engineering.

Anita Carleton, Software Solutions Division Director,
Carnegie Mellon University Software Engineering Institute

Executive Summary

Software Engineering as a Strategic Advantage

We live in an age of software-enabled transformation. Software, and all of the software engineering processes, practices, technologies, and the scientific domains that support it, makes our world-class healthcare, defense, commerce, communication, education, and energy systems possible. It is also a key enabling component in nearly every area of research, such as smart infrastructure (nanotech), human augmentation (biotech), and autonomous transportation. Our dependence on software, however, makes us vulnerable to its weaknesses. Software weaknesses are a direct reflection of inadequacies in the state of the art and practice of software engineering, and they can affect millions of people without warning. Just recently, software issues caused the largest shut-down of an oil pipeline in U.S. history and allowed attacks that paralyzed hundreds of businesses on five continents [Satter 2021]. Software quality problems have also led to loss of life in plane and car crashes, and expensive failures in the space flight industry [Rhee 2020; CBS 2010].

Without a catalyst for investing in software engineering, the situation will worsen as we increasingly depend on ever larger and more complex software-reliant systems. This report is intended to be such a catalyst. Identifying the critical technologies and areas of research that will enable future systems and laying out a roadmap to guide research efforts is a crucial step toward making software a competitive advantage. This study outlines efforts intended to make future software systems safe, predictable, and evolvable. The Carnegie Mellon University Software Engineering Institute (CMU SEI) engaged the software engineering community and assembled an advisory board of visionaries and senior thought leaders to ensure that the views of the broad software engineering ecosystem were represented in this multi-year research and development vision and roadmap.

Findings Reflect New Learnings, Challenges, and Research Needs

Without exception, the work that we surveyed for this study points to software engineering research as a highly dynamic, fast-moving field where technologies can arise quickly and grow to become integral parts of the infrastructure of modern life. While that is perhaps unsurprising, the extent to which recent technology trends are coming together and allowing the emergence of capabilities with both speed and quality is remarkable. Many of these technologies and capabilities were unimaginable even 10 years ago.

The following findings were derived from the state of software engineering practice, new trends and emerging technologies that will help to advance the state of software engineering practice, workshops held with software engineering research communities, a literature survey, interviews with experts in the field, and input from our advisory board. They summarize key learnings, key challenges, and new research needed for the future of software engineering.

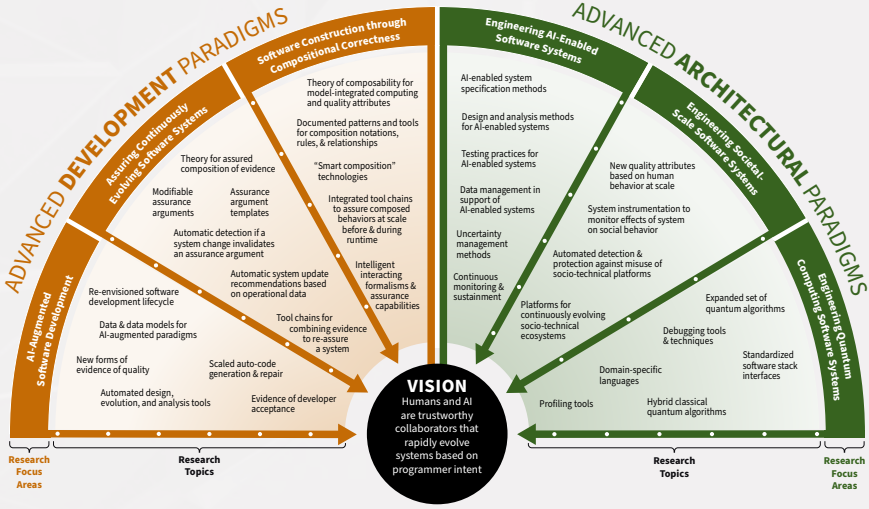
1. **Maintaining national software engineering proficiency is a strategic advantage.** Software engineering affects everything because software is everywhere, including in our nation's infrastructure, defense, financial, education, and healthcare systems. Our ever-growing dependence on software systems makes it imperative to maintain our nation's leadership and strategic advantage in software engineering. We need to raise the visibility of software engineering to the point where it receives the sustained recognition and investment commensurate with its importance to national security and competitiveness.
2. **Maintaining national software engineering proficiency requires sustained research.** New types of systems will continue to push beyond the bounds of what current software engineering theories, tools, and practices can support. Future systems and fundamental shifts in software engineering require new research focus in areas including smart automation, reassuring evolving systems, understanding composed systems, and new system types, such as AI-enabled systems, societal-scale systems, and quantum systems.
3. **Maintaining national software engineering proficiency requires fostering strategic partnerships.** We will need to enable strategic partnerships and collaborations to drive innovation in software engineering research among industry, research laboratories, academia, and government.
4. **Maintaining national software engineering proficiency requires sustained investment.** Policy makers must recognize the benefits of software engineering and make it a critical national capability. Such recognition would imply a sustained investment strategy.
5. **The vision of software engineering needs to change.** The current notion of a software development pipeline will be replaced by one where AI and humans collaborate to continuously evolve the system based on programmer intent.
6. **Focusing on re-assuring systems will enable continuous and rapid incorporation of new capability.** Because software is ubiquitous, there is an ongoing and increasing need for software to continuously evolve to incorporate new capability. We therefore need to understand how to continuously re-assure software reliant systems efficiently without doing harm to existing capability. Elevating the importance of assurance evidence and assurance arguments will be key.

7. **New design principles are needed for societal-scale systems.** The growing recognition of software's impact is generating new quality attribute requirements for which software engineers will need to develop better design approaches. In addition to the traditional ones (modifiability, reliability, performance, etc.), there is a need to add a roster of new quality attributes like transparency, influence, and so forth.
8. **The software engineering workforce needs to be (re-)conceived.** Software-reliant systems are built for many different purposes by a broad collection of people with very disparate skill sets, many of whom do not have formal software engineering training. We need to better understand the nature of the needed workforce and what to do to foster its growth.

A Guiding Vision and Roadmap for the Future of Software Engineering

Our guiding vision, as described in our findings, is one in which the current notion of the software development pipeline is replaced by one where humans and software are trustworthy collaborators that rapidly evolve systems based on programmer intent. To achieve this vision, we anticipate the need for new development and architectural paradigms for engineering future systems.

Our study helped to inform new areas of research that must be met to advance software engineering for future systems. In close collaboration with our advisory board and other leaders in the software engineering research community, we developed a research roadmap with six research focus areas. The following figure shows those areas along with a list of research topics to undertake, and then short descriptions of each of the research focus areas follow. A larger version of this figure appears on the foldout after page 26.



Software Engineering Research Roadmap with Focus Areas and Research Objectives (10–15 Year Horizon)

AI-Augmented Software Development. At almost every stage of the software development process, AI holds the promise of assisting humans. By relieving humans of tedious tasks, they will be better able to focus on tasks that require the creativity and innovation that only humans can provide. To reach this important goal, we need to re-envision the entire software development process with increased AI and automation tool support for developers. A key challenge will be taking advantage of the data generated throughout the lifecycle. The focus of this research area is on what AI-augmented software development will look like at each stage of the development process and during continuous evolution, where AI will be particularly useful in taking on routine tasks.

Assuring Continuously Evolving Software Systems. When we consider the software-reliant systems of today, we see that they are not static (or even infrequently updated) engineering artifacts. Instead, they are fluid—meaning that they are expected to undergo almost continuous updates and improvements and be shown to still work. The goal of this research area is, therefore, to develop a theory and practice of rapid and assured software evolution that enables efficient and bounded re-assurance of continuously evolving systems.

Software Construction through Compositional Correctness. As the scope and scale of software-reliant systems continues to grow and change continuously, the complexity of these systems makes it unrealistic for any one person or group to understand the entire system. It is therefore necessary to integrate (and continually re-integrate) software-reliant systems using technologies and platforms that support the composition of modular components. This is particularly difficult since many of such components are reused from existing elements that were not designed to be integrated or evolved together. The goal of this research area is to create methods and tools that enable the specification and enforcement of composition rules that allow (1) the creation of required behaviors (both functionality and quality attributes) and (2) the assurance of these behaviors.

Engineering AI-Enabled Software Systems. AI-enabled systems, which are software-reliant systems that include AI and non-AI components, have some inherently different characteristics than those without AI. However, AI-enabled systems are, above all, a type of software system. These systems share many parallels with the development and sustainment of more conventional software-reliant systems. This research area focuses on exploring which existing software engineering practices can reliably support the development of AI systems, as well as identifying and augmenting software engineering techniques for the specification, design, architecture, analysis, deployment, and sustainment of systems with AI components.

Engineering Socio-Technical Systems. Societal-scale software systems, such as today's commercial social media systems, are designed to keep users engaged and often to influence them. A key challenge in engineering societal-scale systems is predicting outcomes of the socially inspired quality attributes that arise when humans are integral components of the system. The goal is to leverage insights from the social sciences to build and evolve societal-scale software systems that consider these attributes.

Engineering Quantum Computing Software Systems. Advances in software engineering for quantum are as important as the hardware advances. The goals of this research area are to first enable current quantum computers to be programmed more easily and reliably, and then enable increasing abstraction as larger, fully fault-tolerant quantum computing systems become available. A key challenge is to, eventually, fully integrate these types of systems into a unified classical and quantum software development lifecycle.

Research and Enactment Recommendations Catalyze Change

Catalyzing change that advances software engineering will lead to more trustworthy and capable software-reliant systems. The research focus areas shown in the roadmap graphic previewed earlier in this section and on foldout following page 25 led to a set of research recommendations that are necessary to catalyze change, which are followed by enactment recommendations that focus on people, investment, and sustainment are needed.

The following research recommendations address challenges such as the increasing use of AI, assuring changing systems, composing and re-composing systems, and engineering socio-technical and heterogenous systems.

1. **Enable AI as a reliable system capability enhancer.** The software engineering and AI communities should join forces to develop a discipline of AI engineering. This should enable the development and evolution of AI-enabled software systems that behave as intended and enable AI to be used as a software engineering workforce multiplier.
2. **Develop a theory and practice for software evolution and re-assurance at scale.** The software engineering research community should develop a theory and associated practices for re-assuring continuously evolving software systems. A focal point for this research is an assurance argument, which should be a software engineering artifact equal in importance to a system's architecture, that ensures small system changes only require incremental re-assurance.

3. **Develop formal semantics for composition technology.** The computer science community should focus on the newest generation of composition technology to ensure that technologies such as dependency-injection frameworks preserve semantics through the various levels of abstraction that specify system behavior. This will allow us to reap the benefits of development by composition while achieving predictable runtime behavior.
4. **Mature the engineering of societal-scale socio-technical systems.** The software engineering community should collaborate with social science communities to develop engineering principles for socio-technical systems. Theories and techniques from disciplines such as sociology and psychology should be used to discover new design principles for socio-technical systems, which in turn should result in more predictable behavior from societal-scale systems.
5. **Catalyze increased attention on engineering for new computational models, with a focus on quantum-enabled software systems.** The software engineering community should collaborate with the quantum computing community to anticipate new architectural paradigms for quantum-enabled computing systems. The focus should be on understanding how the quantum computational model affects all layers of the software stack.

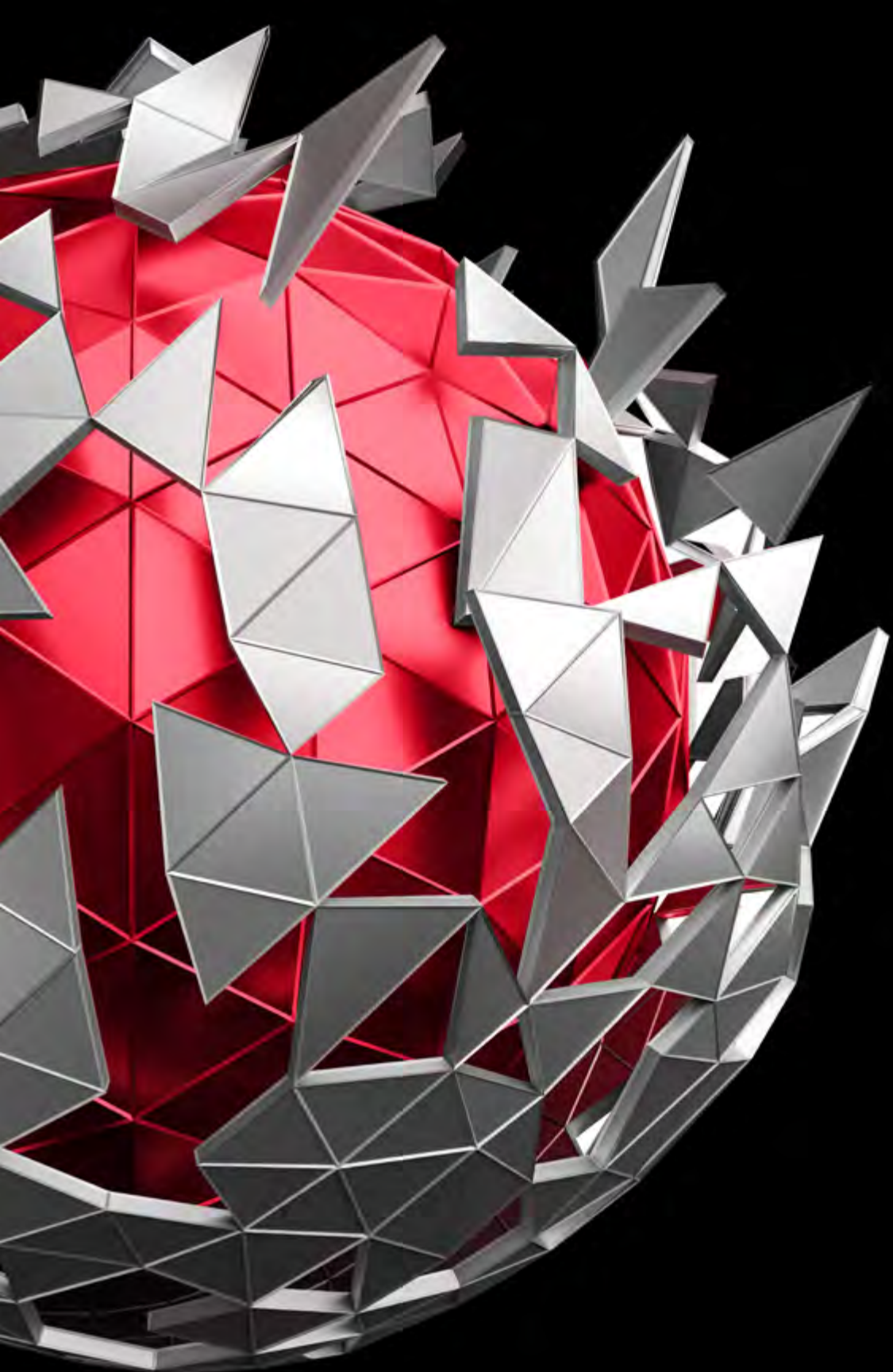
The above recommendations focused on scientific and engineering barriers to achieving change. The following enactment recommendations focus on institutional obstacles, including economic, human, and policy barriers.

6. **Ensure investment priority reflects the importance of software engineering as a critical national capability.** The strategic role of software engineering in national security and global market competitiveness should be reflected in national research activities, including those undertaken by the U.S. White House Office of Science and Technology Policy (OSTP) and Networking and Information Technology Research and Development (NITRD). These research activities should recognize software engineering research as an investment priority on par with chip manufacturing and AI with benefits to national competitiveness and security. Software engineering grand challenges sponsored by DARPA, the National Science Foundation (NSF), and FFRDCs are also suggested.

7. **Institutionalize ongoing advancement of software engineering research.** Sustained advancements in software engineering requires institutionalizing an ongoing review and reinvestment cycle for software engineering research and its impact on software engineering practice. Maintaining national software engineering proficiency requires research funding sources and institutes working with industry and government leaders in the software engineering community to periodically review the state of software engineering.
8. **Develop a strategy for ensuring an effective workforce for the future of software engineering.** Currently, software engineering is performed by a broad collection of people with an interdisciplinary skill set not always including formal training in software engineering. Moreover, the nature of software engineering seems to be changing in reaction to the fluid nature of software-reliant systems. We need to better understand the nature of the needed workforce and what to do to foster its growth. The software engineering community, software industry, and academic community should create a strategy for ensuring an effective future software engineering workforce.

Architecting Future Systems Requires Software Engineering Advances

Due to the conceptual nature of software, it continues to grow, without bounds, in capability, complexity, and interconnection. There seems to be no plateau in the advancement of software. To make future software systems safe, predictable, and evolvable, the software engineering community—with sufficient investment from private and public sources—must work together to advance the theory and practice of software engineering strategically to enable the next generation of software-reliant systems.



1 Introduction

We live in an age of software-enabled transformation. Over the last half-century, software has become profoundly intertwined in our personal lives, and it is vital to our country's global competitiveness, innovation, and national security. As society entrusts software with ever more complex and critical functionality, our reliance on future software systems will increase—yet systems will be significantly more complicated to build and maintain. Software engineering is the discipline entrusted with building and maintaining these pervasive software systems.

Through the application of engineering to software, the necessary theories, tools, and practices are applied that enable the delivery and maintenance of software systems that are capable, reliable, timely, and affordable [Bourque 2014]. As systems continue to evolve, we can be almost certain that new types of systems will push beyond the current bounds of software engineering. We will not be able to develop and maintain future software systems adequately unless appropriate research is done to overcome the engineering problems inherent in new and emerging trends and software technologies. Software offers unlimited potential that can only be realized through advancements in software engineering.

1.1 Software Enables Capability and Innovation

Software provides the capabilities for many activities essential to modern life. It enables the functionality of our cell phones, cars, medical equipment, and much more. Software also enables innovation. In today's cars, for example, every component is connected to a central computer, and millions of lines of code enable all the features we have come to expect [McFadden 2021].

Software is in everything, and everything is in software. Software connects decision makers to data, improves the flow of goods to customers, and enables communication worldwide. We can connect with geographically dispersed friends and family through social networks enabled by software. We can easily access oil, gas, and electricity because software manages their flow through pipelines and power grids. Thanks to software, we often take for granted the appearance of these and other critical commodities in our daily lives.

1.2 Software Is an Achilles Heel

Although software is an enabler that society has grown to depend on, our dependence has also made us vulnerable to its weaknesses. For example, recent software quality problems have resulted in vulnerabilities that allowed hackers to gain access to data from billions of individuals, companies, and government offices [Tunggal 2021]. Software vulnerabilities also caused the largest shutdown of an oil pipeline in U.S. history and allowed attacks that paralyzed hundreds of businesses on all five continents [Satter 2021]. Software quality problems have led to loss of life in plane and car crashes, and expensive failures in the space flight industry [Rhee 2020; CBS 2010]. In fact, the total cost of poor software quality in the United States in 2020 was $2.08 trillion, according to the Consortium for Information and Software Quality (CISQ) [Krasner 2021].

1.3 Software Is the Backbone of Safety-Critical Systems

Many multi-national companies are experiencing the hard failures that come when software engineering efforts do not reach the level of quality demanded by their systems. For example, a major space initiative has been plagued with faulty designs, software errors, and issues with assurance practices [Pasztor 2021]. To identify the root cause of these mounting problems, experts point to a lack of software engineering leadership and discipline [McFall-Johnsen 2020]. Space initiatives are hardly alone in these challenges. For example, unintended acceleration related to software in several different automobiles is thought to have been involved in the deaths of many people over the past decade, and additional problems with braking control in cars can be traced to problematic and poor quality software [Mitchell 2010].

1.4 Software Often Relies on Complex Supply Chains

The modern software supply chain often includes a large number of stakeholders that contribute to the content of a software product or have the opportunity to modify its content. Therefore, the entire supply chain is an important part of the ecosystem that must be considered when we contemplate software quality. (See Figure 1 for an example of the complex relationships that can exist in a supply chain for a DoD system.[1])

The increasingly global nature of software development has raised concerns about supply chain attacks.[2] These types of attacks are rapidly growing in number and scope, and they are made more effective by increasingly interconnected systems and the lack of transparency in software codebases and libraries.

1 Figure modified from the SEI white paper by Dorofee et al., A Systemic Approach for Assessing Software Supply-Chain Risk. Software Engineering Institute. February 2013.

2 Supply chain attacks are cyberattacks that seek to damage an organization or gain access to information by targeting less-secure elements in the supply chain. In this type of attack, hackers can infect a single component that is then distributed downstream to many systems through legitimate software workflows and patches or updates.

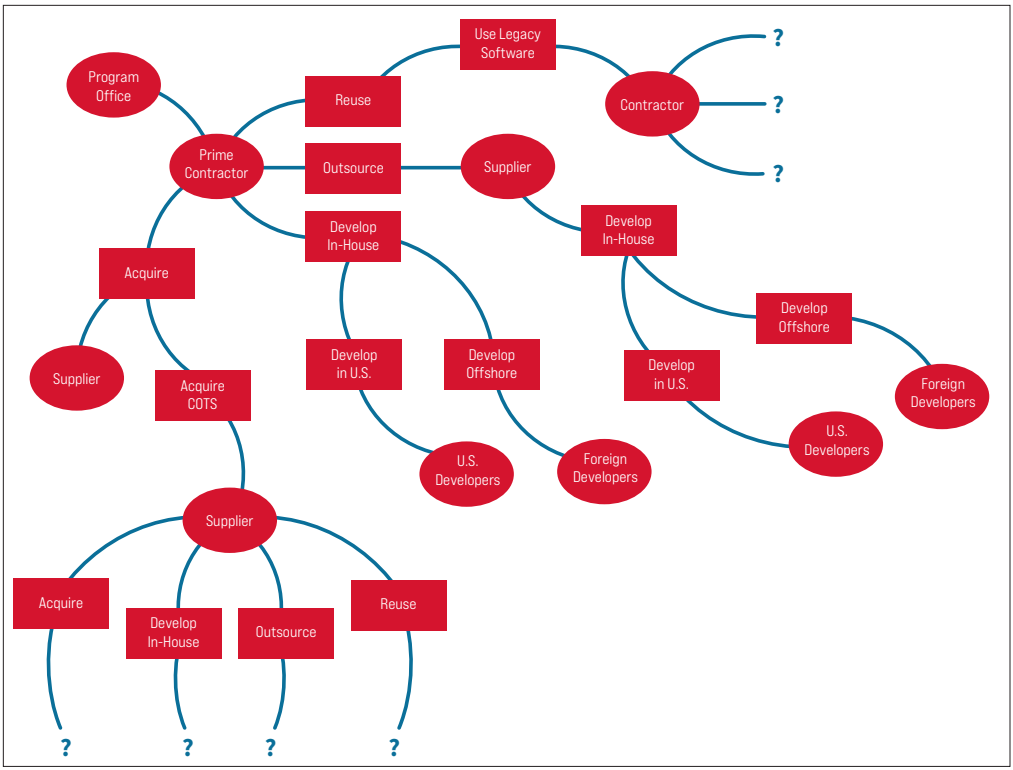


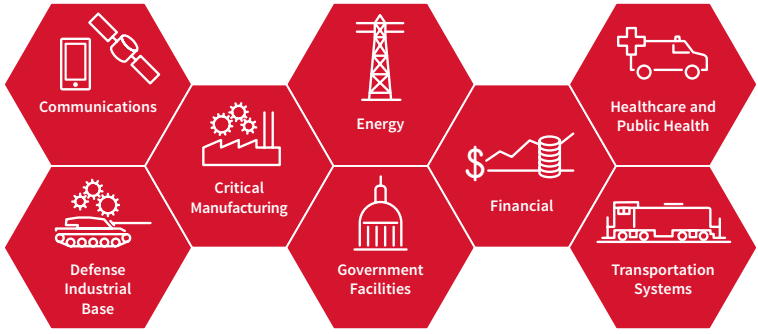
Figure 1: Software Supply Chain Relationships Example

For example, in 2020 a U.S. company that develops software to help businesses manage their networks and information technology infrastructures proved to be an ideal target for the largest known supply chain attack to date. By inserting malicious code into a routine software update, Russian hackers were able to compromise third-party software used by four-fifths of Fortune 500 companies (including Microsoft, Intel, FireEye, and Deloitte) and many U.S. government agencies (including the Department of Homeland Security, the Department of State, and the DoD). Altogether, about 30,000 public and private organizations were using the potentially infected software, which led to a web of compromised data, systems, and networks [Turton 2020].

Understanding the functionality and quality of the code used, and documenting and validating the supply chain, is important. Supply chain integrity is critical, yet the reuse of code that is of low or unknown quality is commonplace. Tracking the provenance of software is one way to combat this problem, and developing techniques and tools for doing so would contribute to improving software quality.

1.5 Software Is a Component of Critical Infrastructure

The development or reuse of poor quality software can introduce vulnerabilities that allow cybercriminals to access the software that controls our critical infrastructure and wreak havoc. One such high-profile attack occurred in May 2021 on the Colonial Pipeline, which transports approximately 100 million gallons of gasoline, diesel, and jet fuel daily, supplying about 45% of all fuel consumed on the East Coast [Eaton 2021]. The hacker group DarkSide launched a ransomware attack[3] against the pipeline's systems, causing the company to shut down all of its operations to contain the attack. This incident is being characterized as one of the most significant attacks on critical infrastructure in history. Ransomware attacks increased by 715% in 2020 and are currently the fastest growing type of cyberattack [Bitdefender 2020]. About 1,000 organizations per week are being hit by ransomware attacks, with utilities the second-most-common target, behind healthcare organizations [Lanowitz 2021].



While many of the long-term challenges in developing and deploying secure software in critical infrastructure are known, the problems remain, and poor quality software code continues to propagate vulnerabilities throughout our infrastructure.

1.6 Software Engineering Determines Software Quality

Software failures are a direct reflection of inadequacies in how software is developed and maintained [van Genuchten 2019; Shaw 2002]. That is, poor quality software is the direct result of the current state of the art and practice in software engineering. Some effects are highly visible, such as the lives lost due to the loss of control of physical objects. Other effects are less visible, such as when vehicle emissions systems perform poorly or cell phone apps collect and share data without permission from the user. Without a catalyst for investing in software engineering, the situation will

3 The U.S. Cybersecurity and Infrastructure Security Agency (CISA) defines ransomware as "an ever-evolving form of malware designed to encrypt files on a device, rendering any files and the systems that rely on them unusable. Malicious actors then demand ransom in exchange for decryption."

worsen due to an ever-increasing dependence on increasingly large and complex software-reliant systems. This report is intended to be such a catalyst for making software engineering a strategic advantage.

1.7 Call to Action

Although advances in software have emerged incrementally and organically from many sectors, and enabled advances that were unimaginable 20 years ago, they do not provide the levels of capability, safety, quality, and evolvability that future systems will require. While sound research in software engineering is being carried out, a focused effort, continual investment, and improvement in critical software engineering technologies are needed; otherwise, assured, next-generation applications may simply not be possible.

A focused effort, continual investment, and improvements in critical software engineering technologies are needed; otherwise, assured next-generation applications may simply not be possible.

This study identifies areas of research that are critical for enabling future systems and provides a roadmap to guide the research efforts of the software engineering community. As we developed this roadmap, we placed primary importance on what is needed to ensure that future software systems will be safe, predictable, and evolvable [DSB 2018; DoD 2018a; DoD 2018b; Office of the President 2017].

This report is a call to action that highlights the need for continual investment in software engineering research to achieve the vision described by the research roadmap. Research investment must be commensurate with software engineering's importance to national security and competitiveness, and motivating industry and government investment partnerships will be a key element of a successful plan.

1.8 Scope

This study addresses the following questions:

- How will software systems of the future be rapidly developed, assured, analyzed, and deployed?
- What major open problems and "grand challenges" are important?
- What software engineering research is needed to invent solutions for these challenges?
- How can we incentivize strategic partnerships and collaborations between government, academia, and industry?

It is important to emphasize that software engineering cannot be considered in isolation and requires a whole-system perspective, which includes software, hardware, and people. We include elements of this thinking in the report, but primarily focus our discussion on the software engineering research agenda. It is also important to emphasize that this study is intended to be applicable to all types of software-reliant systems, such as safety-critical military and commercial systems; business and logistics systems; and systems that support research of all types.

While software-engineering-enabled solutions have the potential for transformative impacts across all sectors of society and the economy, there are also concerns about the security and vulnerability of these systems. This study does not directly address cybersecurity in depth because its importance is already well established, and the SEI and others continue to publish studies this area.

As the resources driving the AI revolution continue to grow, the development and deployment of these technologies is poised not only to continue but to accelerate. We have positioned our discussion around AI as a capability enhancer as well as a source of engineering uncertainty, but we do not propose a research agenda for AI. The SEI and others have significant programs of research dedicated to machine learning and AI.

1.9 Audience

Some of the intended audiences for this study are described below:

- Industrial researchers, academic researchers, technologists, and research laboratories may find interesting the identification of important open problem areas where research solutions could be particularly impactful, and where new modes of working across the industry/academia divide could yield dividends.
- Research funders, policy-makers, and legislative representatives may appreciate the argument that investments in the areas identified have the potential to broadly support important developments in research and practice across numerous domains.
- Software developers, practitioners, and program managers may find useful the reminder of key challenge areas in the field today and may be inspired to work with researchers to help address them more broadly.
- Federally funded research and development centers (FFRDCs) may be inspired to work together on some of these fundamental software issues that will help address national priorities across the many specific areas of focus for the various FFRDCs.
- Educators may appreciate the snapshot of the state of the practice today in the field (and the articulation of its limits) for use in classes and curricula.
- Industry leaders may find areas of research and workforce improvement that complement their needs and identify ways to work with researchers on critical issues in the commercial realm.

1.10 Approach

Software engineering exists as a global ecosystem that includes many stakeholders with different perspectives, including software developers, software tool vendors, companies that integrate software into their products, software researchers, and government sponsors of software research. For this study, the CMU SEI engaged the software engineering community and assembled an advisory board of visionaries and senior thought leaders across commercial industry, academia, and government. With their input, the study team worked to create the multi-year research and development vision and roadmap for engineering next-generation software-reliant systems in this document.

Coordination among these communities was vital to developing the agenda and will also be needed to implement the results. Understanding the diverse software engineering ecosystem, identifying future needs, and determining ways to effect change required a range of activities, as summarized in Figure 2.

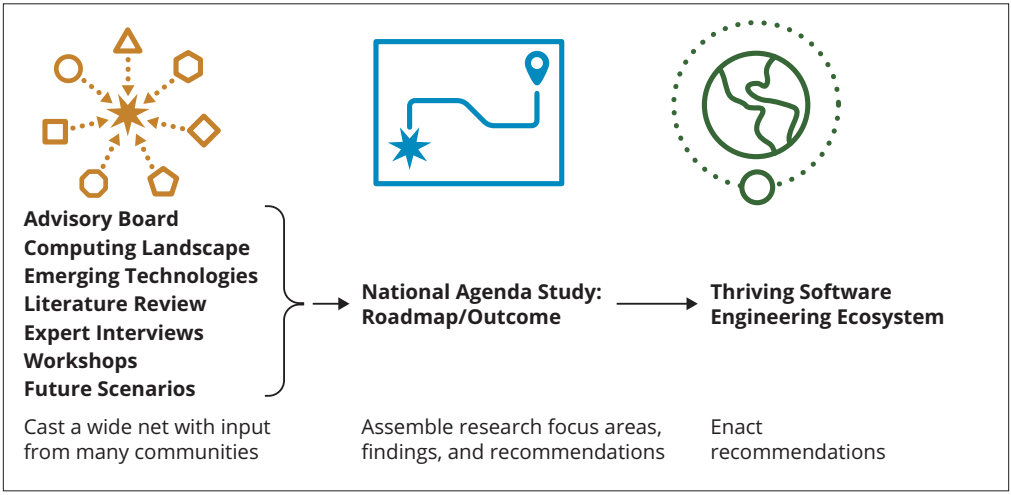
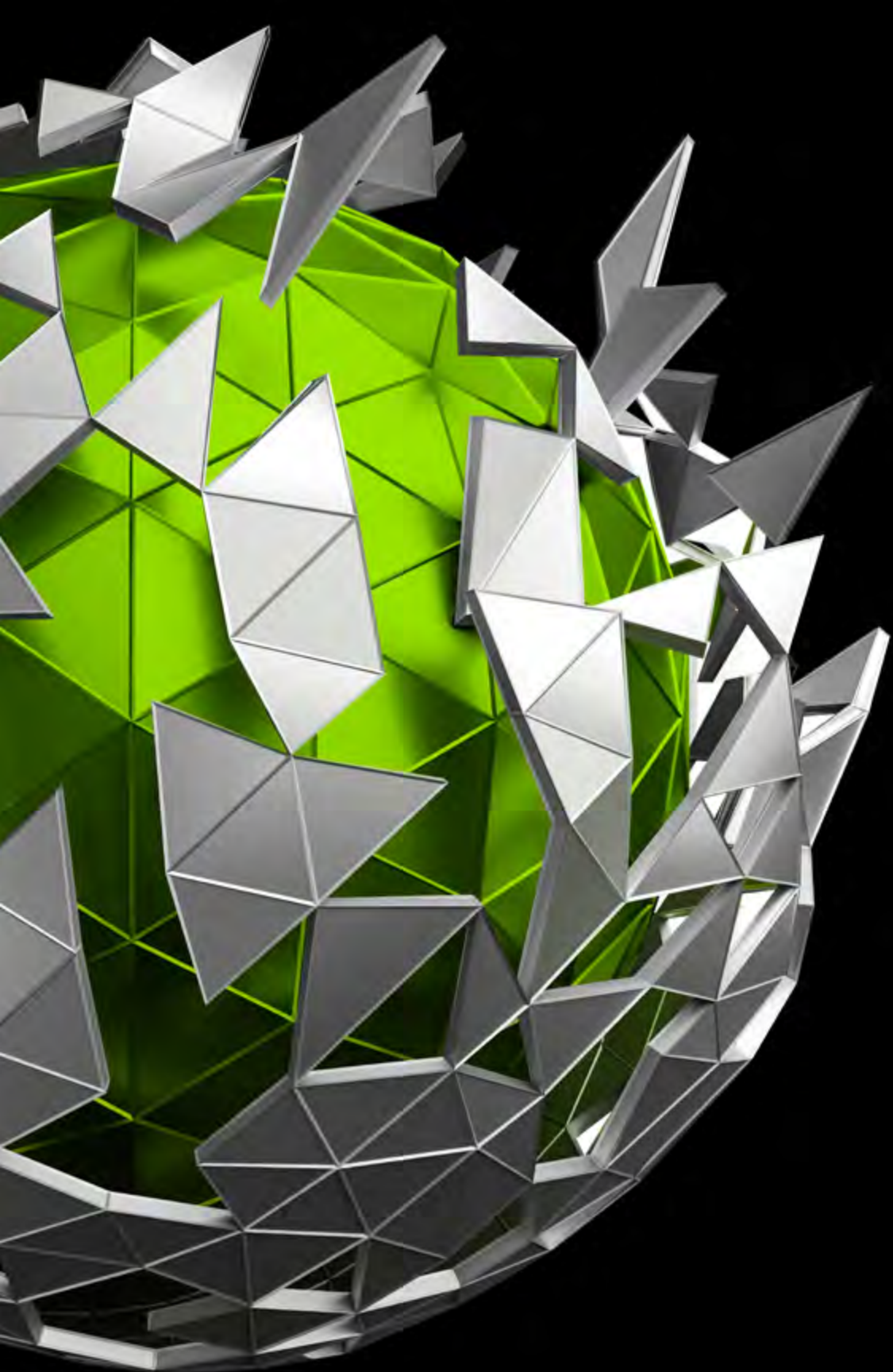


Figure 2: Approach for Developing This Study

As we developed this study, the team conducted background research and literature surveys, held workshops, performed expert interviews, evaluated computing and software trends and emerging technologies, developed future scenarios, worked with our advisory board, and examined software-related economic and business data. Appendix A provides additional information on the workshops that were held to engage with software engineering communities, including a cohosted workshop with the Defense Advanced Research Projects Agency (DARPA).



2 Exploring Emerging Trends and Technologies

Advances in computing technologies continue to be a key driver for U.S. leadership in science and technology, national security, and economic competitiveness [DIB 2019]. To anticipate the research and development that will be needed to support software engineering in the future, it is important to keep a close watch on the emerging trends and technologies that help to inform new challenges and opportunities.

Although it is not possible to explore them all in one document, those included in this section help to paint a picture of the technology landscape that is impacting software engineering research.

2.1 Trends

Trends grow and change constantly in today's fast-moving world. The following paragraphs focus on several current trends that we believe are important for envisioning how software systems will look in the future.

The software engineering pipeline is changing, accelerating the production of code and the ability to deploy software at high velocity. Private and public sector enterprises today face the challenges of a rapidly changing competitive landscape, evolving security requirements, and performance scalability. Enterprises are working to adopt rapid development and deployment with innovation and confidence, bridging the gap between operations stability and rapid feature development. At the scale of large aerospace organizations or product organizations such as Amazon, this often means thousands of independent software teams must be able to work in parallel to deliver software quickly, securely, reliably, and with zero tolerance for outages or errors. Rapid development practices, such as continuous integration/continuous development (CI/CD) and DevSecOps, are being used to deliver software features rapidly and reliably. Further progressing on this rapid development/deployment continuum, the notion of a software engineering pipeline is morphing into a fluid process through which new capability is introduced into ever-evolving systems.

New types of systems will continue to push beyond the bounds of what current software engineering theories, tools, and practices can support [Kim 2019; Murphy 2020; NITRD 2011; Weyuker 2021; Wing 2021]. For example, trends already point toward the development and increasing use of these system types:

- *Very adaptive mission defense systems.* Software increasingly enables new heterogeneous computing systems that combine intelligence, weapons, human-machine teaming, and other capabilities.
- *Systems that perform large-scale data fusion.* Whether for news or intelligence, these systems take advantage of vast data streams, including open source data. These data streams will also drive new ways of constructing future systems.
- *Smart cities, buildings, roads, cars, and other transport.* Software systems are now integral to critical infrastructure in these domains, and they need to handle integration at scale as well as deal appropriately with safety and privacy concerns.
- *Personal digital assistants—that really assist.* Software systems must learn and adapt as part of their integration in home, business, and national security workflows, as well as our personal lives.
- *Dynamically integrated healthcare.* Devices from home, doctors' offices, and hospitals will be increasingly integrated in functionality and data usage. This integration will result in better preventive, corrective, and recovery care.
- *Societal-scale systems.* These platforms, enabled by advances in connectivity, AI, and data science, are becoming larger and more influential. As these systems grow, they influence social behavior and create impact at the societal level. The trend toward these types of systems has exploded over the last decade, with 3.96 billion people using social media worldwide [Dean 2021].

Scale motivates the need for safe and resilient software composition. The scope and scale of software-reliant systems is continuously changing and growing [NRC 2010]. As improvements in computer hardware enable the development of more complex, advanced software, and as more devices connect to the network through sensors and the Internet of Things (IoT), it becomes clear that increasing scale is a trend with no sign of slowing down. Developing and sustaining software components from scratch in these large, complicated systems is no longer realistic. Consequently, a common trend is to integrate (and continually reintegrate) software-reliant systems out of modular components, many of which are reused from existing elements.

The development and sustainment of artificial intelligence (AI) systems shares many parallels with building, deploying, and sustaining software systems. AI has captured the public imagination, as well as extensive investment and research dollars [Gil 2019]. The use of AI is an expanding trend, as it is increasingly employed across industries. While AI is a field unto itself with many sub fields and applications, it has great potential for use in software development. AI-augmented software development holds promise for automating common or tedious tasks and for making processes more efficient, effective, and enjoyable for humans. Research programs in software engineering will need to focus on the challenges that AI elements bring to software analysis, design, construction, deployment, maintenance, and evolution.

Research programs in software engineering will need to focus on the challenges that AI elements bring to software analysis, design, construction, deployment, maintenance, and evolution.

Data privacy and trust are increasingly important design considerations for software systems. Data is now a strategic asset that is bundled, shared, sold, and dispersed around the world. Appropriately using this data while simultaneously protecting it and preventing its misuse presents serious architectural and software engineering challenges related to privacy, trust, and ethics. Technologies are being developed to help protect data, such as those that allow differential privacy. These technologies are important for things like the census, medical analyses, and other data analysis efforts that involve gathering information about individuals. Trust is related to the confidence you have in the data or output of a system, and is of particular concern to society in systems that contain AI. Other technologies have the potential to build trust, such as blockchain, a distributed ledger technology. It is enabling new opportunities in software engineering, with applications in software testing, quality, configuration management, and maintenance [Demi 2021].

2.2 Emerging Technologies

The robust technology ecosystem we have today means new technologies are introduced constantly, and many more are on the horizon. Understanding the capabilities these technologies can bring and how to integrate them into systems quickly, securely, and with predictable performance is key to making sure they are an asset to software systems instead of a source of weakness or instability. In the following paragraphs, we briefly highlight some technologies that directly impact software engineering [Holland 2020].[4]

Advanced computing is creating new engineering challenges in composing and evolving systems. Advanced computing generally refers to a set of capabilities that are beyond the reach of desktop computers and the general public. It often means using specialized software or hardware to provide advanced technical capabilities that support massive, data-intensive projects. Some examples of advanced computing include high-performance computing (often for simulations and modeling), large cloud computing implementations, and the use of quantum mechanics and information theory.

The last decade has seen many developments in advanced computing supported by new hardware, such as multicore chips, graphics processing units, field programmable gate arrays, and application-specific integrated circuits at the chip level. A long-term technological opportunity also exists to develop a software ecosystem that enables scalable quantum computing. Advanced computing underscores the fact that the computing environment of the future will be increasingly heterogeneous, which will create new challenges in composing and evolving systems across computational foundations.

The **smarter edge** presents new challenges due to scale. The smarter edge is a catch-all term for new advancements to push heterogeneous computing power, applications, and data to the edge of the Internet. It goes beyond a conventional computer network and incorporates devices at the edge of the network such as sensors, IoT devices, and mobile phones. While the concept of ubiquitous computing has existed for decades, there have been recent advancements to accelerate the smarter edge, including hardware improvements and the expansion of 5G networks. Edge data is growing rapidly, thanks to ubiquitous sensing and the IoT, and the field of analytics is creating innovative new ways for distributed data analysis using a combination of edge devices and central processing. The future smarter edge might even include more nontraditional devices, such as space-based satellite mega-constellations to enable the next wave of connectivity.

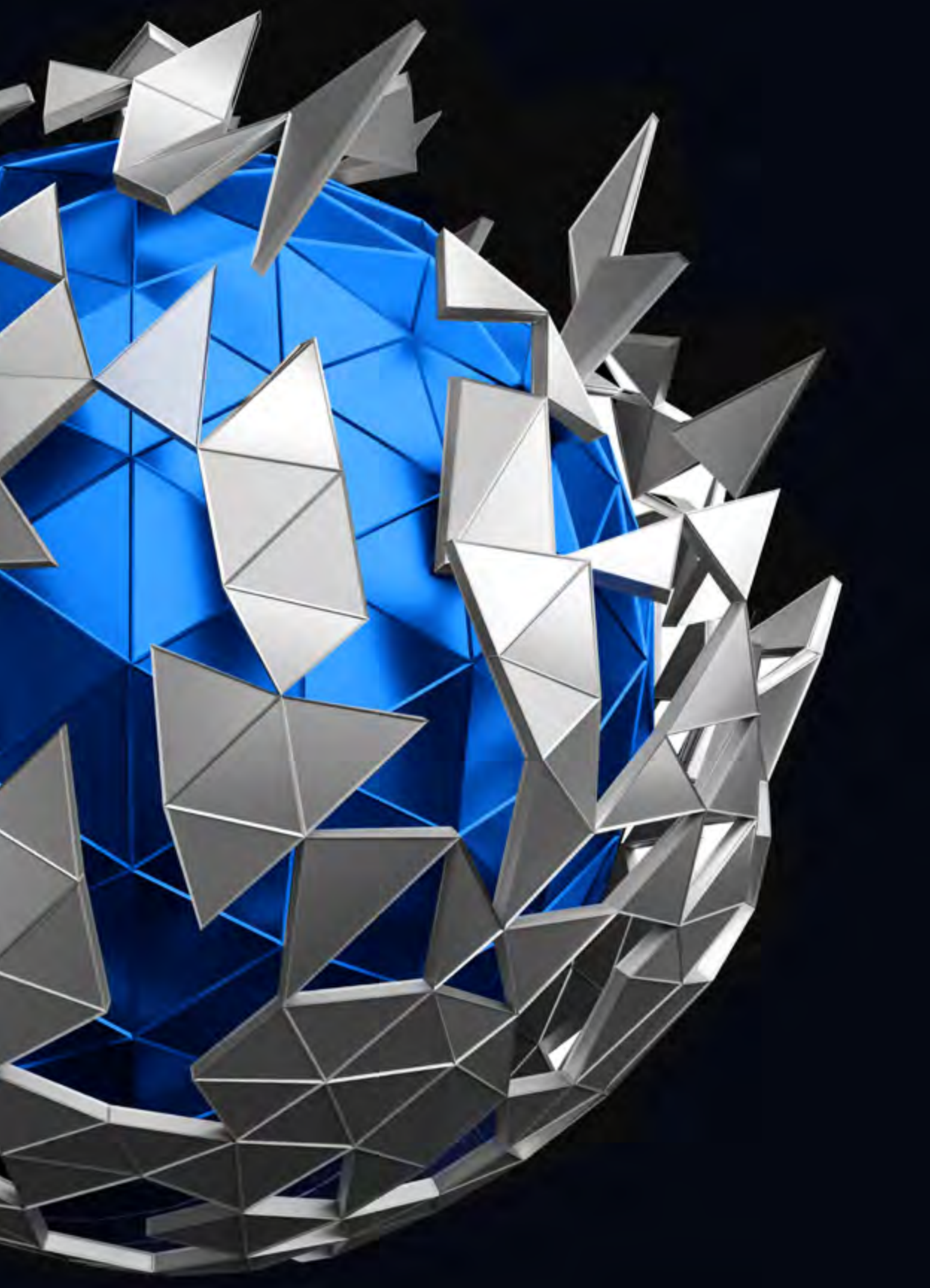
4 See the SEI paper by Holland and Tanenbaum, Emerging Technologies 2020: Six Areas of Opportunity, for more information on the technologies in this section.

Digital twins create new opportunities and challenges for assuring systems. A digital twin is a high-fidelity digital or computer representation of a physical object with some ability to reason about the object's properties. These types of models allow us to find out how real-world objects might behave under a number of different conditions or requirements. Digital twins have begun to incorporate the transmission of real-time data sensed by the real-world object. This new, higher-resolution sensor data allows the digital twin to reason about future behaviors, then transmit feedback to the physical object. Digital twins create new opportunities for software engineers to use data to develop and assure software systems, but they also create new challenges in scale as digital twins are created for more and more systems in the physical world.

Quantum-enabled systems create new challenges in combining disparate computational models. Software engineering is a challenge for quantum-enabled systems: Advances will be required in many areas, including quantum algorithms, development tools, languages, computing platforms, and testbeds. If we imagine that the hardware advances that permit scaling in quantum computing are achieved, then these and many more advances in software and software engineering will be required as well.

Extended reality provides new opportunities for human interaction and for visualizing complex data and systems. Extended reality refers to augmented reality (AR), virtual reality (VR), and combinations of the two. AR includes the use of devices, such as specialized glasses that display supplementary material, that allow the individual to see the real world, but with augmented information. VR, in contrast, refers to the use of specialized devices that enable a person to see only a virtual world. An essential quality of extended reality is its power to radically reshape humanity's reasoning about information. These technologies could provide new interfaces for software engineers to visualize complex data or systems and enable new interfaces with greater productivity.

An essential quality of extended reality is its power to radically reshape humanity's reasoning about information.



3 Findings

The work that we surveyed for this study points to software engineering as a highly dynamic, fast-moving field where technologies can arise quickly and grow to become integral parts of the infrastructure of modern life. While that is perhaps unsurprising, the extent to which recent technology trends are coming together and allowing the swift emergence of high-quality capabilities is remarkable.

Although this was evident in our literature survey, it became even more apparent as we held workshops and interviews with experts in the field. The following findings summarize key learnings, key challenges, and new research needed for the future of software engineering.

1. Maintaining national software engineering proficiency is a strategic advantage.

Software engineering affects everything, because software is everywhere, including in our nation's infrastructure, defense, financial, education, and healthcare systems. Our ever-growing dependence on software systems makes it imperative to maintain our nation's leadership and strategic advantage in software engineering. We need to raise the visibility of software engineering to the point where it receives the sustained recognition and investment commensurate with its importance to national security and competitiveness.

Software increasingly augments human interactions at ever-larger scales and with ever-greater potential impacts. As our reliance on software increases, improved software engineering technologies are needed that can handle the larger and more complicated systems of the future.

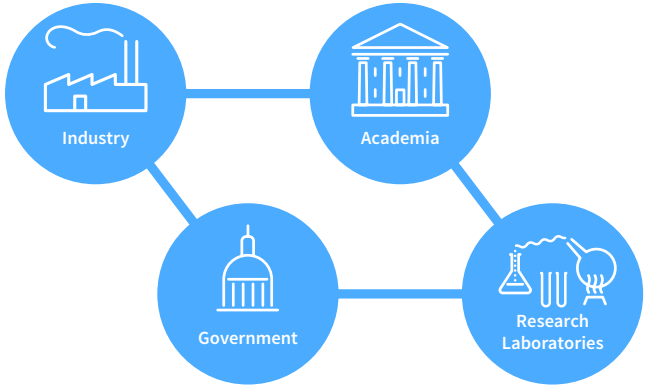
2. Maintaining national software engineering proficiency requires sustained research.

New types of systems will continue to push beyond the bounds of what current software engineering theories, tools, and practices can support. Future systems and fundamental shifts in software engineering require new research focus in areas including smart automation, reassuring evolving systems, and understanding composed systems. New system types, such as AI-enabled systems, societal-scale systems, and quantum systems, also drive the need for new research.

Predictable and pervasive use of AI will also lead to new software engineering principles. Incorporating AI in software systems requires research in AI engineering to enhance the necessary software engineering environments and tools. Incorporating AI also requires an understanding of how AI and non-AI components can work together for overall predictable system behavior. Software engineering tools are a special kind of system, and incorporating AI into these tools will enable more effective software engineering. Once we understand how to do that in a predictable way, it will allow more responsibility to be shifted to AI, and the collaboration between AI and humans will continue to enhance software engineering.

3. Maintaining national software engineering proficiency requires fostering strategic partnerships.

We need to enable strategic partnerships and collaborations to drive innovation in software engineering research among industry, research laboratories, academia, and government. AI into these tools will enable more effective software engineering.



4. Maintaining national software engineering proficiency requires sustained investment.

We must ensure policy makers recognize the benefits of software engineering and make it a critical national capability. Such recognition would imply a sustained investment strategy.

5. The vision of software engineering needs to change.

The current notion of a software development pipeline will be replaced by one where AI and humans collaborate to continuously evolve the system based on programmer intent.

6. Focusing on re-assuring systems will enable continuous and rapid incorporation of new capability.

Because software is ubiquitous, there is an ongoing and increasing need for software to continuously evolve to incorporate new capability. We therefore need to understand how to continuously re-assure software-reliant systems efficiently, without doing harm to existing capability. Elevating the importance of assurance evidence and assurance arguments will be key.

7. New design principles are needed for societal-scale systems.

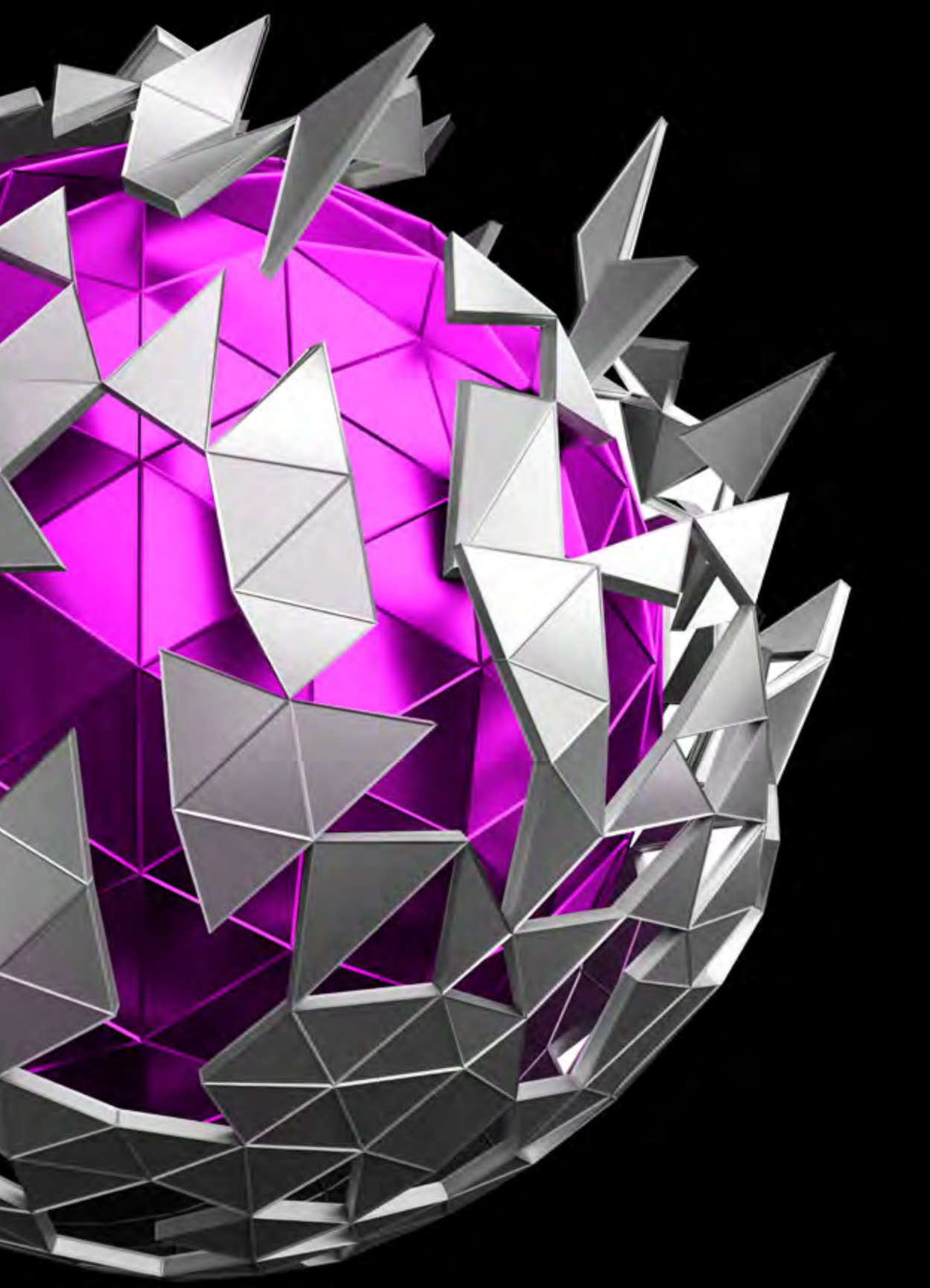
The growing recognition of software's impact is generating new quality attribute requirements for which software engineers will need to develop better design approaches. In addition to traditional ones (such as modifiability, reliability, performance, etc.), there is a need to add a roster of new quality attributes, such as transparency and influence.

Engineering societal-scale systems involves subtle judgments (for example, the appropriate interaction between software systems and free speech principles). One common characteristic of societal-scale systems is that humans are integral components of the system. These systems should provide information or communication channels that can predictably lead to desired outcomes (such as engagement, accuracy, and so forth) from their users. As these systems proliferate, more research is needed to enable such prediction and control of system behavior.

8. The software engineering workforce needs to be (re-)conceived.

Software-reliant systems are built for many different purposes by a broad collection of people with very disparate skill sets, many of whom do not have formal software engineering training. We need to better understand the nature of the needed workforce and what to do to foster its growth.

Society in general has expressed concern about the adequacy and availability of software engineering talent. There appears to be growing concern about a number of topics, including changing technology skillsets, global competition for software engineering talent, and the role of software education. What seems to be clear is that no matter what tools are provided or what level of abstraction we use to construct systems, there will always be an important role for humans to contribute in evolving software engineering.



4 Envisioning the Future of Software Engineering

Imagine it's 2035. What will software engineering look like? Perhaps we can imagine it as more of a technical conversation between humans and computers than a process of manually refining specifications and code.

4.1 Future Scenarios

Consider this scenario: The days of endless requirements and design reviews are gone. A joint team of aeronautical engineers, pilots, and software engineers together design the next space-capable craft by pitching ideas, which are turned into viable designs based on access to extensive codified knowledge about cyber-physical systems, as well as the limitations of physics. These designs are displayed in real time, and the team compares defensive and maneuverability capabilities on the fly using real-time

simulations of representative missions. The final design is selected based on the most desirable balance of cost, capabilities, and timeline. Today's notion of a software development lifecycle might seem almost archaic compared to this fluid, iterative process.

Developing software in the future is likely to become more about expressing desired capability than writing code or having a mental repository of algorithms. Software engineers will have to become adept at expressing intent in a way that readily enables the computer to learn from experience. "Elegant software" will no longer refer to clever code, but will rather be the

result as humans work with automated and AI systems to implement the best ideas humans can imagine in the most timely, affordable, ethical, and secure ways.

Who can "program" and create complex systems will naturally expand as well. Our conversations with computers will take place in the language of our domains, with computational biologists, for example, developing software capabilities by talking about sequencing and genes, not by learning Python. Specialists of all types will be needed to inform the computer properly, and how they interact will look significantly different than it does today.

The use of simulation may turn today's entire notion of test and evaluation into an immersive experience. Imagine that a new hardware configuration and software capabilities are planned for a series of space assets. In a fully immersive virtual reality environment, the changes are emulated with the full telemetry of the current assets feeding the environment. Engineers can view the new space configuration from any vantage point, and not only in a visual range. All the available data and metadata from the current environment is also presented in real time. Where the desired effect is not what was anticipated, the engineer makes changes and immediately sees the impact on the holistic space environment. Moreover, dozens or more additional engineers are observing and manipulating the same environment in a shared experience. Communication between the engineers, enabled by many types of media, and a shared decision process assure that the system as a whole has no unintended or undesired emergent behavior. This same environment will be used once the change is made to support operator training and real-time mission rehearsal.

Once deployed, systems will also be much more adaptable and integrated. Consider a scenario that involves a special forces team on a deployment, and imagine a firefight breaks out. The squad is caught off guard, communications have been disrupted, and they're unsure of the weapons being used against them. Fortunately, they are teaming with a set of micro unmanned aircraft systems that proactively set up a mesh network

using alternate communications channels to re-establish contact with headquarters. Once that network is established, the squad directs the devices to observe and profile the weapons on the battlefield covertly and provide mitigation options while they take cover. As a result, they are not only able to overcome the novel threat locally but also feed their real-time experience to other units at the tactical edge that could be at risk. To make this scenario a reality, software engineers will need to design architectures that are nimble and allow adjustments to systems based on data from operational sensors and other input from users in the field.

Animators designing the next movie might work differently as adaptive user interfaces expand their capabilities. No longer expected to know coding and scripting, they can take their creative design skills much further as they develop fully immersive movies. In the "Age of the Holodeck," they are able to incorporate novel visual storylines, design clothes for next-generation haptic feedback, and create events that react to the viewer's input. As plot possibilities are explored over time, the interactive experiences evolve and improve to tailor to the preferences of the participants, building on the intent of the artists that set it in motion.

Students of software engineering will also learn differently. The AI and automation that help them craft their code might be largely invisible as they focus on their primary job: learning how to best understand and express what the software should (and should not) do. What is their intent for the program? How can they be sure this intent will be carried out over time? Is the end result not only functional, but also evolvable, intuitive for users, and trustworthy? One example project might involve a personalized web "browser" that tailors itself to an individual's needs while also self-updating robust layers of security to protect critical data. The same system could still cultivate and share appropriate information anonymously in real time to help other students working on similar projects.

As software engineers move into the workforce, they will no longer be required to try to understand the ripple effects brought about by changes in increasingly complex systems. As the systems are evolved by both humans and AI assistants, problems will be identified and corrected before implementation. Assuring that proposed changes won't break the system will be done automatically by analyzing the effects a change might have on a system's underlying and evolving assurance arguments, which have been designed by skilled software engineers. Conformance to quality standards will also be guaranteed by design, as part of sophisticated software development frameworks that are put in place by expert engineers, yet remain hidden from programmers who need not be concerned with those aspects of the design.

Despite these advances in software engineering, complex systems of any kind are unlikely to be perfect. In the future, specialized disciplines may emerge, such as those that detect potential system problems, recover capabilities when failures occur, and discover and eliminate the causes. For example, a forensic software engineer might join a virtual meeting with colleagues all over the world to analyze an exploit in a client's security code and determine what impact it may have had. Another engineer working with a socio-technical ecosystem might be called by the system itself to step in if it notices the general expression of sentiment is moving toward an undesirable extreme. When problems such as these are detected, it's the job of specialized software engineers to discover the root causes. Was it a weakly-trained machine learning algorithm, a poorly-expressed intent, a component that was allowed to violate the architecture because it did not account for a particular variable, or something else?

In all these scenarios, software engineering is everywhere, although it looks and acts different than it does today. It enables all the capabilities described, and it does so securely, predictably, and affordably. Future advances in research will enable a diverse set of people to have broader access to

creative development, but software engineering is what will ensure the systems they create have superior capabilities, yet are largely free of the problems and failures we see today.

4.2 Vision for the Future of Software Engineering

While the exact roles that intelligent algorithms and humans will have remains to be determined, the importance of software engineering to our vision is clear: Humans and AI will be trustworthy collaborators that rapidly evolve systems based on programmer intent.

As software engineers continually interact with smart software assistants, computers and humans will be able to do what they both do best. Working in this way, possibilities that we cannot even imagine today will become reality. The research in this report provides the essential groundwork for advancing the discipline of software engineering to ensure that the necessary framework is in place to maximize the advantages these future opportunities can provide.

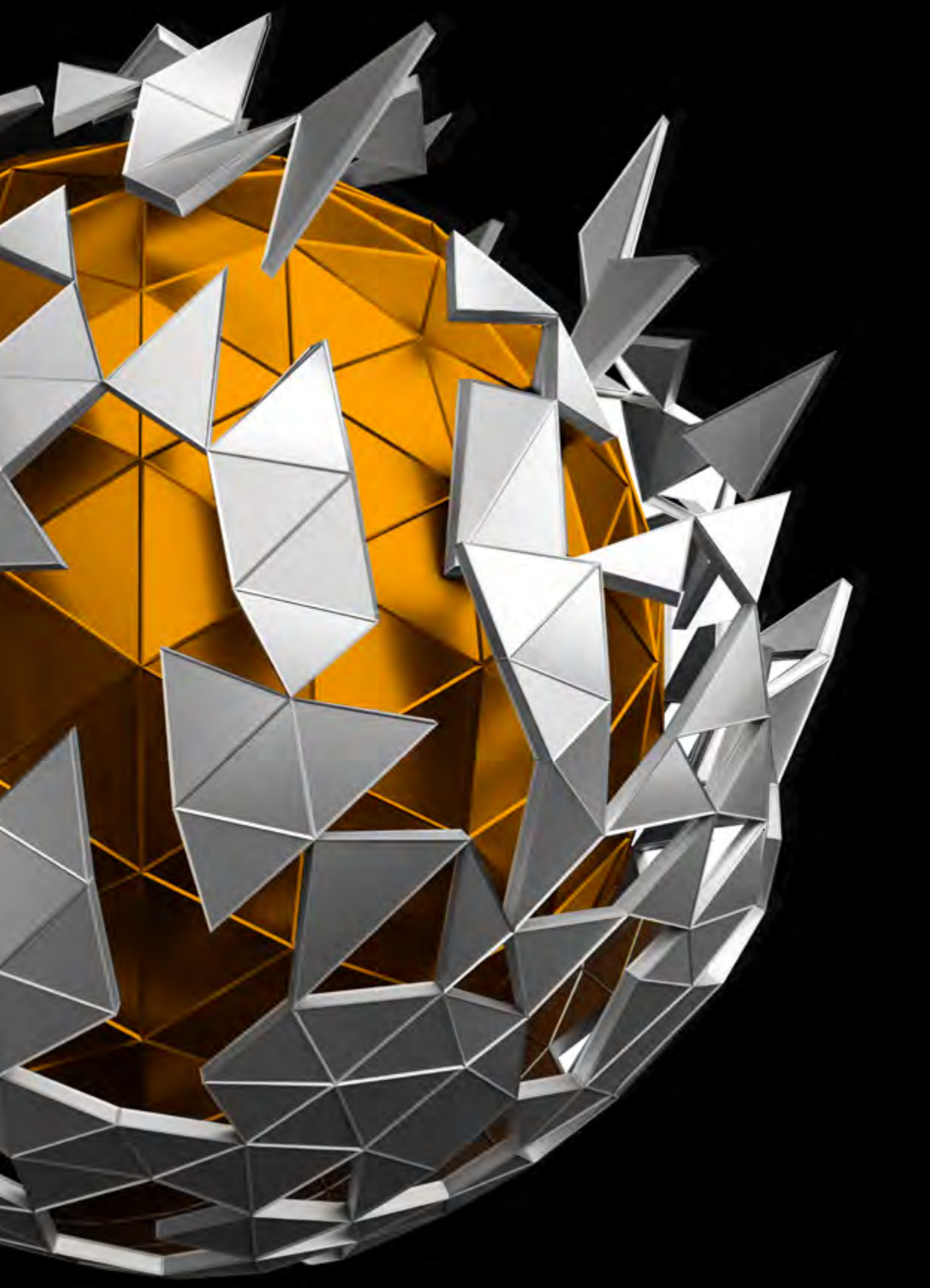
A new vision for software engineering requires new development and architectural paradigms, which also motivate the research focus areas described in Section 5.

Advanced development paradigms, such as the following, will lead to efficiency and trust at scale:

- Humans leverage trusted AI as a workforce multiplier for all aspects of software creation.
- Formal assurance arguments are evolved to assure and efficiently reassure continuously evolving software.
- Advanced software composition mechanisms enable predictable construction of systems at increasingly large scale.

Advanced architectural paradigms will enable the predictable use of new computational models, as described below:

- Theories and techniques drawn from the behavioral sciences are used to design large-scale socio-technical systems, leading to predictable social outcomes.
- AI and non-AI components interact in predictable ways to achieve enhanced mission, societal, and business goals.
- New analysis and design methods facilitate the development of quantum-enabled systems.



5 Research Focus Areas

The fundamental shifts and needed advances in software engineering described in this report require new areas of research. In close collaboration with our advisory board and other leaders in the software engineering research community, we developed a research roadmap with six research focus areas.

This section describes the motivation for these six areas, which are closely related to the findings in the previous section. In this section we also provide a complete roadmap, followed by a discussion of each research focus area in depth.

5.1 Advanced Development Paradigms

Trends toward the use of DevSecOps and digital twins are indicators that the boundaries between fielded systems and development environments are increasingly becoming porous. For example, through maturing AI techniques, software is augmenting human decision making and becoming very useful as a vehicle for improving performance by learning from experience (i.e., data). Operational data is being combined with simulations to give real-time insight into how systems behave. Continually changing mission needs are driving almost continuous system evolution, requiring both efficient system re-assurance and compositional approaches to system development. These trends motivated the first three research focus areas, which we consider fundamental to advanced development paradigms:

1. **AI-Augmented Software Development:** using maturing AI techniques to augment human decision making in software engineering and to enable learning from the vast amount of software engineering data
2. **Assuring Continuously Evolving Software Systems:** recognizing the importance of efficient re-assurance of rapidly changing systems while taking into consideration the many scientific domains and evidence that will be needed to reason about future software-reliant systems
3. **Software Construction through Compositional Correctness:** recognizing that the only viable way of developing and evolving systems will be through technologies that enable compositional development

5.2 Advanced Architectural Paradigms

Some characteristics of future systems pose new and interesting problems for software engineering. In particular, introducing AI components into systems, considering humans as elements of a system, and effectively exploiting quantum computing pose particularly important challenges for future systems. In our vision of software engineering, advanced architectural paradigms will enable the predictable use of these new aspects of systems. These challenges motivated the following three research focus areas that we consider fundamental to advanced architectural paradigms:

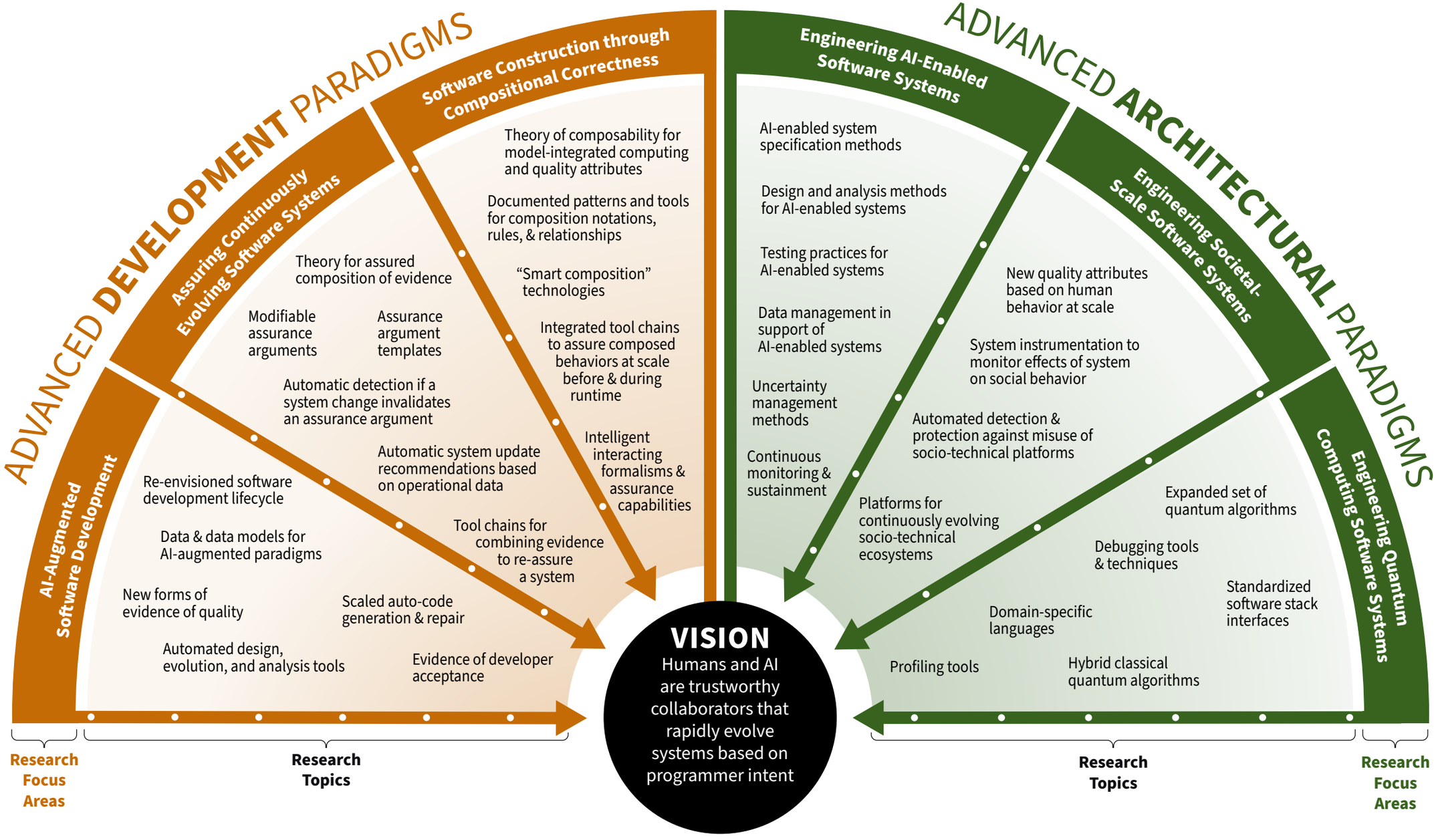
4. **Engineering Societal-Scale Software Systems:** discussing the challenge of modeling human behavior
5. **Engineering AI-Enabled Software Systems:** focusing on the challenge of handling the uncertainty that AI components bring to a system
6. **Engineering Quantum Computing Software Systems:** considering what aspects of the quantum computation should be hidden from or exposed to higher levels of the software stack

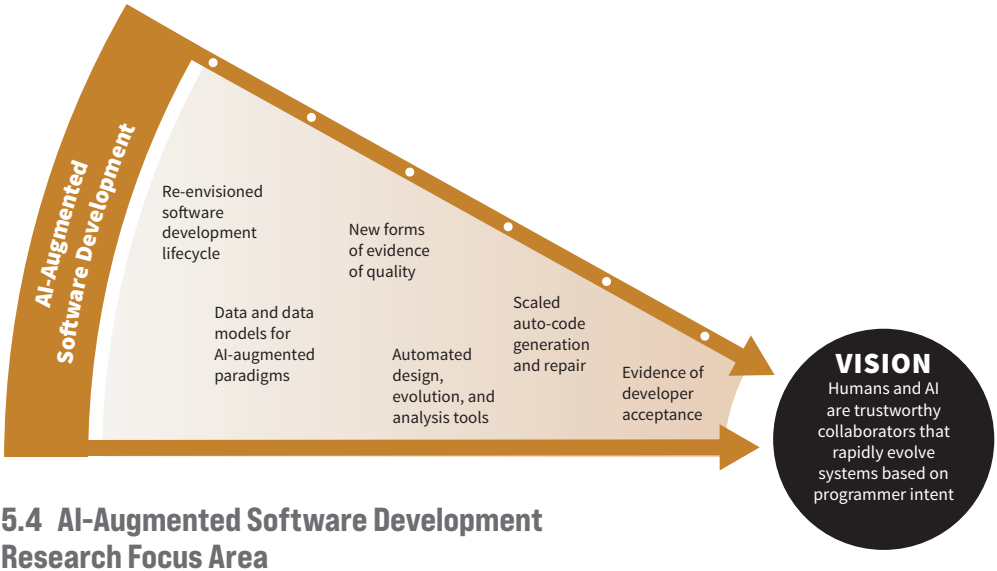
5.3 Research Roadmap

The research areas we identified are meant to be mutually synergistic. For example, AI-augmented software development needs to consider continuously evolving systems and, ultimately, assurance arguments need to be used by AI tools when they offer software development advice. The relationship between software construction through compositional correctness and assuring continuously evolving software systems is also strong, because compositional technology and reasoning will be key enablers of incremental re-assurance as systems evolve. Likewise, all of the advanced development paradigms are applicable to each of the new system types discussed under advanced architectural paradigms.

The graphic on the foldout following this page shows the research focus areas and a suggested course of research topics to undertake.

Software Engineering Research Roadmap with Focus Areas and Research Objectives (10–15 Year Horizon)





5.4 AI-Augmented Software Development Research Focus Area

5.4.1 Goals

The need to improve the efficiency of software engineers and reduce their cognitive load has driven and will continue to drive trends toward improved automation and formalisms to support software development tasks. Software engineers using AI-augmented approaches will also be able to focus on tasks that require critical thinking and creativity. This research area focuses on developing approaches for automating AI-relevant software engineering tasks and accelerating the development of reliable automation for engineering. Outcomes of this research area will consequently enable the design, development, and deployment of reliable software by further shifting the attention of humans to the conceptual tasks that computers are not good at and eliminating human error from tasks where computers can help.

Many of the automated approaches developed during the previous decade, including model-based software engineering, DevSecOps tools, defect and vulnerability analysis, automated bug fixing, modern code review, and value stream management tools, were developed with the goal of improving software development efficiency and quality [Lago 2015; Rahman 2017; Morrison 2018; Le Goeus 2019; Sadowski 2018; Murphy 2019]. Despite these advances in automation, failures, software security and quality issues, and overspending continue to be the norm. To put the size of the challenge in context, the U.S. government alone (excluding any private industry spending) spent over $90 billion for system maintenance and operation in 2019 [GAO 2019].

We need to create tools that allow software engineers to easily express the changes they care about, including requirement and design trade-offs and different solution options, and then trust that automation will correctly resolve most, if not all, of the details at the programming language level. For example, many systems would benefit from the development of tools to help developers avoid, detect, and fix defects as they develop software. A range of techniques that includes safer programming languages, better-designed frameworks, cheap and easy automatically generated tests, and tools that recommend fixes will collectively provide better results than relying on any one technique alone. In the next decade, AI approaches will provide an opportunity to rethink how we achieve programming goals, in particular by providing improved capabilities for the elimination of trivial and repetitive mistakes that later become hard to detect and fix.

These advances will inevitably drive a re-envisioning of the software development process, with increased intelligence and support to developers. Taking advantage of the data generated through the software development lifecycle will be a beneficial and natural byproduct of the process. Consequently, this research area asks the question: What will AI-augmented software development look like in the future?

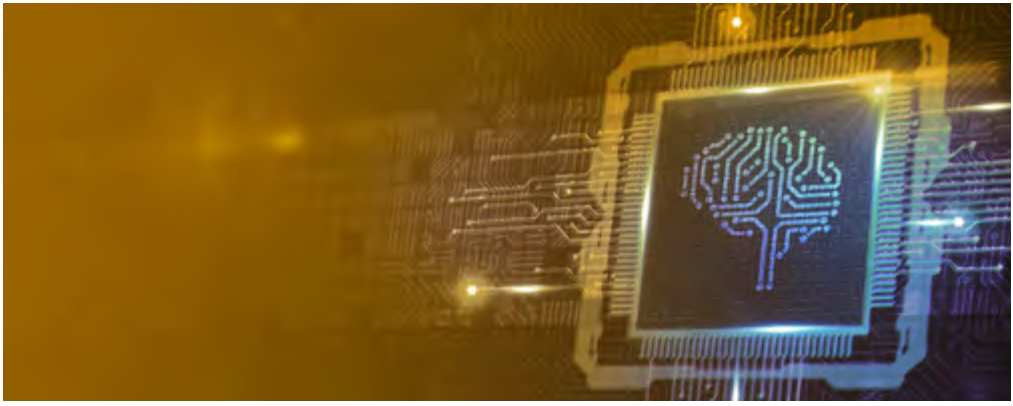
5.4.2 Limitations of Current Practice

Developers are expected to understand the ripple effects within increasingly complex systems without having effective tools.

Today, software development is human-intensive, test-intensive, and error prone. In particular, current software development practices are hindered by the following limitations:

- Developers are expected to be experts in many topics (requirements, architecture, design, programming languages, analytic models, a dizzying array of technologies and frameworks, quality attributes, testing approaches, platforms, and much more). Software engineers often naively rely on software development processes that are not followed properly to orchestrate these activities and the artifacts created along the way.
- From inception to deployment, a significant number of artifacts are generated from requirements specifications, such as design documents, analysis artifacts, test cases, and deployment scripts. Streamlining these artifacts toward successful system delivery continues to be a resource-intensive challenge.
- Developers are expected to understand the ripple effects within increasingly complex systems (in terms of size, distribution, concurrency, etc.) without having effective tools.
- Formal methods and model-based approaches have been created with the promise of reliably generating code and evolving systems, but even in safety-critical systems they do not scale beyond limited aspects of the system.

- Time spent designing and testing systems continues to be cut short when schedule challenges hit, further jeopardizing the quality of the systems developed.
- System sustainability and evolution, especially for legacy systems, continue to be manually driven, high-risk efforts.
- Conformance to quality standards and intended architectures are not guaranteed as part of the software development framework or tool chain.



5.4.3 Topics for Research

At almost every stage of software development, AI holds the promise of assisting humans and making the process more efficient, effective, and enjoyable. Each new generation of tools and advances toward this end will find acceptance by developers and reach wider adoption if it can meet the following goals:

- Perform tasks that developers already do, but do them more efficiently (e.g., test faster).
- Perform tasks that developers already do, but do them better (e.g., catch more bugs).
- Perform tasks that developers are not able to do currently (e.g., leverage new data to integrate new conformance checks or generate new tests).
- Reduce hand-offs and integrate elements that are currently disconnected (e.g., provide requirement traceability).
- Teach developers how to do tasks better as they go (e.g., advise and/or mentor with real-time feedback on implementation errors).
- Help to scale what developers can already do (e.g., allow them to consider more alternative design options).

To best evaluate the efficacy of such new, automated tools and approaches in practice, they also need to seamlessly augment and integrate with the developers' environments, even during the research stages. The AI-augmented software engineering advances that we outline in the following sections take advantage of continuous integration and deployment environments for data collection, iterative feedback loops, and testing developer buy-in.

5.4.3.1 AI-Supported Re-Envisioned Development Workflows

Agile and lean software development processes encourage elimination of waste by helping developers focus on the top priorities and understand what tasks stay in inventory [Reinertson 2019]. For example, test-driven development workflows might advocate software requirements being converted to test cases before the software is fully developed. Then software development can be tracked by repeatedly testing the software against all the test cases, which will drive significant improvements in the efficiency of software development and improve system quality. In a similar analogy, AI-supported development workflows will target data-intensive and tedious activities, which might result in different task dependencies in the the software development lifecycle. For example, developers may not need to test for certain classes of bugs when AI-augmented bug fixing becomes a reality at scale.

Incorporating AI-based developer support tools will also trigger the need to envision more effective workflows for developers. For example, improved real-time assistance in code quality conformance can reduce reliance on the added static code analysis checks during testing and deployment, improving the balance of local conformance analysis versus analysis of system-wide, cross-cutting, and harder-to-conduct architectural quality and runtime concerns. Today, system-wide analysis is often a manual effort that is rarely supported by automation.

In an AI-augmented development lifecycle, the developers and the AI assistant will both have a supervisory role. Developers will guide and consequently improve the AI assistants. AI assistants will take on a supervisory role by providing real-time feedback and, in time, demonstrating repeated mistakes to developers. On a developer team, there will always be some developers who you trust more than others (perhaps due to experience, skill sets, or demonstrated performance). The AI-assisted development workflows will trigger the need to think of AI "partners" in the same way. In what roles do humans and AI perform most effectively as part of an overall team that produces software of sufficient quality?

5.4.3.2 Automated Code Repair

In what roles do humans and AI perform most effectively as part of an overall team that produces software of sufficient quality?

Automated code repair is a specialized application of auto code generation. Code generation includes a broad collection of approaches and technology that accomplish different tasks, and the precise meaning depends a lot on what portion of a code base you are discussing. Code generation has been available for decades to generate portions of code, such as class declarations to match a unified modeling language (UML) diagram. Such work is usually used as a starting point, with developers expected to take over implementation details within this shell. There has been fragmented research on several formalisms to drive code generation. Similarly, existing computer-aided software engineering tools have limited code generation capabilities.

The opportunity that AI poses for code generation is the ability to search existing code and identify patterns that are similar to the intended new code. Rather than starting from a specification, such as a UML model, existing code can be used as an input and transformed to address specific classes of problems. ML-based techniques can assist by generating similar code to what is already available. This suggests opportunities for starting small using AI-based code generation approaches to take advantage of commonly repeated applications.

Automated code repair can similarly use existing code as input and then transform it to address specific classes of repair problems [Oliveira 2018; Klieber 2016]. The research gaps to fill include figuring out which parts the user needs to specify for the program to generate usable solutions at scale and how to generate partial, but acceptable, solutions to make incremental progress. Other gaps involve determining whether code bases provide relevant data to label and create ML models for complex code generation tasks (in particular, those that can also resolve semantic inconsistencies) and deciding how developers can determine whether what was generated is appropriate.

The chances of human error increase with the exponential increase in the volume of code and other software artifact data generated; the number of overlooked bugs also increases simultaneously. AI, and in particular ML, is good at recognizing patterns in huge amounts of data. Success will depend on the ability to identify small, scalable portions of auto code generation and repair problems [Kirbas 2021].

5.4.3.3 Eliminating the Design/Code Conformance Gap

Aligning the design of a system and its implementation improves product quality and simplifies product evolution. Especially for DoD systems, the ability to conform to particular architectural approaches, such as open architectures, not only supports technical goals, such as the ability to evolve easily, but also mission goals, such as the ability to integrate across domains. The same challenges exist in industry systems as well,

such as in smart cities. Today, such conformance goals are achieved through qualitative techniques. While developers are empowered with AI-augmented tools and techniques that increasingly assist them in implementation tasks, the abstraction gap between code and design limits automation for conformance and design tasks. Code all too often diverges from a system's intended design and qualities designed into the system are not realized, increasing maintenance effort. This could be prevented if developers could confirm whether each code commit conformed to the intended design and quality. To accomplish this in a scalable way, automation is needed that can compare as-implemented designs to as-intended designs as part of continuous integration pipelines. ML and search-based algorithms provide an opportunity to revisit this otherwise thorny problem.

5.4.3.4 Multi-Artifact System Analysis

There are opportunities—and dangers—in relying on AI assistants to help developers find related examples in artifacts, code bases, defect and vulnerability records, or documentation recommendations. For example, many developers have copied insecure code, propagating bad practices. AI assistants, however, when proactively designed to safeguard for such errors, can contribute to reducing the risks posed by such practices. For example, recommendations can incorporate features from multiple artifacts, along with confidence ratings to improve relevance. AI assistants that support multi-artifact analysis, for example, can support engineering tasks during the planning phase, when a project can take advantage of inputs from requirements, analysis, and technology selection to improve contextualization of reused code from other projects.

The ability to combine information from multiple software engineering artifacts, supported by ML approaches such as natural language processing algorithms and boosting algorithms that turn weak learners into strong learners, will open improved avenues of traceability in the software development lifecycle. For example, if a developer is describing something they are not that certain about, the system can continuously rebuild aspects of itself based on human or AI intervention until the description becomes clearer. In so doing, it can take advantage of features potentially common to different software engineering artifacts that drive the development of such learners.

The body of work in mining software repositories has progressed over decades [Hassan 2008; MSR 2021], in particular in areas of defect analysis and prediction, developer sentiment analysis, and code review practice. This body of work provides a foundation to improve the traceability of information shared within the artifacts, driving more timely and correct decision-making support to developers.

5.4.3.5 Automated Evolution and Refactoring

Software design, development, and deployment is a trade-off-based activity. Search-based software engineering techniques [Harman 2015] show promise of accommodating the Pareto-optimal nature of software design and, in fact, recommend multiple viable solutions representative of the design trade-off space. Despite progress in providing software engineers with tools that automate an increasing number of development tasks, complex activities—such as redesigning and reengineering existing software—remain resource-intensive or are supported by tools that are error prone.

The vision for automated evolution and refactoring at scale calls for automation that takes direction from developers in the language of design and automates the code changes required to realize those changes [Ivers 2020]. Questions to address include what portions of the process can and should be automated and how AI approaches, such as search-based algorithms, can reliably generate solutions. Automating tedious, repetitive, and error prone activities (e.g., crawling through thousands of dependencies) is a clear starting point. Should tools change the bare minimum to achieve a refactoring goal, or should they create opportunities to "clean things up" along the way to improve the system along general quality improvement goals (e.g., introducing design patterns to improve maintainability along with other top-priority quality concerns)? Can we refactor data (e.g., database schema and stored procedures) together with code? Tools should refactor test suites along with code. Can they also create unit tests as a byproduct of refactoring at scale? Addressing these questions will enable a future in which AI-augmented software development will also be able to support more high-speed change.

The vision for automated evolution and refactoring at scale calls for automation that takes direction from developers in the language of design and automates the code changes required to realize those changes.

5.4.4 Research Questions

Questions derived from the research topics include the following:

- How do we transform high-level specifications of what the program should do into a low-level program that implements it (i.e., use abstract logical reasoning to create programs that satisfy a given specification)?
- In what ways can AI be used to make specifications more precise and resolve ambiguities, consistent with the needs of system stakeholders? Are developers more or less error prone in writing specifications than in writing code, and where can AI supplement gaps in both writing code and specifications?
- Are there new phases or activities that become part of the software development lifecycle in an AI-assisted paradigm? Do we need to rethink aspects of the lifecycle to incorporate the elicitation of intent? Can the feedback obtained from AI-assistants be purposed for fostering the skills of developers?
- Can AI-assistants help developers orchestrate continuous system evolution, since software will not be static and hard-coded as it is today but rather will dynamically adjust to continuously fulfill its purpose?
- What roles will AI-supported tools take on and what roles will be retained by human developers?
- How can AI-augmented software development tools generate the metadata needed to efficiently verify or validate code?
- Can AI capabilities support debugging code by instrumenting the development environment (e.g., can automated analysis look at issue logs in a critical system and understand how long it is taking to address certain types of problems, then feed that forward to help with decision making)?
- What are the right levels of abstraction? We have developed higher-level programming languages, architecture patterns, and so forth. What new levels of abstraction are needed to support different developer tasks?
- What new software development data needs to be collected (ethically and in a way that ensures security and privacy) to enable such future research? What activities lack the traceability that could be established if we collected such data?

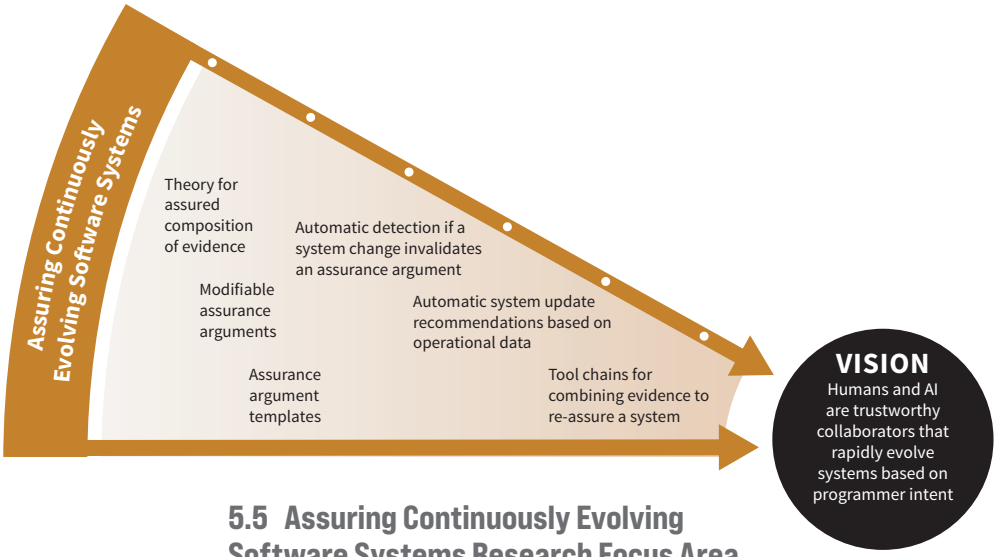
Trust in AI-based solutions, whatever their form, boils down to a risk assessment. What's the probability that the result is correct? What's the impact if it's wrong, and how can the associated risks be dramatically reduced if AI makes the wrong recommendations? Developers will initially have to take the driver's seat, and initial progress will be slow; however, advances in answering such questions will accelerate progress.

AI-augmented software development research, consequently, will result in creating reliable automation support for developers, improving their efficiency, effectiveness, and the quality of the systems they develop.

5.4.5 Research Topics

To make progress in these areas, each area must be able to inform the others, in particular by generating data and using that data to identify the next relevant research steps. Progress will be iterative and incremental, with the following milestones guiding success:

- *Re-envisioned software development lifecycle.* The way that the software development lifecycle will change in an AI-augmented paradigm needs to be considered. As research progresses, it is important to clarify the different roles that humans and AI-augmented tools perform, ranging from AI as a trustworthy assistant to AI completely replacing some tasks.
- *Acquire data and developed data models for AI-augmented paradigms.* Data to model each stage or workflow of an AI-augmented paradigm are needed. An assessment is needed to identify what additional data would help, how to collect missing data with the least intervention, and how to assemble this data ethically.
- *Identify new forms of evidence of quality.* There is a need to automatically accumulate and carry along evidence of quality and to verify that the results are correct. AI generates metadata to efficiently verify or validate code and generate traceable evidence with code.
- *Automate design, evolution, and analysis tools.* Reliable automated tools that interact with developers using their vocabulary are needed to assist with evolution and refactoring tasks.
- *Scale auto code generation and repair.* Model-based techniques and formal methods need to be augmented with AI techniques to increase the scope and scale of their applicability.
- *Collect evidence demonstrating developer acceptance and efficacy of AI assistance.* Developers must accept the new forms of interactions with confidence; empirical data demonstrates that developers spend the most time on design tasks rather than on software complexity and challenges to improved quality.



5.5 Assuring Continuously Evolving Software Systems Research Focus Area

5.5.1 Goals

This research area focuses on how to create an evolvable assurance argument, which should only require incremental changes as incremental changes are made to a system. Incremental modifications to a system should also require less assurance than a complete de novo assurance effort. In addition, the argument should be provably sufficient (i.e., the only source of uncertainty in the argument should be deficiencies in the evidence—the reasoning structure should be completely deductive, providing correct conclusions to the extent that the evidence and inferences from evidence are completely correct).

5.5.2 Limitations of Current Practice

The scope, scale, and pervasiveness of software-reliant systems continue to grow and change. Incremental assurance is the only practical way to get a system deployed and updated in a timely manner with adequate confidence in its operational behavior. Providing such confidence is particularly important for safety- and mission-critical systems in which software defects can have catastrophic impacts on lives or property.

Systems today change incrementally and continuously. It is necessary to move from a mindset of "build and assure" (i.e., a one-time event) to a mindset of "rebuild and re-assure" (i.e., continual events). Moreover, today's software-reliant systems evolve, in part, by reusing components of varying provenance in settings that were not necessarily considered and where third parties are responsible for making updates. Therefore, addressing limitations for assured composition of components and on efficient re-assurance of evolving systems is important.

5.5.2.1 Limitations in Assured Composition

Software-reliant systems are typically built today by incorporating third-party components, mostly in the form of platforms (e.g., operating systems, middleware, and graphical user interfaces), libraries, and frameworks (that combine platforms and libraries). To allow developers to use third-party components, component developers provide application programming interfaces (APIs) that describe information, such as

- how to call the functions of the components (e.g., name of the function and what parameters to give)
- what results to expect in return (e.g., the specific output value type and high-level description of the computation)
- what restrictions (if any) govern the sequencing of different functions (e.g., to open a file before reading it)

Unfortunately, while APIs are often sufficient to perform code composition, they are insufficient to perform assured composition. The reason APIs are insufficient for this purpose is that they are designed for use by programmers rather than for integration into assurance arguments.

5.5.2.2 Limitations in Reassuring Evolving Systems

Today's approaches for assuring software-reliant systems focus primarily on testing, simulation, and (limited) application of formal methods. Most assurance in practice is accomplished by testing (e.g., automated regression testing, stress testing, and penetration testing) and inspection (e.g., design and code reviews).

For some enterprise and mission-critical systems, Agile development and DevSecOps pipelines increasingly provide the basis for incremental evolution. In these types of systems, it is common to use techniques that provide quick feedback about possible errors. Various informal assurance techniques are used, including

- test-driven development, which ensures that tests exist for new and modified capabilities
- continuous integration, which provides the opportunity to make and test frequent small changes, thereby giving incremental increases in confidence that system requirements are being met
- continuous delivery, which ensures quick feedback on whether incremental changes are meeting actual user needs (i.e., whether the requirements need to change)

In all of these methods, quality assurance engineers leverage regression testing to gain evidence that previously assured system behavior is maintained despite changes incorporated in a new release.

Assurance cases are increasingly being required for assuring critical systems [Maksimov 2019; Denney 2018]. The assurance case concept, however, focuses on arguing the properties of a system at a particular point in time. The notion of evolving an assurance argument in parallel with system evolution activities is not commonly discussed, even in large research projects such as DARPA's ARCoS project (which is focused on developing assurance cases more quickly by automating some parts of the process) [Richards 2019].

5.5.3 Topics for Research

5.5.3.1 Developing a Theory of Assured Software Evolution

Software-reliant systems are neither static nor infrequently updated engineering artifacts, but rather are fluid engineering artifacts (i.e., artifacts that are expected to undergo continuing updates and improvements). Therefore, the effort required to re-assure and recertify an entire system must be minimized by bounding the consequences of changing individual components and subsets of composed components.

The overarching goal of this research topic is, therefore, to enable efficient and bounded re-assurance of continuously evolving systems where most, if not all, of the re-assurance effort is confined to the part or aspect of the system that changed. Achieving this goal requires developing a theory and practice of assured software evolution, including how to appropriately structure a system and its assurance argument. Central to the theory is an artifact that we will refer to as the "assurance argument," which should possess the following properties:

- Precision. The representation is amenable to automatic and formal reasoning.
- Soundness. A viable assurance argument must be based on explicit and precise assumptions and be provably sufficient relative to claims about the behavior the system must exhibit.
- Multi-domain applicability. A viable assurance argument consists of a large collection of interacting sub-arguments (e.g., about logical correctness, timing correctness, and security), each of which must be precise and sound, while accounting for interactions with other sub-arguments.
- Modularity. An assurance argument that supports efficient re-assurance must bound the effects of change to avoid having to re-assure all behavior every time the system changes. But in addition, inter-module dependencies must be made explicit and traceable.

The next two sections discuss topics focused on overcoming limitations associated with assured composition and efficient reassuring of evolving systems.

5.5.3.2 Toward an Ecosystem Architecture and Proof System for Argument Sufficiency

It is hard today to explain to what extent the combination of different types of evidence (such as from testing, formal verification, simulation, and digital twins) provides increased (and justified) confidence that a system will meet its requirements (i.e., behave as expected). It is likewise hard to explain when certain types of (potentially expensive) evidence are not needed to obtain sufficient confidence in system behavior. Justified confidence depends on developing and sustaining a well-structured assurance argument that rigorously codifies how system behavior can be inferred from a collection of evidence. To address this challenge, we propose research on creating or refining theories, methods, and tools for combining disparate evidence types into a sound assurance argument.

While we are motivated by the notion of an assurance argument actually being a proof system, we are also aware that there are no irrefutable arguments for practical systems. In particular, there is no absolute certainty that an assumption wasn't overlooked or that an inference from every evidence type is always correct. Nevertheless, we believe that the quest for sound arguments, while being mindful of important practical considerations, will lead to rigorous—yet practical—techniques.

5.5.3.3 Combining Disparate Evidence

An assurance argument uses evidence of different types. Consider the need to assure that an autonomous system (such as driverless car or drone) will never hurt humans. One type of evidence might come from real-time analysis that can be applied to guarantee that deadlines for timing-critical tasks are never missed. Another type of evidence might be required to estimate the probability of the vision system correctly recognizing humans. These different types of evidence—in addition to other evidence—must be combined in ways that enable making a formal assurance argument about human safety. Making a formal assurance argument using evidence from different domains also requires combining proof systems from the different domains. This leads to two challenges: (1) determining whether there are dependencies between the proof systems, and if there are dependencies, (2) determining what parts of the original proof systems remain valid and what parts need to change.

The assurance argument must provide the appropriate guarantees, even if empirical evidence is part of the argument. For example, in an airbag system we may determine that it is not possible to guarantee that an adult-detection module always detects whether an adult passenger is present. In that case, it might be possible to determine that the probability of not detecting a human should be kept under 0.001%. A sufficient argument for achieving this level of precision would require designing experiments with proper sampling mechanisms based on probability theory to prove that the required probability-of-failure threshold can be reached. A sufficient argument would also need to account for the possible mutual effects of using probabilistic and deterministic reasoning in the same assurance argument.

5.5.3.4 Assured Composition

Assured composition will be key for combining disparate evidence and producing rigorous assurance arguments. Achieving assured composition requires expanding attention from code composition to assurance argument composition. Discussing both programming and assurance concerns requires using a more general definition of components and interfaces.

We define a component as an encapsulation of a behavior that provides a summary of such a behavior in an interface. This interface is sufficient to combine components and create a new behavior. The specifics of the behavior encapsulation and its summary depend on the type of composition. For example, code composition will use code encapsulations (e.g., libraries) and APIs, while assured compositions may use, for example, value transformation formulas and post- and pre-conditions. Related considerations include

- *Software composition is heavily focused on code composition in current practice,* which means that component interfaces focus on providing information to programmers with little direct concern for assurance. Some efforts have gone beyond this conventional approach by incorporating some form of assumed and/or guarantee reasoning, such as design-by-contracts schemes [Myer 1992].
- *Formal arguments are typically developed for particular domains* without considering cross-domain effects. This trend has led to component and interface descriptions that only support, for instance, value transformation claims and ignore other aspects, such as timing, security, and the effects of faulty behavior. Other communities focus on, for instance, timing behavior and ignore value transformations. Such domain-specific interface descriptions capture only part of the complete behavior of the component. These domain-specific claims are actually key to scalable assurance within a specific domain. For instance, the real-time task abstractions in rate-monotonic analysis (which is a specific type of real-time analysis that only uses task execution time, period, and priority in the simplest case) can lead to analyses that are of linear complexity in the number of real-time tasks. Adding additional information, such as computational state, can lead to exponential analysis complexity (as for a timed automaton, for example) [Alur 1992].
- *Cross-domain dependencies.* When a claim is made in a particular domain (e.g., to verify value transformation properties), modification to this claim can affect claims in different domains (e.g., claims about the worst-case execution time of a task, the heat dissipation of a processor, or failure independence in a fault-tolerant replication structure). These effects stem from cross-domain dependencies, which are often ignored or treated informally.

- *Scalable multi-domain compositions.* To deal with the cross-domain dependencies, some research efforts have merged multiple domains to address different cross-domain concerns simultaneously [Alur 1992, 1993]. Unfortunately, these efforts tend to lead to unscalable assurance procedures. Consequently, the challenge is to preserve the scalability of independent domains while accounting for the cross-domain dependencies in a sound manner. Work by Chaki et al. achieves scalable assurance of concurrent real-time systems in two domains: concurrent systems and real-time schedulability [Chaki 2011]. Exploiting assumptions of real-time schedulability analysis (specifically, the periodicity of job arrivals) greatly reduced the number of possible process interleaving that needed to be checked in periodic concurrent systems.

Experience gained from observing, experimenting and a sound assurance argumentation system should be capable of automatically proving that the provided arguments are sufficient to demonstrate that the system behaves as intended. patterns can be used to trigger re-assurance or bound re-assurance

5.5.3.5 Argument Sufficiency

A sound assurance argumentation system should be capable of automatically proving that the provided arguments are sufficient to demonstrate that the system behaves as intended, assuming there are no deficiencies in supporting evidence. However, most methods used by practitioners today use surrogate criteria (such as branch coverage or range coverage), which provide inadequate evidence for assurance sufficiency and rely on human judgment. This might be unavoidable, since it might not be possible or practical to reason about behaviors exhaustively (e.g., all possible values of the software variables and all interleaving of thread executions). However, a sound assurance argument should provide a way to pinpoint where evidence deficiencies impact confidence in claims.

Unfortunately, the ever-increasing complexity of software-reliant systems—particularly safety- and mission-critical systems—has long escaped the human capacity to judge argument sufficiency. Moreover, this judgment not only must be done within each behavioral domain, but across domains. Clearly, the argumentation system must be practical, scalable, and largely automatable to support the increasing complexity of software-reliant systems.

Such an argumentation system should allow us to prove whether or not arguments are missing. For instance, consider a simplified airbag system that requires proving three logical behavior claims: (1) presence of an adult passenger, (2) occurrence of a crash, and (3) trigger inflation if and only if (1) and (2). The analysis of an assurance argument should be able to deduce insufficiency (e.g., it should be able to identify the lack of a timing behavior claim and the corresponding physics behavior claim). In this case, the missing claims are from other behavioral domains, but this can also occur within a single domain. Deducing insufficiency will require domain and cross-domain proof templates; canonical arguments for typical classes of systems that can be instantiated for specific systems and help identify missing elements of an argument.

5.5.3.6 Representing Assurance Arguments

A general and sound verification argumentation framework needs to both (1) explain to humans why we have confidence in the argument, and (2) allow the full formal and automatic integration of all the arguments that comprise the case.

As presented in our airbag example above, different technical domains use formalisms for deductive reasoning that abstracts away (or deletes) some information. For instance, logical verification (e.g., model checking) models the value transformation nature of computation. However, its models do not contain information about elapsed time or execution time. It can therefore reason about value transformation but not computation time.

Similarly, real-time scheduling models for timing verification reason about elapsed time and execution time but ignore values that the computation produces. As a result, these models cannot reason about value transformation. Finally, physical models using differential equations completely ignore that computation takes time or that algorithms represent numbers in different formats or number of bits. Hence, these models cannot reason fully about the value transformation correctness or computation time.

Some combinations of multiple disciplines (such as hybrid and embedded systems) have been developed over the years to model transformations of values in computation and the evolution of physical variables in a combined model [Alur 1993]. However, these combinations often tend to increase the complexity of the assurance and move away from practicality.

An alternative direction was proposed by Benveniste et al., who developed a framework to specify Assume/Guarantee contracts on sets of behaviors [Benveniste 2015]. In this case, the specific formalism applied to capture the behaviors is left undefined. However, the framework can define a contract algebra based on set theory that reasons about composable verification. For example, the framework can define how to combine guarantees of modules to verify a global property and how to verify that a modification to a module does not modify the assumptions it makes or the guarantees it provides.

The formalisms to define the behaviors to support the low-level description of the modules can come from different scientific disciplines, as soon as the set operations of the contract algebra are properly implemented (e.g., set intersections, inclusion, and union). Moreover, these operations can be carefully designed to minimize the interaction across formalisms and avoid a complexity explosion. Benveniste et al. conducted some initial work in this direction [Benveniste 2015]. These properties allow changing one part of an assurance argument without having to reverify any other parts of the argument that interacts with this module. This approach supports assuring software evolution through composition reasoning, which is

important for reassuring a system as it evolves. Ruchkin presented other research performed on architectural models combining different types of analyses [Ruchkin 2014].

5.5.3.7 Ecosystem Architecture

Combining evidence relies on the structure of the system, which in turn imparts structure on the system's assurance argument. The structure of the argument should influence the development–assurance pipeline. What is needed, therefore, is a proof system and supporting infrastructure that includes notions of what evidence is needed, how the evidence is gathered, how to ensure that the evidence is acceptably valid, where it's stored, when it's updated, what dependencies exist, and so forth [Richards 2019]. In particular, the dependencies between the assumptions and guarantees of components in the system and the dependencies of different types of evidence in the assurance argument must be precise.

These relationships suggest a notion of architecture that includes the relationships described above, which we refer to as the "ecosystem architecture." In this context, ecosystem architecture is defined as the aggregate structure of the software-reliant system, the assurance argument, the DevSecOps pipeline, and the development organization. It is not surprising that these structures are related and interact, since it is well-known that quality attributes impact software architectures [Bass 2012].

For a system that continuously evolves, there is a natural tension between the speed at which new capabilities are deployed and the comprehensiveness of assurance.

5.5.3.8 Reassuring Evolving Systems

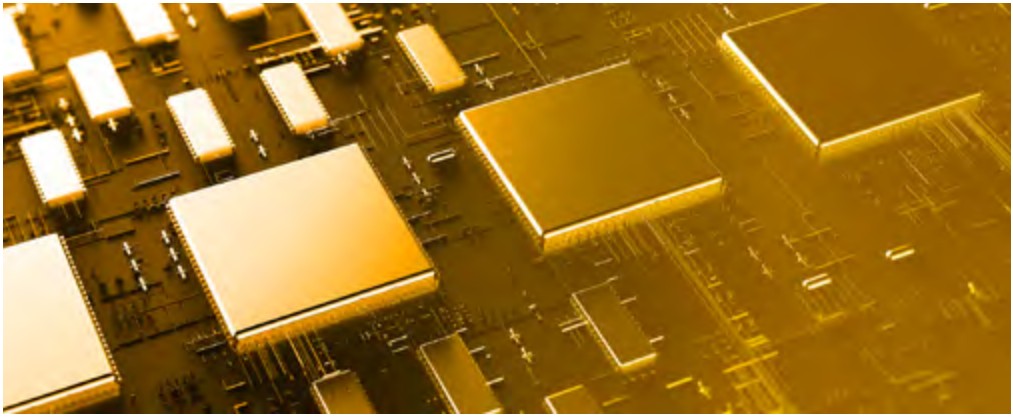
For a system that continuously evolves, there is a natural tension between the speed at which new capabilities and fixes are deployed and the comprehensiveness of assurance. In particular, it is hard to have speedy deployment with assurance due to the time it can take to decide how to modify an existing test so it satisfies some sufficiency criterion for the test suite as a whole. Likewise, it can be hard to understand how the consequence of change propagates through a system, particularly if the system is composed of modules developed and evolved independently (especially if these modules are provided by third-party suppliers only in binary form).

However, reducing re-assurance effort for a changed system is important. We know that for some safety-critical systems today, little or no changes are allowed after they pass certification. The reason for this restriction is that the only accepted way to ensure that no critical flaws have been introduced by a change is to recertify the entire system (e.g., by re-running all operational tests, which can take months). Due to such recertification challenges, some systems are not upgraded for years, which is clearly at odds with other requirements, such as upgrades to meet rapidly evolving threats or applying security patches to fix vulnerabilities.

5.5.3.9 Bounding Propagation

A theory of software evolution will provide a basis for (1) determining (and bounding) change propagation, and (2) suggesting or anticipating the need for change that is based (at least in part) on system behavior, changes in requirements, changes in operational environments, and so forth. Changes to a system as it evolves are obviously not conducted in isolation, but rather are performed in the context of an existing system (even greenfield software development can be viewed this way). For any system—and especially large-scale systems and systems-of-systems—it can be hard (if not nearly impossible) to fully comprehend the future consequences of any given change.

For example, has a change inadvertently introduced a new failure mode or vulnerability? Complex feature spaces are too hard for humans to keep in



mind. When the "distance" from a change to the place where an error manifests is large, it is nearly impossible for humans to anticipate resulting problems. Moreover, when the consequence of a change propagates across different "technical domains" (such as from security to real-time performance to material stress analysis) finding newly introduced problems can be very hard. AI- and ML-enabled components in systems exacerbate this problem, because it is not always possible to anticipate or detect changes in the environment that affect the system (such as a change in the distribution of data from that originally used to train a classifier).

5.5.3.10 Reassuring Emergent Properties

It can be hard to regain confidence in non-functional or emergent properties of a system after a change has been made, because such properties typically depend on interactions between functionally independent parts of a system. For example, unless a system is carefully architected, changes can unintentionally impair a system's security or timing properties. Likewise, changes to a system's external environment may invalidate or weaken

assumptions underlying assurance claims about functional and non-functional behaviors. Similarly, changes in usage patterns (e.g., a change in the mix of requests to a system) can stress a design, leading to behavior that is unacceptable in the changed environment. In general, current re-assurance practices are not robust when it comes to assessing the possible effects of changes on desired quality attributes.

Experience gained from observing, experimenting, and examining change patterns can be used to trigger re-assurance or bound re-assurance.

5.5.3.11 Assured Multi-Behavior Composition

As more software-reliant systems are developed using component-based technologies and platforms (many of which are provided by third parties), we need theories of composability—along with the associated methods and automated tools—that enable the specification and enforcement of composition rules that both allow (1) the creation of required behaviors in all the behavioral domains (i.e., functional and non-functional) required by the system (logical, timing, control, safety, security, etc.), and (2) the assurance of these behaviors, both during initial deployment and during subsequent evolutions. These theories, methods, and tools must be capable of reasoning about the consequences of integrating components within each behavioral domain and across domains, especially as their structure and functionality evolves dynamically over time.

5.5.3.12 Reassuring Systems Using Reusable Components

Assurance practices have not been focused on assurance efficiencies that may be possible when a system is composed largely from reusable components (such as product line systems). For example, it is not necessarily clear how argument fragments associated with such components can be reliably and efficiently composed and reused to lessen the assurance effort for the composed system.

5.5.3.13 Learning from Operational Experience

If an evolving system is already operational, then data may exist about its reliability, safety, and so forth. AI and ML could make use of this data in at least two ways. First, it could be used to detect deviations from normal; such techniques have been used to analyze network traffic for cybersecurity (using macro-level statistical indicators), but could also be used to monitor other indicators, such as order queue lengths for e-commerce systems (e.g., Amazon.com). Second, in the spirit of Netflix's chaos monkey, hypothetical changes could be imposed on the system, and AI or ML tools could determine patterns of propagation for different types of change. This analysis could be used to characterize potentially problematic classes of change.

Generally, we posit that the experience gained from observing the system in operation, experimenting with it, and examining patterns of change that invalidate sufficiency arguments can be used to trigger re-assurance or bound re-assurance. Operational experience also evinces confidence in the sufficiency of a system's assurance argument.

5.5.4 Research Questions

- What are suitable representations of an assurance argument?

  A formal and automatable representation of an assurance argument must be developed that can handle the numerous domains of assurance (e.g., logical correctness, timing correctness, and security correctness) and their interactions. An overall argumentation mechanism must be devised that uses proof systems from all of the relevant domains to prove the properties of interest and demonstrate the extent to which the available evidence supports the arguments.

  More specifically, this overall mechanism should show that (1) the collection of arguments is sufficient to prove the properties for all of the relevant domains, and (2) that all possible effects that one domain can have on the other domains have been considered. An overall argumentation mechanism must also support the discovery of missing assumptions and inadvertently omitted domains. Argumentation templates or patterns could help with this.

  To make this concrete, an assurance argument itself can be reasoned about formally (e.g., is it sound?; are its assumptions consistent?; and does it appropriately account for all aspects of the system?). Providing a representation that can address these issues is a foundational challenge for assuring an evolving system.

- What are patterns of evolvable assurance arguments?

  Given a generalized assurance argument about a system and a subsequent change to the system, determining the extent to which the assurance argument needs to change to account for the system change, while otherwise remaining sound, is a challenge. For continuously evolving systems, bounding the cost of re-assuring will be especially important. Just as for software architecture, bounding the effect of change depends on the structure of the argument. Structuring arguments will require developing notions of interface and encapsulation for assurance arguments.

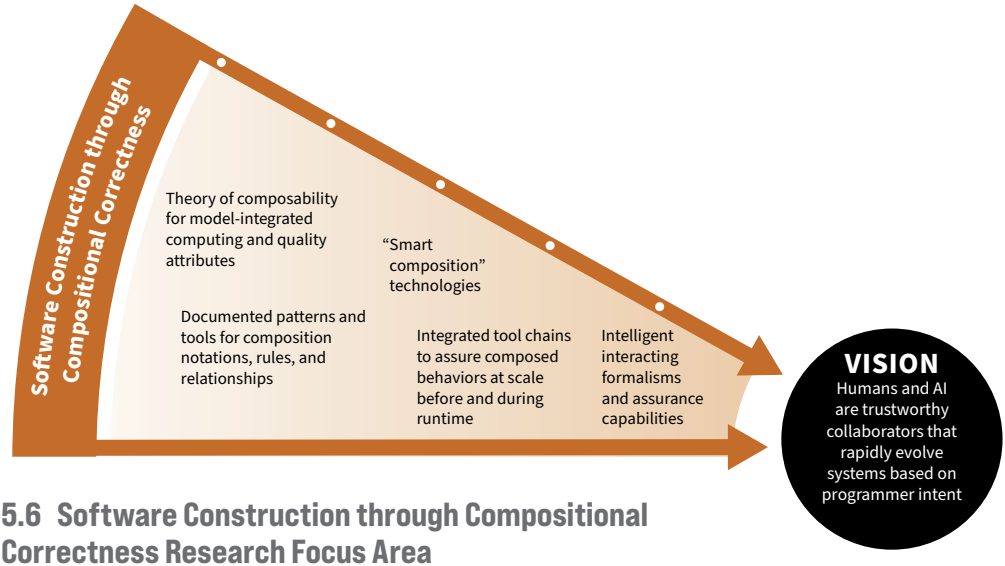
- Are there opportunities to use ML to automatically detect needed changes in assurance arguments?

  Automatically learning from operational experience (i.e., actual use) can be used to predict the need for changing the system by discovering patterns of operational behavior that indicate that assurance assumptions have been violated by new, possibly unanticipated operating conditions or by discovering patterns of argument that might be flawed.

5.5.5 Research Topics

These research topics are important to assuring continuously evolving software systems and will contribute to developing a theory of assured software evolution. The following milestones are indicators of progress toward developing such a theory:

- *Theory for assured composition of evidence.* Development of an overall approach or theory for assured composition of proof systems allowing combining disparate evidence from different domains to assure key software-reliant system properties.
- *Assurance argument templates.* Development of an approach for ensuring argument sufficiency by using assurance argument templates to identify missing assumptions, domains, or interactions between domains and appropriately representing assurance arguments.
- *Modifiable assurance arguments.* Development of an approach for creating modifiable assurance arguments through the notion of an ecosystem architecture and gaining a better understanding of reassuring systems using reusable components.
- *Automatic detection if a system change invalidates an assurance argument.* Development of techniques for automatically determining whether a system change introduces a new failure mode, a new vulnerability, or generally invalidates an existing assurance argument by determining how to bound propagation of change and re-assure emergent properties.
- *Automatic system update recommendations based on operational data.* Development of techniques for automatically making recommendations for system updates based on using data gathered from the operational system to learn from operational experience.
- *Tool chains for combining evidence to re-assure a system.* Development of integrated tool chains including AI and/or ML techniques to support combining multiple types of evidence with operational data to re-assure a system.



5.6 Software Construction through Compositional Correctness Research Focus Area

5.6.1 Goals

To address challenges associated with scale, complexity, and time-to-market, software-reliant systems are increasingly developed using component-based technologies. To ensure component-based systems meet their business, technical, and financial requirements and constraints, research is needed on theories of composability—along with the associated methods, platforms, and automated tools—to enable the specification and enforcement of composition rules that allow (1) the creation of desired behaviors (both functionality and quality attributes), and (2) the assurance of these behaviors during initial deployment and throughout the lifecycle. Key goals of this research area are, therefore, to develop and/or refine theories, methods, platforms, and tools that enhance our ability to construct software by composing components correctly and reasoning about the consequences of various compositions, especially as the structure, functionality, and provenance of component-based systems evolve over time.

5.6.2 Limitations of Current Practice

The scope and scale of software-reliant systems continues to grow and change continuously, such that developing and sustaining software from scratch is no longer realistic for most production systems, including mission- and safety-critical cyber-physical systems where the right answer delivered too late becomes the wrong answer. Moreover, the size and complexity of these systems makes it unrealistic for any one person or group to understand the entire system. It has therefore become common—and often necessary—to integrate (and continually reintegrate) software-reliant systems using modular components. Many of these components, however, are reused from existing elements that may not have been designed initially for composition, integration, or evolution. This situation is particularly problematic in heterogeneous computing environments, where components are written in a mélange of programming languages atop platform technologies with questionable quality and provenance.

Since the 1950s, software researchers and developers have been creating abstractions that help them program in terms of their design intent rather than the vagaries of the underlying computing environment—such as CPU, memory, and network devices—and shield them from the complexities of these environments. From the early days of computing, these abstractions included both language and platform technologies. For example, early programming languages, such as assembly and Fortran, shielded developers from the complexities of programming directly with machine code. Likewise, early operating system platforms, such as OS/360 and Unix, shielded developers from the complexities of programming directly to hardware.

Although these early languages and platforms raised the level of abstraction, they still had a distinct "computing-oriented" focus. In particular, they focused on abstractions of the solution space (i.e., the domain of computing technologies themselves) rather than abstractions of the problem space (i.e., application domains, such as telecom, aerospace, healthcare, insurance, and biology). Moreover, too much effort was needed to create or re-create applications from scratch atop these low levels of abstraction rather than expressing designs in terms of application domain concepts and then composing applications using reusable components that could be integrated into application frameworks and software product lines.

Advances in languages and platforms during the past several decades have raised the level of software abstractions available to developers. For example, developers today typically use more expressive programming languages, such as Python, Java, Kotlin, or C++, rather than Fortran or C. Likewise, today's reusable class libraries (such as the C++ Standard Template Library or Java Collections) and application frameworks provided by popular software platforms (such as Android, iOS, and Spring middleware) minimize the need to reinvent common and domain-specific

middleware services, such as discovery, event notification, transactions, security, and resource management. Due to the maturation of these third-generation languages and reusable platforms, software developers are now better equipped to shield themselves from complexities associated with creating applications from scratch using earlier technologies and can instead focus on composing them using reusable components and frameworks provided by common platforms.

Despite these advances, however, several vexing problems remain. At the heart of these problems is the growth of platform complexity, which has evolved faster than the ability of general-purpose programming languages to mask it. For example, popular middleware platforms, such as Node.js, Sparks, the Data Distribution Service (DDS), and Android, contain thousands of classes and methods with many intricate dependencies and subtle side effects that require considerable effort to compose, adapt, and tune properly. Moreover, these platforms often evolve rapidly (and new platforms appear regularly), so developers expend considerable effort manually porting their application software to different platforms or newer versions of the same platform.

A related problem is that most application and platform software is still written and maintained manually using third-generation languages, which incurs excessive time and effort, particularly for key integration-related activities, such as system deployment, configuration, and quality assurance. For example, it is hard to write Python or C++ code that correctly and optimally deploys large-scale distributed systems with hundreds or thousands of interconnected software components, which is becoming more common in societal-scale software systems (see Section 5.7).

Even using newer notations, such as Extensible Markup Language (XML)-based deployment descriptors popular with middleware platforms like Android, is fraught with complexity. Much of this complexity stems from the semantic gap between levels of abstraction. For example

- the design intent (e.g., deploy components 1-50 onto nodes A-G and components 51-100 onto nodes H-N in accordance with system resource requirements and availability)
- the expression of this intent in thousands of lines of handcrafted XML whose visually dense syntax conveys neither domain semantics nor design intent

Due to these types of problems, the software industry is reaching a complexity ceiling where modern platform technologies, such as reactive microservices in cloud deployments [Gillberg 2020], have become so complex that developers spend years mastering and wrestling with platform APIs and usage patterns, and yet are often familiar with only a subset of the platforms they use regularly. Moreover, third-generation

The software industry is reaching a complexity ceiling where modern platform technologies have become so complex that developers spend years mastering and wrestling with platform APIs and usage patterns.

languages require developers to pay close attention to numerous imperative programming details, such as terminating loops correctly; detecting and handling errors and exceptional conditions properly; and avoiding buffer overflows, null pointers, and double-deletions of dynamically allocated memory [Seacord 2005].

These types of tactical issues make it hard for developers to focus on strategic architectural issues, such as system-wide correctness and the performance of applications composed from reusable components. Although modern interactive development environments (IDEs) help address some of these tactical issues (such as managing dynamic memory management properly), IDEs are limited in their ability to detect and help rectify common programming mistakes. Likewise, IDEs provide limited support for guiding the correct application of architectural patterns, which still require considerable manual design and implementation effort.

Today's fragmented tools and methods also make it hard for software developers to know which components and subsystems of their applications are susceptible to side effects arising from changes to user requirements and language and platform environments. This lack of an integrated view—coupled with the danger of unforeseen side effects resulting from composing software components originating from a wide range of sources of questionable quality and provenance—often forces developers to implement suboptimal solutions that unnecessarily duplicate code, violate key architectural principles, and complicate system evolution and quality assurance. Suboptimal solutions are particularly problematic when developing and assuring mission- and safety-critical cyber-physical systems that evolve continuously (see Section 5.5).

5.6.3 Topics for Research

5.6.3.1 Compositional Correctness via Model-Driven Engineering (MDE) Tools

One promising approach for addressing platform complexity, as well as the inability of third-generation languages to alleviate compositional complexity and express domain concepts reliably and securely, is to develop and apply model-driven engineering (MDE) technologies [Schmidt 2006], such as MATLAB, Simulink, Rhapsody, and the Architecture Analysis and Design Language (AADL) [SEI 2019]. Rather than programming with third-generation languages such as Java, Java Script, and C++, these model-driven approaches enable software developers to program at a higher level by combining the following:

- Domain-specific modeling languages (DSML), whose type systems formalize the application structure, behavior, and requirements within particular domains, such as software-defined radios, avionics mission computing, online financial services, warehouse management, or even the domain of middleware platforms [Voelter 2013]. DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.
- Transformation engines and generators that analyze certain aspects of models and then synthesize various types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations. The ability to synthesize artifacts from models helps ensure the consistency between application implementations and analysis information associated with functional and quality of service (QoS) requirements captured by models. This automated transformation process is often referred to as "correct-by-construction" [Ge 2018], as opposed to conventional handcrafted "constructed-by-correction" software development processes that are tedious, error prone, and hard to assure.

Existing and emerging MDE technologies apply lessons learned from earlier efforts at developing and composing software via higher-level platform and language abstractions. Instead of general-purpose notations that rarely express application domain concepts and design intent, DSMLs can be tailored via metamodeling to precisely match the domain's semantics and syntax. DSMLs express behaviors or computations in a manner that resembles a specific application domain in which end users operate.

For example, financial systems could express behaviors or computations in terms of accounts and ledgers. As another example, avionics systems could express behaviors and computations in terms of speed, velocity, and altitude. Essentially, MDE technologies enable programming software at a higher level of abstraction using DSMLs that expand the syntax and semantics of third-generation languages to develop applications on top of popular platforms and frameworks. MDE methods and tools can thus simplify software construction through compositional correctness by enabling software developers to focus largely on domain-centric business logic expressed via models created using DSMLs. The bulk of their applications can then be synthesized and/or integrated from these higher-level models, which can be mapped reliably, securely, and efficiently onto the underlying software frameworks and platforms.

DSMLs often express elements in a domain graphically, rather than purely textually (as is the case with third-generation languages). Having graphic elements that relate directly to a familiar domain not only helps flatten learning curves but also helps a broader range of subject matter experts, such as system engineers and experienced software architects, ensure that software systems meet user needs. Moreover, MDE tools impose domain-specific constraints and perform model checking that can detect and prevent many errors earlier in software and system lifecycles.

In addition, because today's platforms have much richer functionality and QoS than those in the 1980s and 1990s, MDE tool generators need not be as complicated because they can synthesize artifacts that map onto—and compose into—higher-level, often standardized, middleware platform APIs and frameworks rather than lower-level OS APIs. In particular, MDE methods and tools can take expressions in DSMLs and auto-generate large amounts of the code that is then connected with components created via modern middleware platforms, such as Spring or React.js. Consequently, it is often easier to develop, debug, and evolve MDE tools and applications created using these tools and associated model-driven middle frameworks [Costa 2017].

Integrating MDE tools and middleware platforms to deploy application services end to end can help developers configure the right set of services into the right part of an application in the right way. MDE analysis tools can help determine the appropriate partitioning of functionality that should be deployed into various component servers throughout a network. For example, tools like MATLAB, Simulink, TimeWiz, and RapidRMA allow application developers to model and visualize their application end to end (and their QoS requirements). In particular, Simulink allows application developers to model, analyze, simulate, verify, and rapidly prototype applications for mission- and safety-critical cyber-physical systems.

5.6.3.2 Compositional Correctness via Dependency-Injection Frameworks

The past decade has witnessed the evolution of another approach to address the problem of tedious and error-prone code that often results from the sole reliance on third-generation languages to create systems. Instead of representing software behaviors in the form of MDE constructs (such as custom-built domain-specific modeling languages that may be unfamiliar to many programmers), software developers can write the bulk of their business logic in their programming language of choice and then declaratively annotate this code with various tags that provide information used by annotation processing tools. In turn, these tools can then auto-generate and compose software via dependency-injection, which is an advanced form of separation of concerns used to construct and compose components by "auto-wiring them together," thereby increasing readability, code reuse, and automation.

Software developers can write the bulk of their business logic in their programming language of choice and then declaratively annotate this code with various tags that provide information used by annotation processing tools.

Dependency-injection frameworks, such as Spring and Dagger, allow programmers to declaratively specify certain properties and attributes via annotations [Patel 2018]. These annotations are then processed automatically by tools that generate and connect large amounts of "glue code," thereby minimizing software development for aspects like persistence, remote communication, and security, as well as data access and manipulation. These annotation processing tools inspect, generate, and compose the various aspects needed to synthesize a working application.

Annotation-based dependency-injection frameworks simplify software construction through compositional correctness [Bojkic 2020]. In particular, software developers focus largely on business logic written in familiar programming languages and then place annotations into certain classes. Annotation processing tools then inspect and generate much of the "glue code" needed to create and compose a working application.

For example, to assign a particular class as one that provides data, an @Data annotation can be used to instruct an annotation processing tool to generate code that has certain additional capabilities, such as setter/getter methods to update/read fields in a particular object. Annotations like @Value can also indicate that an entity (such as an object) will be stored persistently in a database. Certain fields can also be annotated (e.g., via an @Id annotation) to indicate that they are intended for use as primary keys in a database.

These declarative annotations enable software developers to express metadata about the fields and elements in a class so that programmers need not write all this code themselves. The annotation tool infrastructure instead generates glue code and other important tasks, such as determining dependencies between the different defined components. These tools also automatically compose dependencies together without having to undergo explicit composition. For example, a software developer can determine

that a particular class depends on another class and then annotate it with another annotation, such as @Autowired. This annotation indicates to the tool infrastructure that it needs to retrieve an implementation of that entity and arrange to connect the pieces together.

Annotation processing tools work together with the dependency-injection framework to instantiate other dependent components and connect them all together, so software developers need not manually establish all the connections and integration and composition. Instead, software developers declare those dependencies and the framework identifies the implementations and automatically composes them. This loose coupling enables software developers to write their components in a more modular manner and then leverage the tool infrastructure to perform correct compositions on their behalf.

Beyond their worth as a more efficient, less-error prone means of constructing new software systems, annotation-based dependency-injection frameworks have become popular because they are widely used to program the World Wide Web. In particular, frameworks like Spring encapsulate popular low-level protocols and notations, such as the Hyper-Text Transfer Protocol (HTTP), JavaScript Object Notation (JSON), and Extensible Markup Language (XML), that encode and exchange data types back and forth between clients and servers. These low-level communication and data-transfer mechanisms can, in turn, be encapsulated via annotations (such as @GetMapping and @PostMapping) that automatically convert HTTP GET and POST requests sent over Transmission Control Protocol (TCP) connections into conventional method calls on objects whose business logic is written using popular third-generation languages.

Dependency-injection frameworks such as Spring enable software developers to annotate (e.g., via the @RestController tag) a conventional class to designate that incoming HTTP GET and POST requests should be routed to its endpoint methods and processed using conventional method calls. In particular, annotations such as @PostMapping or @GetMapping can direct the incoming requests at the HTTP level, route them to the appropriate endpoint method, and then dispatch this method to process the contents contained in the HTTP message. This annotation-based approach enables software developers to focus on business logic instead of the intricacies involved in sending messages from a client to a server, all of which is accomplished via declarative annotations and auto generation, rather than developers manually writing this code imperatively.

By enabling software developers to define their components and applications without concern for the communication mechanisms used to operate between them, it is no longer necessary to create inflexible monolithic applications where all components reside in a single address space. Instead, microservice architectures have emerged that construct software by composing many smaller elements (i.e., microservices) whose configurations can be customized and deployed onto the underlying computing infrastructure, such as Amazon Web Services (AWS), an on-premises private cloud, or an IoT sensor network [Lira 2019]. This loosely-coupled architecture enables system scalability by leveraging multi-core processes and distributed core clusters, because a system is composed from small components whose location can change relatively flexibly and transparently.

Dependency injection frameworks also employ software build managers, such as Gradle and Maven, that allow application developers to specify the libraries they depend on in a highly flexible, programmable, and late-binding manner. These capabilities, in turn, enable the dynamic assembly and evolution of component implementations from many libraries and packages available via the World Wide Web. In particular, these components need not reside on the local build computer, but instead can be downloaded automatically from remote repositories and installed during the development process.

This late-binding approach to software dependency resolution allows developers to compose software from components, some of which they wrote (and annotated), but most of which were written by other developers. The dependency-injection framework and build tools track all these dependencies, compose everything, and connect all the components without requiring extensive manual effort on the part of the software developers.

5.6.4 Research Questions

There are numerous research questions for achieving software construction through compositional correctness based on the methods, tools, and platforms described in the previous section, including:

- How can we assure systems developed via loosely-coupled components?

  While software developers can derive benefits from MDE tools and dependency-injection frameworks to generate and compose large portions of their software systems, challenges persist because models and annotations are often hard to debug at the "source" level and, thus, are hard to statically assure. One reason for this difficulty stems from the implicit dependencies existing in such loosely coupled systems. In particular, when examining the source code itself, it is hard to determine, statically, what implementations will be provided after various components are synthesized and connected.

  This challenge motivates the need for advanced research on annotation browsers and static dependency analyzers that understand the semantics of the various layers and tool chains that comprise loose-coupled component-based systems. Likewise, there is also need for research on "smart composition" methods, tools, and platforms that can intelligently and correctly compose components not initially designed to work together by automatically generating adapters that provide efficient and type-safe semantic integration.

- How can we ensure a system will function properly and securely well before runtime?

  Software engineers and operators often do not know if a loosely-coupled component-based system will function correctly until it starts to run due to the lack of advanced visualization or testing tools, such as quality assurance pipelines based on DevSecOps methods and tools that understand the semantics of the annotations and the late-binding of dependency-injection frameworks. While unit tests or integration tests represent a time-honored approach to assuring conventional software system, these methods are often untenable for certain types of safety-critical or mission-critical systems (i.e., avionics, medical devices, or nuclear reactors) where system operators must have confidence that they will function properly and securely well before runtime.

  With annotation-based dependency-injection frameworks, software developers need greater assurance that a system is going to work correctly earlier in the software lifecycle. This need presents an opportunity for researchers to develop better tools that can conduct deeper analysis and offer greater confidence that the system will work as intended prior to its runtime execution.

- How can we debug software that is developed with model-driven or dependency-injection approaches?

  If model-driven or dependency-injection approaches are used, there is a general lack of debuggers that allow developers to step through their software at the level it is written in (as opposed to the level at which it has been generated). This problem should be familiar to prior generations of software developers. In the early days of compilers and other language processing tools, programs could be written at a higher-level language, such as Fortran or C, but source-level debuggers that allow program debugging at the higher-level language level were not initially available. Rather, software developers had to debug at the assembly code level, which was tedious and error-prone.

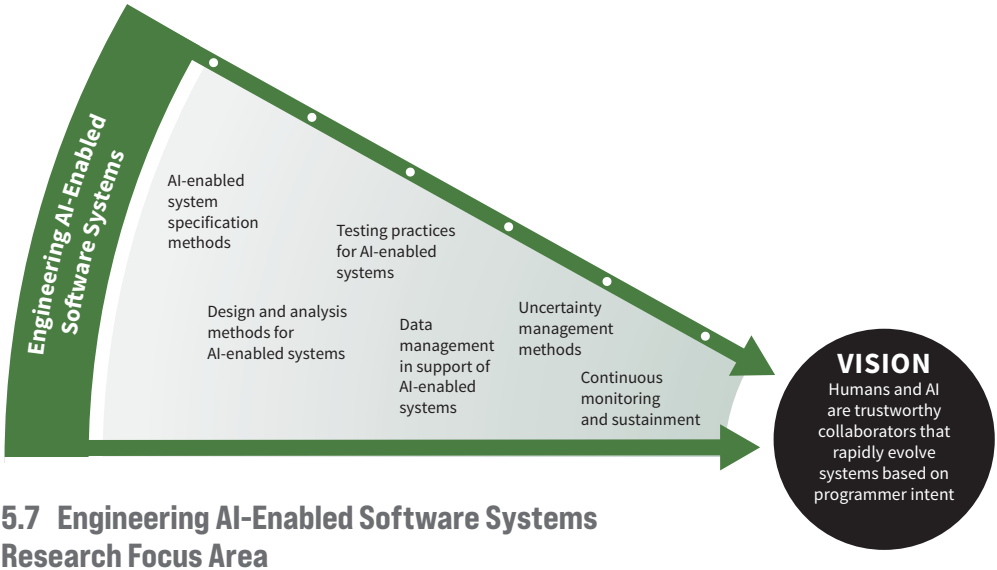
  These types of limitations made it hard to debug higher-level code, because developers had to keep falling back on lower-level code generated by the compiler instead of the code they wrote and that the compiler used as input. With model-driven engineering and annotation-based dependency-injection frameworks, the level of abstraction has been raised by many more layers. This ever-growing stack of abstraction levels motivates research on developing tools that can debug code at the model level or annotation level instead of the level of the code generated by these automated processing tools.

5.6.5 Research Topics

As the preceding sections demonstrate, the software community needs to focus on evolving formalisms and tools to address the issues described. We invite the community of researchers and practitioners to raise the level at which mission- and safety-critical systems developed using these approaches can be assured.

Research topics will be incremental and are summarized below:

- *Theory of composability for model-integrated computing and quality attributes.* Develop a new composability theory that uses modeling and quality attribute concepts to integrate components developed in accordance with component-based technologies (including, but not limited to, model-driven engineering and/or annotation-based dependency-injection frameworks) with assurance "baked in" to the completed composition.
- *Documented patterns and tools for composition notations, rules, and relationships.* Develop tools, notations, and rules for composition that enable assurance of the composed system in terms of quality attributes, and reduce the need for purely runtime testing (i.e., be capable of detecting defects during earlier phases of the software lifecycle).
- *"Smart composition" technologies.* Create methods, tools, and platforms that can intelligently and correctly compose components that were not initially designed to work together by automatically generating adapters that provide efficient and type-safe semantic integration.
- *Integrated tool chains to assure composed behaviors at scale before and during runtime.* Develop tools that inform quality attribute engineering trade space decisions to enable composition of systems at scale and assure composed behaviors.
- *Intelligent interacting formalisms and assurance capabilities.* Build an approach that empowers developers to use multiple formalisms (e.g., combining model-driven and annotation-based approaches) to compose systems and measure the impact on assurance.



5.7 Engineering AI-Enabled Software Systems Research Focus Area

5.7.1 Goals

The systems of the future—from smart cities and buildings, to defense and transportation systems, to healthcare—will likely incorporate AI elements. For national defense applications in particular, AI-enabled software systems promise the ability to improve the speed of response to changing missions and promote information dominance by developing adaptive systems.

Advances in ML algorithms and the increasing availability of computational power are already resulting in huge investments in systems that aspire to exploit AI. AI-enabled systems, software-reliant systems that include data and components that implement AI algorithms mimicking learning and problem solving, have inherently different characteristics than software systems that do not use AI components. These differences are driving academia, industry, and governments to explore the creation of a new discipline of engineering called AI Engineering [Horneman 2019; Bosch 2020; Santhanam 2019]. Developing and adopting transformative AI solutions that are safe, ethical, and secure will require cultivating the field of AI engineering. CMU has defined an abstract technology model called the AI Stack for driving the clear understanding in the development and deployment of AI-enabled systems [Moore 2018]. The SEI has further identified scalable AI (focusing on how to scale algorithms, data, and infrastructure), robust and secure AI (focusing on understanding the challenges around securing AI systems against new adversarial threats), and human-machine teaming (focusing on the challenges around interactions driven by decision making with AI elements) as three important pillars to drive the definition of AI engineering practices [SEI 2021].

However, AI-enabled systems are, above all, software systems. The development and sustainment of these systems have many parallels with building, deploying, and sustaining software systems. Research programs in software engineering will need to focus on the challenges that AI elements bring to software analysis, design, construction, deployment, maintenance, and evolution. The goal of this research area is therefore to explore what existing software engineering practices can reliably support the development of AI systems and what new software engineering research challenges need to be solved in order to reliably construct AI-enabled software systems.

5.7.2 Limitations of Current Practice

AI-enabled systems in the next decade will likely continue to be dominated by advances in ML algorithms, related in particular to advances in deep learning due to a spike in research and the rapid advances that are emerging. Incorporating AI and ML components into software systems exacerbates many of the existing challenges involved with engineering software, with or without AI elements: for example, how to have confidence in systems, how to predict and control emergent behaviors, how to formally specify requirements when there is uncertainty in data and the patterns to be extracted from the data, and how to contain and manage change. Attempts to introduce ML component development into systems development highlight bottlenecks and mismatches between model development, software development, infrastructure development, and operational processes and artifacts [Amershi 2019; Lewis 2021].

Applying current software engineering practice to the development of AI-enabled systems faces the following limitations:

- Software development processes, including Agile processes, can pose challenges when aligning traditional software development activities with the experimental, iterative, and incremental nature inherent in the development of ML models and other AI components [Amershi 2019; Rahman 2019]. ML model development relies on generation and test approaches that make it hard to align with sprint boundaries and the identification of done criteria common in most software development processes.
- Systems developed to train ML models may be expensive. One approach to address this challenge focuses on developing self-supervised systems that do not require the training of models in advance and shift the effort of labeling [Hendrycks 2019]. In these systems, engineers focus on monitoring the model and reacting to changes. This fundamental shift to self-supervised systems will continue. However, new techniques will need to be developed to address what elements of the system can be self-supervised, how self-supervised elements can work together with self-adaptation elements (in particular with self-adaptive software systems), and the resulting challenges for system monitoring and observability (among other things).

- Platforms that support system integration and model development assume existing software platforms will scale out-of-the-box to support the integration of AI components with the rest of the system. Techniques to ensure successful deployment and sustainment of AI-enabled systems after deployment are lacking. Although approaches for building on existing software engineering techniques, such as MLOps (which applies DevSecOps principles to ML component development), have emerged, they still rely on existing tools without appropriate metrics and analyses to provide timely and relevant information to developers.
- Systems that contain AI components cannot be reliably tested, verified, and certified. Maintaining safety and security as new computational paradigms such as AI are introduced cannot be guaranteed. While the emergent field of AI engineering focuses on techniques such as algorithmic trust, there is a dire need for software testing and analysis techniques to support testing of AI components and AI-enabled systems.

There is a dire need for software testing and analysis techniques to support testing of AI components and AI-enabled systems.

5.7.3 Topics for Research

To overcome these limitations, research will need to focus on augmenting software engineering techniques in the specification of systems with AI components and their design, architecting, analysis, deployment, and sustainment. In particular, progress needs to be made in the areas described below.

5.7.3.1 AI-Enabled, Systems-Specific Quality Attributes and Architecture Concerns

Quality attributes, or properties used to evaluate the quality and fitness of a system for its mission goals, drive the selection of architecture approaches and, consequently, the structure and behavior of software systems [Bass 2012]. Business and mission goals drive the domain and relevant types of systems, and they are all critical in identifying high-priority quality attributes. Identifying the driving quality attributes for AI-enabled systems, specifying them, and understanding how they can be analyzed will require further research. While particular mission and business goals will shape the expected quality attributes, others will likely emerge as top priorities as well. These attributes include explainability, data centricity, verifiability, monitorability, observability, and fault tolerance, at a minimum [Pons 2019]. Analysis techniques to assure their correct design and implementation will need to be developed. These attributes will also drive the development of techniques for addressing fairness, unintended bias, and ethics. Ethics is a complex subject which requires progress in policy, social sciences, and technical realms. However, some aspects of ethics will fall under the umbrella of our ability to design them into AI systems [Ozkaya 2019].

5.7.3.2 Methods for Specifying Uncertainty and AI-Enabled System Behavior

There are systems whose requirements we can and do know up front, or that we can discover easily through iteration. Those tend to be manageably sized systems that we have learned how to develop over the years. In reality, many software systems, even without AI components, need to model uncertainties and "unknown unknowns" throughout their development. Uncertainty is the dominant characteristic of AI-enabled systems. In particular, learning from data and the discovery process introduced with ML-modeling activities introduces many uncertainties. Existing software requirements engineering and traceability techniques will need to be expanded to decouple AI problem and model specification (which drive ML component development) from the system specification (which drives the test and evaluation of the resulting system—including the AI, data, and other software elements).

5.7.3.3 Techniques to Analyze and Manage Change

The hard-to-trace dependencies, in particular those induced by data dependencies, become a significant source of failure in ML systems. These hidden and unstable data dependencies make applying known architectural patterns to manage system evolution and separate concerns challenging (for example, when inputs come from another ML model that updates over time). Introduced by Google engineers as the Changing Anything Changes Everything (CACE) principle, systems with ML components not only become highly coupled but also more complex [Sculley 2015]. However, the reality is that hidden dependencies have always been a challenge to manage, in particular runtime dependencies, because software engineers lack tools to analyze, model, and visualize these dependencies. Data dependencies that are inherent in AI-enabled systems suggest that we need the tools we already lacked even sooner. The CACE principle implies that there is a dire need to better manage change propagation, both to reduce the uncertainty of the expected results and to improve the engineer's ability to debug systems. Advancing software engineering tools and techniques to analyze and safeguard systems for change propagation will be an essential priority.

5.7.3.4 Reliability in AI-Enabled Systems

Challenges related to analyzing and designing for reliability in AI-enabled systems are similar to those of embedded real-time systems. Both are often developed by integrating many disparate software and hardware components, some of which may be developed and owned by different parties. AI components will also increasingly be developed independently; hence we will need techniques to analyze their attributes to predict their behavior and integrate them into the rest of the system reliably. The reality of the future of software systems will be an increased number of such disparate components and the heterogeneity they introduce to system design and operation. These disparate and heterogenous systems will require software and AI engineers to assume normal failure and develop safeguard techniques to ensure reliable system design, development, and operations.

5.7.3.5 Monitoring and Self-Adaptation

The increased awareness of the role of data in the success of AI-enabled systems will drive rapid progress in overcoming challenges stemming from a lack of data, data with noise, and techniques to label data. It will also drive the development of robust data engineering pipelines. Consequently, challenges will shift to efficient deployment and sustainment of AI-enabled systems. Current monitoring approaches mostly rely on collecting common system metrics, such as the throughput and resource consumption of systems. Monitoring techniques to provide information on drift detection and the optimal time to retrain need further research. Different aspects of monitoring and self-adaptation need to be taken into account, including

the monitoring of data and changes in data, monitoring of the model (and whether it continues to behave as intended), and monitoring of the system with AI components. The existing software engineering body of work on self-adaptation and self-healing systems will need to consider challenges introduced by AI components, in particular the dependence on data and how these data dependencies affect the rest of the system and its adaptive responses. Accounting for the implications of unintended consequences and incorrect model behavior will require the development of new monitoring and adaptation techniques.

5.7.3.6 Testing, Deployment, and Sustainment of AI-Enabled Systems

The development and deployment of AI-enabled systems, in particular ML-enabled systems, involves three distinct perspectives (along with their own workflows and roles): data science, software engineering, and operations. These three distinct perspectives, when misaligned due to incorrect assumptions, can cause mismatches that result in failed systems. Improved automation and formalism in specifying and detecting these mismatches, incorporating these tools into deployment workflows and MLOps pipelines, and developing testing techniques that extend existing software testing approaches to effectively test AI components will be needed. Currently, testing of AI components relies on ad hoc or manual testing practices. Testing techniques for ML components, similar to those that exist for traditional software components and systems, is a gap that needs to be addressed.

5.7.4 Research Questions

The need to support AI-enabled systems through software engineering research has reached a point similar to the period in which we realized security, usability, and privacy had to be treated as primary quality concerns in software systems: If we do not design for system users and architect for usability, systems fail. Today, security, usability, and privacy are among many other mainstream system concerns, and we have common vocabulary and analysis methods to design and check for such attributes. Similar progress needs to be made in identifying and understanding AI-enabled system-specific qualities. Existing design, test, evaluation, and data management techniques will help us understand how to design, deploy, and sustain the structure and behavior of AI-enabled systems, and they will also provide a stepping stone for addressing the following key research questions in the next 5 to 10 years:

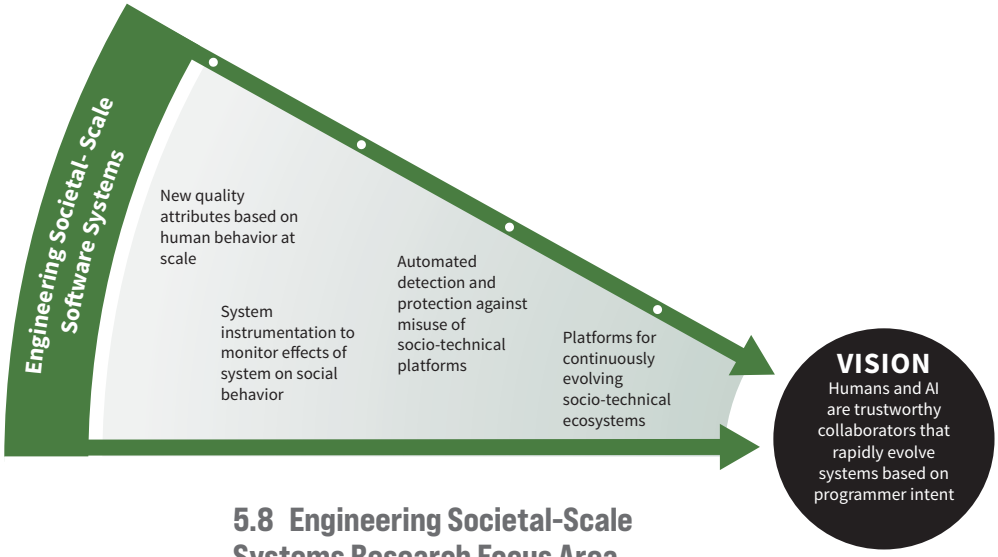
- What are key quality attributes and architecture patterns to support explainable and trusted AI-enabled systems? What design tactics and analysis techniques support enforcing these attributes?
- How can uncertainty be modeled to help specify and monitor AI-enabled system behavior?

- What metrics enable fine-grained monitoring of AI and non-AI components to derive timely sustainment decisions, such as retraining, decommissioning, new data collection, and mission-driven decisions?
- How does an AI-enabled system self-heal and correct errors once it is deployed?
- How can operational analysis be incorporated seamlessly into AI-enabled system development and deployment, and how can it be supported by tools in MLOps frameworks?
- Which existing software testing techniques can support testing AI components? What do unit, integration, and regression testing for systems with AI components look like?

5.7.5 Research Topics

The ability to make progress in any of these areas will have dependencies on the other areas, and work on all of them needs to start immediately. Progress will be iterative and incremental, with the following milestones guiding success:

- *AI-enabled system specification methods.* Methods for specifying AI-enabled system behavior need to be developed.
- *Testing practices for AI-enabled systems.* Unit, integration, and regression testing practices for AI-enabled systems need to be well understood.
- *Design and analysis methods for AI-enabled system.* Key AI-enabled system quality attribute concerns, including explainability, monitorability, reliability, and trust, will need to be supported by architectural patterns, tactics, and analysis methods.
- *Data management in support of AI-enabled systems.* Understanding the impact of data on system behavior, data architecting, and change management needs to be well supported by analysis and conformance tools.
- *Uncertainty management methods.* There need to be techniques to model, analyze, and design for uncertainty.
- *Continuous monitoring and sustainment.* AI-systems need to be effectively monitored, self-healed, evolved, and sustained.



5.8 Engineering Societal-Scale Systems Research Focus Area

Many societal-scale software systems, such as today's commercial social media systems, are designed to influence people and keep them engaged. Companies that develop and deploy such systems generally derive revenue from selling targeted advertisements to promote products and services, or they exercise influence in other ways, such as advocating particular political views. Avoiding bias and ensuring the accuracy of information are not always goals or outcomes of these systems.

Software engineering for societal-scale systems focuses on predicting the full range of impacts, including unintended consequences and the potential for misuse and manipulation that arise when humans are integral components of a system.

5.8.1 Goals

Software engineering for societal-scale systems focuses on predicting the full range of impacts, including unintended consequences and the potential for misuse and manipulation (which we refer to as socially inspired quality attributes) that arise when humans are integral components of the system. The goal is to leverage insights from the social sciences to build and evolve societal-scale software systems that fulfill their intended purpose and pose minimal risk of undesired or unintended consequences. Research will enable better prediction of system behavior for building and evolving societal-scale socio-technical software systems, constructing and evolving systems with humans as components, and continuously mitigating risks of unintended bias, misplaced trust, violations of privacy expectations, concealed influence, or unrestrained social manipulation [Feiler 2006].

Societal-scale systems consist of more than conventional social media systems. The essential characteristic of these systems is that they are information and communications channels that foster desired outcomes (such as engagement and action) as a primary source of revenue. Some examples include the following:

- social media platforms, such as Facebook, Twitter, and Instagram
- search platforms, such as Google Search and YouTube, that help people find desired content on the Internet or hosted by the service and that also provide individualized recommendations based on data
- systems that detect a software developer's knowledge and adapts their experience to mentor or train the developer
- systems that attempt to predict events (such as school shootings, flu outbreaks, and super spreaders) based on search or other data
- gamification to increase engagement in areas such as personal health or financial activities

5.8.2 Limitations of Current Practice

There exists a very limited understanding of how these systems influence the behaviors of individuals over time, or how the aggregate behaviors of millions of people can be predicted.

Societal-scale software systems connect people and provide new communication mechanisms that enable many benefits, from finding a long-lost friend to instant updates from family members. However, systems today are designed to maximize engagement and influence people. There exists a very limited understanding of how these systems influence the behaviors of individuals over time or how the aggregate behaviors of millions of people can be predicted. The limitations of engineering societal-scale systems under the existing state of the practice, which lacks the understanding and safeguards required to mitigate these issues, creates serious risks to both individuals and society as a whole because bias and misinformation create unforeseen and unrestrained consequences.

Although societal-scale systems are popular around the world and are among the most frequently used software systems, there are significant limitations with current practices for engineering these systems, including the following:

- Societal-scale software systems incorporate AI, create new information flows, and reshape societal knowledge, but there is little understanding of how a resulting socio-technical system will behave. People change their behaviors in response to these information sources in ways that are not always predictable or even visible [Centola 2018]. Understanding influence and response is made even more complex by the fact that the information flows change frequently and the information is not always vetted for truthfulness, bias, or manipulation.
- Today, societal-scale systems are developed primarily to maximize engagement through individual interactions that collectively create social networks. These social networks can be used to influence social behavior or perception. These technologies use highly targeted personalization to influence individual action. This influence can lead to polarization and can result in an aggregate warping of social knowledge. Studies indicate that engagement algorithms can have negative side effects that drive people toward extremes and can result

in new security risks [Carley 2020]. Current engineering practices have no framework to assess individual engagement algorithms or create safeguards against uncontrolled warping of social knowledge.

- Socio-technical systems provide direct connections to billions of people daily around the world, which dramatically increases the potential scale for social manipulation. Societal-scale systems use active interaction and vast quantities of interaction data compared to previous passive media (such as television), which enables targeted persuasion with unprecedented effectiveness and scale. This capability essentially democratizes influence, enabling manipulation by individuals, organizations, or nation states [Waltzman 2017]. The manipulation can be unintentional, due to a lack of understanding of these systems, or intentional, as nation states or rogue actors create online campaigns to manipulate populations. For many people, these systems create a different virtual society and enable communication with different rules and behavioral norms, which can lead to false consensus, bias, and polarization.

5.8.3 Topics for Research

Building new software engineering approaches for societal scale systems requires a cross-disciplinary approach. As stated in a recent research agenda from the Computing Community Consortium (CCC), software research "...will require a blend of humanities, social science, education, journalism, and computer science, with comprehensive support and participation from a broad range of organizations and institutions" [Bliss 2020]. Understanding what we can draw on from these disciplines will be critical to informing software engineering research to build the software systems of the future.

5.8.3.1 New Quality Attributes and Architectures

Societal scale socio-technical systems have new quality attributes that are not well understood. Today, most software architectures are created to support tradeoffs across well understood quality attributes, such as performance, reliability, and safety. Part of the goal of defining new quality attributes includes defining the metrics of merit and how to measure it.

The purpose of discussing quality attributes is to understand the relationship of design decisions in societal-scale systems and the behaviors of the systems in use. New approaches are needed to deal with the many dimensions of human behavior. Engineering systems with predictable impact on humans will help avoid surreptitious ideological influence. Trust, privacy, and bias are not totally new, but what is new is the challenge of predicting and monitoring these attributes.

We want to use the social sciences as a basis for understanding these quality attributes, similar to the way we use physical sciences as a basis for designing for other quality attributes. One approach is to separate the measurable manifestation of a quality attribute from the interpretation of *it.* For example, using votes as a measure of merit (like the best product or solution) seems to be useful, but also has challenges, including bias and understanding the biases in samples of voters. Additional considerations include the following:

- What are the new quality attributes of societal-scale socio-technical systems?
- How can we identify and capture societal-scale requirements?
- How do we relate individual choices to predict larger aggregate behaviors and determine whether they are within the "expected" range from an engineering perspective?
- How can we develop privacy models that scale across organizations and systems and enable individual control?
- How can we ensure our confidence in data?

5.8.3.2 Software Development Using Socio-Technical Systems

A special case occurs when socio-technical systems are used by software engineers to build software. This is an interesting use of societal-scale systems because of the influence of these systems on software engineers and software engineering activities. These systems and open source environments allow people to access huge amounts of code and create a type of digital infrastructure that both influences and is influenced by software engineers. Examples include:

- Stack Overflow, where people may use solutions without considering their origin or effects
- GitHub, where mining code could lead to assumptions about software qualities and possibly influence developers

- social coding environments, where popularity, number of users or followers, links, and a host of other types of social information are used in the technical decision-making process, such as the adoption of libraries and frameworks
- new ways of learning, from massive open online courses (MOOCs) which enable recognized experts to teach to anyone, to developers learning through YouTube tutorials

Researching how societal-scale systems influence software development is important to better understand how to improve software quality and to consider how these influences could be focused to help address workforce challenges. For example, if socio-technical systems influence software developers to make better decisions, then these systems could serve as one enabler for AI-augmented software development.

5.8.3.3 Analysis Tools that Support Emergent Social Network Topology

Architectural and software analysis tools must be automated to enable continuous analysis, but they must also be able to support emergent network topologies. The network topology, or prioritized network of data sources and influence mapping, changes because who a person connects to strongly impacts what they see. Historically, the assumption has been that systems were created to inform as opposed to influence people. The influence mapping topology constantly changes, and sometimes topics surge or "go viral," resulting in new and unexpected connections. Specific challenges in this area include the following:

- developing automated and continuous analysis of very large and emergent network topologies
- creating analysis tools that enable situational awareness and human understanding of very large-scale trends and predictions

5.8.3.4 Developing a Theory of Socio-Technical Knowledge Creation

Future software developers must consider multiple dimensions of individual interactions and understand how these individual interactions enable or create risks to society's general knowledge. Addressing this challenge requires understanding how knowledge is propagated and identifying mechanisms to provide "guardrails" that limit how much information is warped or influenced by speculation.

The communication mechanisms in socio-technical systems are new pathways of information flow, and they impact social structures, norms and understanding. Previously, information flowed from a few sources to large audiences through video, audio, or paper media organizations that generally included information about sources. While these media organizations were sometimes biased, they generally undertook serious efforts for verification or vetting. Today, information flows quickly through social networks without explanation of the source, little vetting of the

information, and frequent bias. Figure 3 illustrates a conceptual "funnel," where the data going into the funnel follows multiple steps before being considered fact or knowledge.[5] Different fields (e.g., journalism, intelligence, and academic fields) have different versions of the funnel, but knowledge-based professions all have ways of testing and selecting information to identify what is and is not well founded.

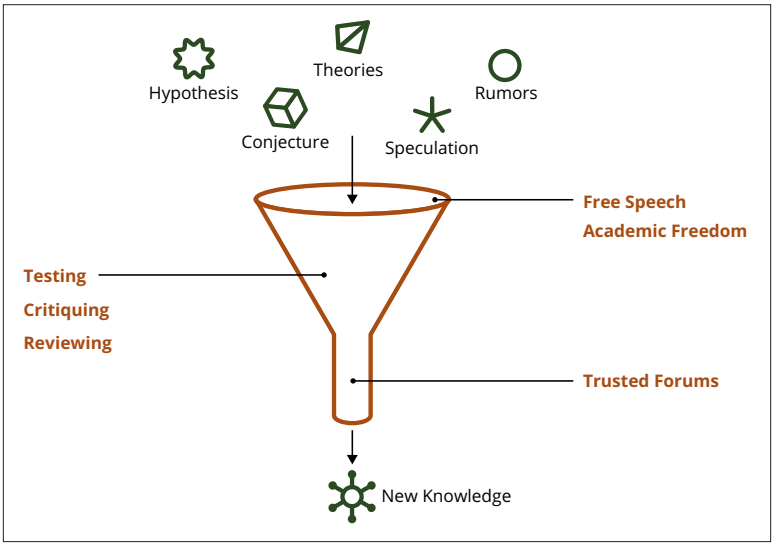


Figure 3: Conceptual Process for Creating Knowledge in Societal-Scale Systems

Current computational methods alone are insufficient to make the judgments needed to move data through the funnel. Specific areas of interest include the following:

- protecting social epistemology by identifying the knowledge funnel mechanisms [Rauch 2018]
- identifying the essential characteristics of effective online moderation systems for vetting data and rating data sources (e.g., Karma on Reddit)
- understanding and mitigating unintended consequences at scale in social media
- identifying and mitigating risks of social epistemology influencers that negatively impact knowledge creation

5 Graphic derived from Jim Herbsleb, Carnegie Mellon University, Institute for Software Research [Rauch, 2018].

5.8.3.5 Continuously Evolving Socio-Technical Systems

Automatically detecting and bounding behavior is critical to meeting the scale and volume of data in socio-technical systems. This requires consideration of the constant change and uncertainty in architectural evolution, data flows, and governance policies of these systems. Systems of the future must constantly monitor and adjust as the system changes.

It is important to highlight that the process of evolving societal-scale systems also requires understanding and adjusting for how they change the behaviors of the people using them. We don't fully understand what kinds of impacts these systems can have on people over time, and there is a range of impacts that need to be explored. This creates an interesting feedback loop in which the people using the system are also reshaping the system.

Continually evolving socio-technical systems includes questions such as the following:

- How can we control versions or fix bugs when these systems are constantly changing? Is it a new way of developing software when the system and data are never the same?
- How do we understand the state of the socio-technical protection mechanisms that lead to the knowledge that was created?

5.8.3.6 Protecting Against Misuse of Socio-Technical Platforms

Socio-technical systems connect billions of people every day and can have profound effects on society. Detecting societal-scale disinformation or manipulation is difficult because the information can take many forms, such as text, video, or audio [Twetman 2021]. Rapidly reacting to propagating misinformation in different types of data, at scale and fast enough to respond, requires significant architectural and data mechanisms.

Additional topics of interest include

- manipulating politics and public opinion, which present national security concerns, especially when done by foreign powers
- detecting and addressing unanticipated consequences
- tracking sources of content (i.e., provenance)

5.8.3.7 Adherence to Policy

Governments around the world are creating new policies and rules to govern socio-technical systems. These rules can vary at national, state, or even local government boundaries even though the socio-technical systems exist at a global scale. New mechanisms are needed to engineer these systems when policy rules vary across locations and provide the measurement and audit capability required for government regulation. It is important for software engineering research to create data-driven and openly understood techniques for industry, government, and society to establish a technically grounded and adaptable governance framework for societal-scale systems. This topic requires further research because engineers face many tradeoffs as governance addresses multiple considerations, including the following:

- how to require transparency when influence mechanisms are operating in the software
- how to monitor policy adherence while preserving the privacy of individuals
- how to protect against companies, governments, or other actors misusing the ability to filter content
- how to build automated monitoring and reporting for policy adherence

This topic requires further research because engineers face many tradeoffs as governance addresses how to protect against companies, governments, or other actors misusing the ability to filter content.

5.8.3.8 Experimentation and Testing

Socio-technical systems operate at such large scale, and with such diversity of human interaction, that typical experimentation and testing approaches are ineffective. Modeling is one option, but this requires confirmation that the system and the model are consistent. It also requires a deeper understanding of individual and aggregate behaviors. Another solution is experimentation and testing as part of the operational system, such as the A-B testing carried out by many tech companies. But, as a general strategy, this approach has technical challenges in bounding the experimentation. It also has potential legal and moral concerns related to human subject testing (testing the system without the knowledge of the people using it), potentially increasing the risk of manipulation.

A deeper understanding is needed about what testing really means for societal-scale systems and how experimentation and testing should be conducted. Some additional questions include the following:

- Should people affected by a system have a role in its design (i.e., participatory design)?
- How do we create an experimentation environment to explore these areas and test new approaches? (This question involves understanding the threshold of what is acceptable to test on live users versus what must be revealed to users to let them know they are part of a test.)
- If so many people depend on socio-technical systems, should there be reliability or safety expectations (like in telecommunications) to test when a main hub fails or to determine how a system can fail safely?

5.8.4 Research Questions

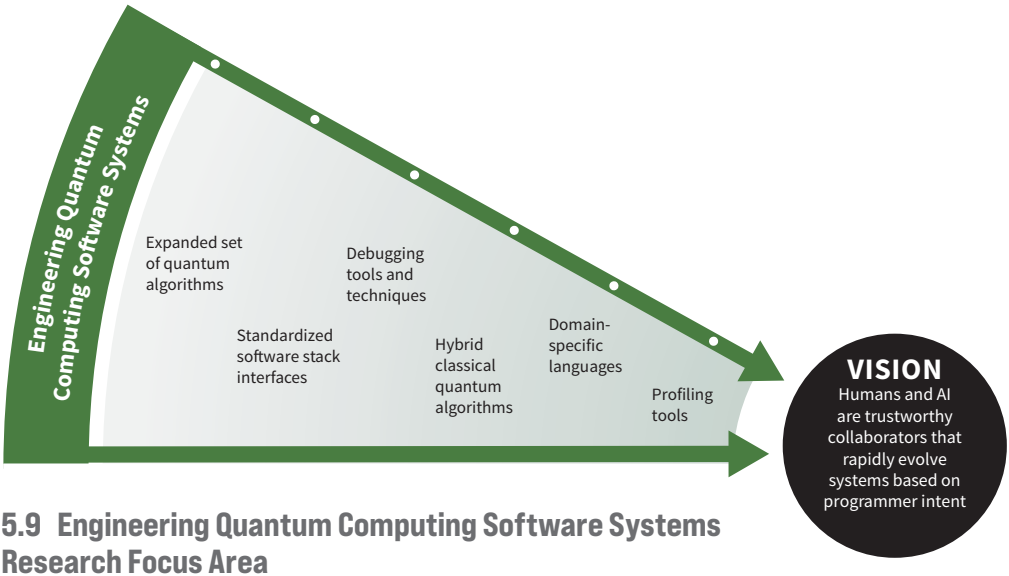
There are numerous challenges and questions involved in engineering societal-scale systems. Answering the societal-scale questions requires leveraging expertise outside of software engineering from the social and information sciences. The following questions highlight this point:

- How do we test solutions and fully consider the widely varying cultural and behavioral background of people?
- How do we monitor social epistemology impacts, and which social vetting processes are effective to guard against intentional or unintentional warping of knowledge?
- What are the software engineering expectations for transparency when using influence mechanisms in systems, particularly in health or financial systems?
- What are the important aspects of societal-scale measurement to enable continuous assessment of compliance of systems to evolving government policies, and when might the measurement infringe on privacy?

5.8.5 Research Topics

Research topics will be incremental and are summarized below.

- New quality attributes based on human behavior at scale. Identify new quality attributes and architectures that enable engineering prediction of human behavior at scale with consideration for uncertainty.
- System instrumentation to monitor effects of system on social behavior. Build continuously evolving socio-technical systems analysis tools, analysis tools that support emergent network topology, and new approaches for experimentation and testing.
- Automated detection and protection for misuse of socio-technical platforms. Develop a theory of societal-scale knowledge creation and develop protections against misuse of societal-scale platforms.
- Platforms for continuously evolving socio-technical ecosystems. Build continuously evolving societal-scale system analysis tools and support adherence to variable policy rules.



5.9 Engineering Quantum Computing Software Systems Research Focus Area

Quantum computing seeks to change the style of computation by leveraging quantum mechanical effects. In the 1980s and early 1990s, the theory of a new type of computer was developed: A quantum mechanical Turing machine was described and shown to be able to simulate everything a classical Turing machine could do. The basic unit of computation in these systems is not a classical binary one or zero—a bit—but, rather, an analog value that can represent intermediate values and simultaneously multiple values: both one and zero—a quantum bit, or qubit. This expressivity, called superposition, plus the ability of multiple qubits to be entangled—to join together to express states that cannot be expressed in terms of concatenated, separate, single-qubit systems—gives a quantum computer its power. In fact, a quantum computer with n qubits can encode 2n—1 complex numbers. Exciting theory was developed to show that quantum computers could, for select problems, be much more capable than classical computers. Such computers are not limited by the Church–Turing theory, which says that the performance of all classical computers can be only polynomially faster than a classical probabilistic Turing machine. Better yet, algorithms were soon developed that showed that a quantum computer could be exponentially faster than classical computers at solving specialized problems.

That groundbreaking theory motivated additional work on the design and realization of quantum computers. Two classes of quantum computers have been pursued: The first class comprises computers that initialize state and then evolve that state directly so that the final system state has a high probability of encoding the correct answer to the computation. This is called analog quantum computing, and includes the quantum annealing computers built by D-Wave Systems. The second class of quantum computers breaks computation down into a small set of primitive operations, then sequentially performs those operations, eventually producing a probably correct result. These are called gate-based systems, and are the systems we focus on here, because the discrete nature of these systems should enable the use of error correction, which will allow the systems to scale much larger and thus solve much more complex problems.

Small-scale implementations of these gate-based computers are starting to be developed from different technologies. There are many challenges in developing such systems—qubits need to be isolated from the environment, entangled, and precisely controlled, so it is still not known which qubit technologies will scale to the desired, large systems. Leading contenders include trapped ion qubits (explored by IonQ and Honeywell) and superconducting qubits (explored by IBM and Google), although academics and others are exploring photonic qubits and neutral atoms. Major providers of high-performance computing (e.g., IBM) and cloud computing services (e.g., Google, Microsoft, and Amazon) are enabling access to today's small quantum computers through their platforms. Today, the small size and noisy qubits of these systems have limited their applications; they have been used mostly for experimentation and scientifically interesting, but commercially unimportant, demonstrations. Still, the promise of such systems to compute things beyond classical computers is enticing. Recommendations in this section have been influenced by conversations with practitioners as well as by several excellent recent reports [Martonosi 2018; NASEM 2019].

5.9.1 Goals

If we imagine that hardware advances that permit scaling are achieved, then advances in software and software engineering will also be needed. The 2018 National Strategic Overview for Quantum Information Science [NSTC 2018] identifies grand challenges in areas such as ML, simulation of many-body systems for materials discovery, chemical processes, quantum field theory, and dynamics of biological processes. To this list we add software engineering.

For quantum computers, there is much to be done. We are working in a world with quantum computers of only 50-100s of barely functioning qubits—what John Preskill called "Noisy Intermediate-Scale Quantum" (NISQ) technology [Preskill 2018]. The software tools we have are powerful, yet each qubit is individually controlled by the programmer, with only limited automation. Our goals are to first enable these NISQ computers to be easily programmed and then to have increasing abstraction as larger, fully fault-tolerant quantum computing systems become available.

5.9.2 Limitations of Current Practice

There are so many missing software engineering pieces that it is easiest to think about the needed advances as layers in a software engineering stack. We group those advances in software into the following categories: quantum algorithms, software engineering, development tools and languages, computing platforms, and testbeds.

5.9.3 Topics for Research

5.9.3.1 Advances in Quantum and Classical Algorithms

In this section we consider software engineering for special-purpose systems that have not yet reached their full potential—universal gate-based quantum computers. Today's systems are small and unreliable, so the research described in this section that relies on large-scale systems may take more years to perform than other sections due to their lack of availability. Still, enough is known to start to envision the software engineering challenges and research required for such systems, even if the time frame may be less accurate.

Theoretical computer scientists focus on the asymptotic behavior of algorithms. Fast quantum algorithms are tuned to the unique aspects of a quantum computer: superposition and entanglement. These are both limited in scope and different in style from classical algorithms, requiring substantial creativity and expertise.

In fact, the total of all quantum algorithms that exist today that in theory could outperform a classical algorithm are listed online as the "Quantum Algorithm Zoo."[6] There are four major areas of such algorithms: (1) algebraic and number theoretic algorithms; (2) oracular or searching algorithms; (3) approximation and simulation algorithms; and (4) optimization, numerics, and machine learning algorithms.

Unfortunately, today's quantum computing systems perform individual calculations much more slowly (in wall-clock time) than classical computers. They have slower clock rates and produce probabilistic results, thus requiring either many runs or a means of checking the discovered solution. After the relatively limited performance of today's real hardware is factored in, only those algorithms that provide super-polynomial speedups are likely to achieve true quantum advantage. Only a few experiments have demonstrated quantum advantage—the ability to solve a problem faster than it could be solved on a classical computer. This will remain challenging, because both quantum and classical computing architectures will continue to improve.

The following are research topics for this area:

- Expand the set of known quantum algorithms.
- Leverage new insights from quantum algorithms to improve classical algorithms.
- Build hybrid algorithms that leverage the best quantum and classical algorithms working together.
- Develop provably correct libraries that can be called from higher-level languages.
- Develop benchmarks, so the performance of different machines can be compared.

6 **https://quantumalgorithmzoo.org/**

5.9.3.2 Advances in Software Engineering, Development Tools, and Quantum Computing Languages

Quantum computer tools are in their infancy. There are several quantum programming platforms. Microsoft offers a quantum development kit that allows programming in the object-oriented Q# language [Svore 2018]. Amazon offers a development kit that supports programming in the Bra-Ket language [Amazon 2021]. IBM offers programming in QISkit, a multi-layered language that supports programmers with expertise in other fields, quantum circuit developers, and quantum mechanics experts [Abraham 2019]. In addition, there are several academic (e.g., Scaffold) and open-source (e.g., Quipper) languages and tools. Both functional and imperative languages have been developed [Qiskit 2012; Gay 2006].

Many of these languages are low level, roughly akin to classical assembly language. This characteristic encourages the programmer to think about the unique aspects of quantum computers but makes it hard to think in terms of higher-level algorithms. It is not clear that this is the best model for developers, or that any one of these early languages is the best possible way to program. In addition, the toolchain has many simple components, and there are enticing hints that more sophisticated solutions will produce dramatic speedups. For example, efficient mapping computations from a language onto a specific quantum computer can double the performance of the algorithm.

The following are research topics for this area:

- Develop new domain-specific programming languages that allow the programmer to directly express quantum-unique parallelism while preventing impossible actions, thereby making the quantum programmer more efficient.
- Develop quantum compiler optimization techniques that map programming languages to multiple target architectures, thereby improving the runtime and efficiency of the implementation and enabling benchmarking.
- Develop tools to support continuous integration for quantum computers.

5.9.3.3 Advances in Hardware/Software Computing Platforms

In most cases, users access today's quantum computing systems via the cloud as a special-purpose co-processor of a classical computer. Most future algorithms will perform some actions (i.e., loading data and initializing the system) on a classical computer and some computations on a quantum computer. Deciding which computation to perform on what component will remain a challenge, as will measuring the performance of the system. Finally, debugging cannot be done on a quantum computer in the usual way.

The following are research topics for this area:

- Develop new tools for profiling quantum algorithms and hybrid classical-quantum algorithms.
- Develop new tools for debugging quantum algorithms and hybrid classical-quantum algorithms.
- Refine the interfaces: command-line, application-level, and application programming interfaces.

5.9.3.4 Advances in Simulators and Testbeds

Simulators and testbeds will be needed to advance the field. Simulators exist and can leverage multicore and high-performance computing infrastructures (e.g., QuEST and high-performance simulation of quantum computers) [Jones 2019]. At base, a simulator can be used to verify the expected outputs of a quantum computer. At higher levels of abstraction, a simulator can trace execution and reveal the state of logical qubits. There will be limits to these simulations, because current simulation techniques simulate gate operations using sparse matrix manipulation, where an N qubit computer grows as 2N. Current supercomputers can perform simulations of about 50-qubit systems.

The following are opportunities for possible research directions:

- Develop techniques to allow for larger simulations: new approaches, or new decomposition of existing approaches, enabling sequential simulation of multi-step and complex algorithms.
- Develop techniques to automatically allow comparisons of intermediate representations from a simulator and a quantum computer.

5.9.4 Research Questions

- What additional quantum algorithms are there?
- What new tools and techniques would be useful for debugging quantum algorithms?
- What new approaches, and new decomposition of existing approaches, would support larger-scale simulation of multi-step and complex algorithms?

5.9.5 Research Topics

It is difficult to order the tasks above, which are organized as a quantum software stack. However, there are three important epochs for quantum computers:

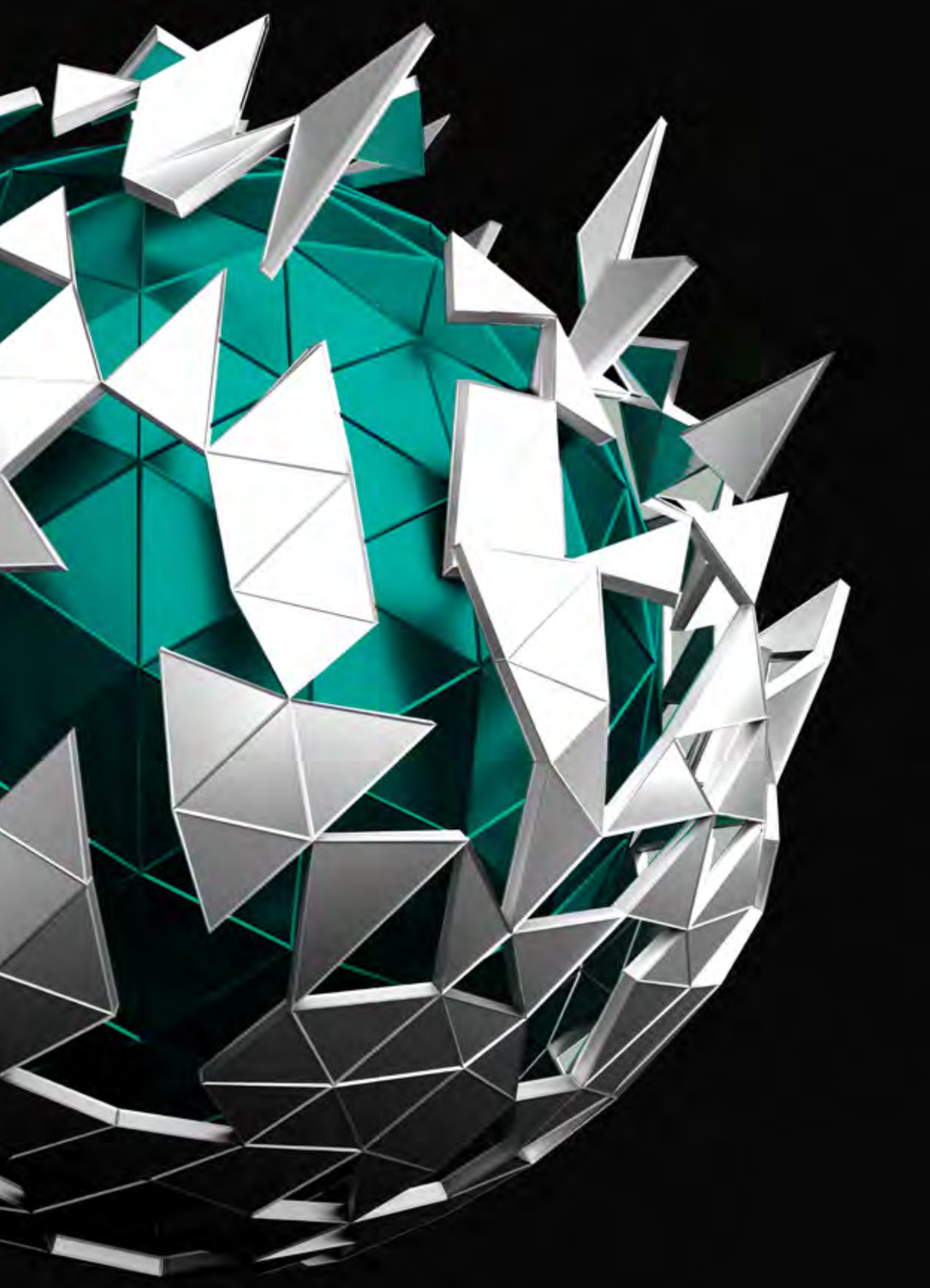
- Scientific: Learning how to build a quantum system that is perfectly isolated from the outside world, and yet the qubits in that system are able to strongly interact with each other. The problems that need to be solved are mostly problems for the field of physics. Software challenges include developing low-level tools, akin to device drivers, that coordinate the control signals to individual qubits. There is still a lot of interesting research to be done, but that is not the focus of this report.
- NISQ: This is the era when systems of 50 to a few hundred qubits are in the system, mostly isolated from the outside world, and able to interact with each other as desired. Several quantum computers have been built that reach this scale and performance. There are many interesting software engineering problems.
- Fault-Tolerant Quantum Computing: This is the era of a few hundred to millions of qubits, isolated from the external world and strongly interacting with each other in carefully controlled ways.

We subdivide milestones in the following table, focusing on problems of NISQ and Fault-Tolerant Quantum Computing, and forcing a selection of no more than two of the following categories:

- Short-term. These are efforts that should be pursued over the next couple of years.
- Mid-term. These are efforts that should be pursued over the next three-to-five years.
- Long-term. These are efforts that should be pursued over the next six or more years.

Table 1: Quantum Computing Research Milestone Time Frame

Research Direction	Short Term	Mid-term	Long-term
Benchmarks for quantum computing	X		
Large-scale simulation techniques	X		
Expanded set of quantum algorithms	X	X	
Quantum algorithm insights leveraged for classical algorithms	X	X	
Standardized software stack interfaces	X	X	
Intermediate comparisons: QC and simulators	X	X	
Debugging tools and techniques		X	X
Hybrid classical-quantum algorithms		X	X
Proven correct libraries		X	X
Domain-specific languages with greater abstraction		X	X
Target architecture mapping techniques		X	X
Tools for continuous integration		X	X
Profiling tools		X	X



6 Recommendations

This report is intended as a call to action in response to current and anticipated future deficits in software engineering capability. Both the research recommendations and the enactment recommendations in this section help to define the actions needed for the successful development of future systems.

The research focus areas in Section 5 led to the research recommendations, followed by a set of enactment recommendations that focus on people, investment, and sustainment as vehicles for change. Work should begin as soon as possible on implementing the recommendations in this section.

6.1 Research Recommendations

The research recommendations were motivated by the following key observations:

- AI is both a capability enhancer and a source of engineering uncertainty.
- As software pervades everything, it increasingly helps us imagine new ways it can be used. This leads to the engineering challenge of ensuring software evolvability while efficiently reassuring it.
- As software systems continue to grow in size and interconnectivity, evolvability will increasingly depend on engineering by composing and recomposing systems from existing pieces.
- In the past, social groups and societies were constrained to some extent by physical proximity. Now social media enables interactions at an enormous scale, virtually without limit. This leads to a need for developing new engineering principles for societal-scale, socio-technical systems.
- As software continues to touch almost everything, software-reliant systems inevitably become increasingly heterogenous, consisting of data, humans, organizations, sensors, different types of computational devices and other physical objects, and other elements. This heterogeneity of system parts brings engineering challenges due to the many disparate and interacting domains.

The following research recommendations are intended for public and private researchers and practitioners.

Recommendation 1—Enable AI as a Reliable System Capability Enhancer

The software engineering and AI communities should join forces to develop a discipline of AI engineering (perhaps starting with the Association for the Advancement of Artificial Intelligence (AAAI) and IEEE Computer Society). This would contribute to the development and evolution of AI-enabled software systems that behave as intended. Moreover, this would enable AI to be used as a software engineering workforce multiplier by helping with routine software engineering activities, such as generating code based on programmer intent aiding in refactoring, and ensuring conformance between a system's implementation and its architecture.

Recommendation 2—Develop a Theory and Practice for Software Evolution and Re-Assurance at Scale

The software engineering research community should develop a theory and associated practices for re-assuring continuously evolving software systems. A focal point for this research is an assurance argument, which should be a software engineering artifact equal in importance to a system's architecture. Research should include developing representations for assurance arguments and approaches for structuring assurance arguments so that small system changes only require incremental re-assurance.

Recommendation 3—Develop Formal Semantics for Composition Technology

The computer science community should focus on the newest generation of composition technology to ensure that technologies such as dependency-injection frameworks preserve semantics through the various levels of abstraction that specify system behavior. This will allow us to reap the benefits of development by composition while achieving predictable runtime behavior.

Recommendation 4—Mature the Engineering of Societal-Scale Socio-Technical Systems

The software engineering community should collaborate with social science communities to develop engineering principles for socio-technical systems. Theories and techniques from disciplines such as sociology and psychology should be used to discover new design principles for socio-technical systems, which in turn should result in more predictable behavior from societal-scale systems such as social media.

Recommendation 5—Catalyze Increased Attention on Engineering for New Computational Models, with a Focus on Quantum-enabled Software Systems.

The software engineering community should collaborate with the quantum computing community to anticipate new architectural paradigms for quantum-enabled computing systems. The focus should be on understanding how the quantum computational model affects all layers of the software stack. Predictably, exploiting quantum computing will require determining where the specifics of the quantum model should be hidden versus known by other elements of the software system.

6.2 Enactment Recommendations

While research recommendations focus on scientific and engineering barriers to achieving change, enactment recommendations focus on institutional barriers, such as economic, human, and policy barriers.

- It takes investment to fuel change. The institutional challenge is to compel investors.
- Regardless of the level of automation, human engineers build systems. The institutional challenge is to reimagine our software engineering workforce.
- Self-sustaining change requires institutionalization of policy and practices.

The following enactment recommendations are for research funders, policy makers, and industry leaders.

Recommendation 6—Investment Priority Should Reflect the Benefits of Software Engineering As a Critical National Capability

The software engineering community, software industry leaders, national labs, and federal departments should recognize software engineering as a national priority. This higher recognition in policy is needed to enable sustained government and industry investment in software engineering research, with benefits to national competitiveness and security.

The strategic role of software engineering in national security and global market competitiveness should be reflected in national research activities, including those undertaken by the White House Office of Science and Technology Policy (OSTP) and Networking and Information Technology Research and Development (NITRD). These research activities should recognize software engineering research as an investment priority on par with chip manufacturing and AI. For example, the risk of the U.S. economy being dependent on foreign chip manufacturing recently motivated multiple U.S. actions, including industry investments of around $50 billion and a proposed government investment of another $50 billion. AI technology investment has followed a similar path, where a possible U.S. technology gap motivated government investment from DAPRA and NITRD, along with major industry investments. In both of these examples, increasing awareness of the risks to national security and the U.S. economy motivated action that included industry and government investment. It is equally important to invest in software engineering research (see the table on the following page).

Without continual investment and improvement in software engineering technologies, next-generation applications will simply not be possible. Software, and therefore software engineering, is the common enabler of rapid innovation across most new technologies. As outlined in the research focus areas of this report, the software engineering technical challenges ahead require new solutions.

As stated in the 2018 National Defense Strategy, "The security environment is also affected by rapid technological advancements and the changing character of war. The drive to develop new technologies is relentless, expanding to more actors with lower barriers of entry, and moving at accelerating speed." The environment of rapid technological advancement highlights how quickly technology leadership can be lost, and the lower barrier of entry for software means that nation states, or even non-state actors, can quickly leverage technology.

Software engineering grand challenges sponsored by DARPA, the National Science Foundation (NSF), and FFRDCs are also suggested. Grand challenges have become an effective way to quickly mobilize existing capability on critical issues while enabling new partnerships across academia and industry.

Table 2: Investment in U.S. Software Technology

Chip Manufacturing

Risk: U.S. economy dependent on foreign chip manufacturing

- US capacity fell to approximately 13% in 2015 compared to 30% in 1990 and 42% in 1980
- 2020–2021: World-wide shortages post pandemic

U.S. Actions

- 2017: President's Council of Advisors on Science and Technology (PCAST) report on US Leadership in Semiconductors
- 2020–2021: Intel more than $20 billion, Taiwan Semiconductor Manufacturing Company (TSMC) more than $30 billion in U.S. fabrication investments
- 2021: $50 billion request in president's budget goals

AI Technology

Risk: U.S. AI technology gap compared to other nation states

- many nations interested, but becoming a two-nation race
- multiple nations announcing multi-billion-dollar investments in AI

U.S. Actions

- 2018: DARPA "AI Next" $2 billion
- 2019: Executive order AI strategy and investment
- 2021: Networking and Information Technology Research and Development (NITRD) investments—#1 of 12

Software Engineering Research

Risk: Software engineering advances have not kept up with the critical nature of software for U.S. national security and competitiveness.

This is important because

- software is the backbone of all critical systems
- software includes complex supply chains
- software is infrastructure

Initial U.S. Actions

- 2019–2020: NITRD Future Computing Community of Interest; National Strategic Computing Initiative Update; and Software Productivity, Sustainability, and Quality Working group
- 2021: CMU SEI A National Agenda for Software Engineering Research & Development study

Recommendation 7—Institutionalize Ongoing Advancement of Software Engineering Research

The software engineering community, software industry leaders, national labs, and federal departments should recognize software engineering as a national priority. This higher recognition in policy is needed to enable sustained government and industry investment in software engineering research, with benefits to national competitiveness and security.

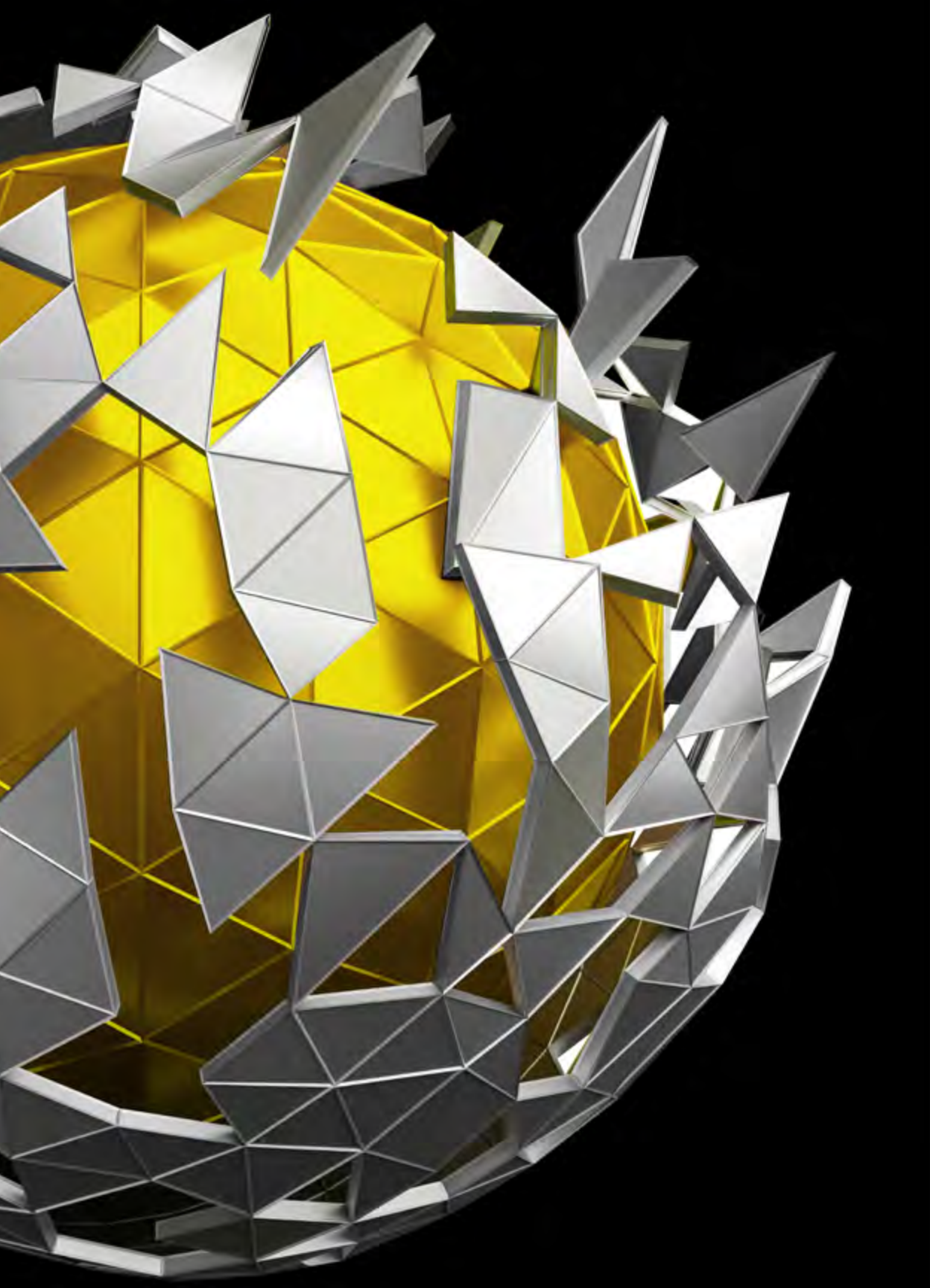
Sustained advancements in software engineering require institutionalizing an ongoing review and reinvestment cycle for software engineering research and its impact on software engineering practice on the fabric of our software engineering ecosystem. Maintaining national software engineering proficiency requires research funding sources and institutes working with industry and government leaders in the software engineering community to periodically review the state of software engineering. The DoD's National Defense Strategy (formerly the Quadrennial Defense Review) can serve as an exemplar. The responsibility for ensuring that such reviews take place should be part of a high-level influential organization such as OSTP or President's Council of Advisors on Science and Technology (PCAST).

Recommendation 8—Develop a Strategy For Ensuring an Effective Workforce for the Future of Software Engineering.

Currently, software engineering is performed by a broad collection of people with an interdisciplinary skill set that does not always include formal training in software engineering. Moreover, the nature of software engineering seems to be changing in reaction to the fluid nature of software-reliant systems. Because of these trends, the traditional areas of expertise, such as architecting, designing, implementing, and testing, could give way to other specializations.

We need to better understand the nature of the needed workforce and what to do to foster its growth. The software engineering community, software industry, and the academic community should create a strategy for ensuring an effective future software engineering workforce.

These issues and other consequences of this research roadmap need to be studied, to lead to a detailed set of workforce recommendations.



7 Conclusion

Architecting the Future of Software Engineering: A National Agenda for Software Engineering Research and Development is the result of a yearlong, community-based activity to re-validate the importance and centrality of software engineering; identify current and future challenges in the discipline; and develop a research agenda to catalyze the software engineering ecosystem to prepare for the future.

We conclude this document by summarizing several insights we gained (or re-gained) along the way.

Software touches all aspects of life and all aspects of infrastructure; is key to research in many disciplines; and, in general, is important to all aspects of national security. Thus, software is an important enabler of ever-more aspects of society. It is hard to look around you and not see something with software inside. The importance and pervasiveness of software naturally implies that we have to pay careful attention to how we construct and evolve it. That is, we have to pay careful attention to software engineering. Otherwise we risk having software engineering become not just an enabler of new capability, but also a source of vulnerability.

Software engineering was originally conceived in the spirit of other, older engineering disciplines, but it seems to be finding a niche of its own due to its unique nature. Perhaps, unlike other engineering disciplines such as civil engineering, software engineering will not evolve into a "mature" discipline. If it does not, it will be because, after its practice become routine, it will soon thereafter be automated, and software engineering will advance to tackle a new challenge. Then, due to the conceptual nature of software, the practice of software engineering will continue to grow and change—without bounds—in capability, complexity, closeness to other domains, and interconnection. There seems to be no plateau in the advance of software and, therefore, no end for challenges in software engineering.

The continual advance of software is a natural driver for automation, which leads to increasing responsibilities and authority for the software that helps humans create and evolve software. AI is playing an important role in helping software tools move beyond their role as mere extensions of programmers. It is also creating a new role for engineers as peers and, ultimately, collaborators with AI. This expanded role will enable the engineering of software, in part, by allowing software engineers and eventually users to "program" software by letting it know what it is they expect it to do. Programming through intent in this way will become an important specialty.

Increased reliance on software drives the need to continuously and rapidly change it. Indeed, what used to be a "want" has become a "need" for defense systems. Nimble threats drive the need for nimble responses and enhanced capabilities through better sensors. And AI analytics drive the need for rapid fielding. These new capabilities, in turn, drive new mission concepts. Rapid change requires rapid re-assurance, which makes it increasingly important to structure evidence and assurance arguments in a way that allows re-assurance to be done incrementally and compositionally.

Future research must be planned with the software ecosystem in mind, and it must be representative of key software engineering challenges. Software engineering can be examined along two orthogonal dimensions: "doing software" and "what software does." "Doing software" led us to consider several advanced development paradigms. We focused on three that seem to be central to the changing nature of software engineering described above: the drive for increased automation; the need for re-assurance; and the vital nature of compositionality. "What software does" led us to consider three types of challenges that could give rise to advanced architectural paradigms: humans as part of the system; AI components in a system; and quantum processors as an exemplar of a new computational model.

Our goal in looking at software engineering in this way was to be representative and encompassing without having to be exhaustive. We hope this framework invites others to consider other advanced development paradigms that are key to the changing nature of software engineering and other types of challenges that might give rise to additional advanced architectural paradigms, thereby extending the roadmap that we've started.

Ongoing self-assessment needs to be institutionalized. As noted before, there seems to be no end in sight for advances in software engineering. This means that software engineering requires ongoing reflection and incentives to support advancement. This, in turn, requires a high-level advocate, along with funding and periodic reviews to continue what this study started.

Software has sometimes been compared to air: It's invisible and everywhere, and everyone and everything needs it. This feeling can lead to two different ways of considering software and, hence, software engineering: (1) letting it remain invisible and taking it for granted, or (2) nurturing it, caring for it, protecting it, and improving it. We hope this report has convinced you that there really is only one viable way of considering software engineering.

Appendix A: Engaging the Software Engineering Community Through Workshops

The following workshops provided the opportunity to gain perspectives from diverse audiences and are summarized in the following sections:

- *A-1: National Agenda for Software Engineering R&D Workshop—Software Engineering Researcher Edition*

  Focus: Leverage significant experience of Carnegie Mellon SEI researchers and technical staff to identify future trends in software engineering.

- *A-2: Voice of the Customer Workshop*

  Focus: Obtain perspective of future-leaning software engineering trends and challenges from the point of view of our customers.

- *A-3: Future Scenarios Workshop—Developing Plausible Alternative Futures*

  Focus: What role will software engineering play in our nation's security in 2030?

- *A-4: DoD Senior Leaders Workshop*

  Focus: Perspectives on software challenges and future demands from DoD Senior Leaders.

- *A-5: Software Engineering Grand Challenges and Future Visions Workshop*

  Focus: SEI-DARPA hosted workshop with the software engineering research community to outline software engineering's key research challenge areas for the next decade.

Appendix A-1: National Agenda for Software Engineering R&D Workshop—Software Engineering Researcher Edition

Workshop Date and Goal
May 1, 2020. Leverage experience of SEI senior researchers and technical staff to identify future trends in software engineering.

Workshop Pre-work
Attendees were asked to consider five questions and send a short whitepaper with their ideas and responses to one or more of the following questions:

1. What do you think will be examples of "software systems" of the future? (For example, considerations might include quantum, hypersonics, autonomous, intelligent systems, and so forth.)
2. How will software systems of the future be developed?
3. What are key technical challenges and breakthrough ideas in software engineering?
   (For example, consider challenges in areas such as architecture, assurance, acquisition, model-based software engineering, autonomy, DevSecOps, AI, resilience, and so forth.)
4. What other types of advances aren't on our radar yet—but should be?
5. Discuss ideas for a research roadmap: How can we energize and coordinate all of the needed constituencies to achieve the desired future?

More than 60 responses were received and a cross section of topics were presented as five-minute lightning talks.

Guest Speaker Intro Talk
The workshop opened with guest speaker Keith Webster, Carnegie Mellon University Dean of University Libraries and Director of Emerging and Integrative Media Initiatives. His presentation was Envisioning the Future with a Futurist: Introducing the Notion of Futures and Foresight Studies.

Lightning Talks and Themes from Lightning Talks

Thirty-six SEI technical staff members presented lightning talks. Designated "listeners" collected key ideas and themes in real-time from these talks as related to the five questions asked in pre-work, as follows. (This is a sample list and not intended to be comprehensive.)

1. What do you think will be examples of "software systems" of the future?
   - commodity heterogenous interconnected computing platforms
   - prevalence of embedded systems
   - new domains in space
   - massively distributed computing
   - cognitive architectures
   - voice to code—voice to simulate—voice to test
   - software-defined everything
   - software developed by open source community supported by industry users
2. What things do we need to consider for future systems?
   - computing solutions in socio-embedded computing
   - increasing security and reliant systems
   - importance of human interaction in the creation of future systems
   - new quality attributes including observability
   - operating in uncertain environments
   - design automation to support systems of the future
   - ethical human software development
   - tiered workforce in software engineering (the job of the programmer is changing dramatically)
   - how to engineer quantum systems at scale (a very challenging field)
   - container-based deployment
   - automated virtual testing
   - data development process
3. How will software systems of the future be developed (i.e., themes for developing future systems)?
   - certification and validation
   - extending DevSecOps
   - AI and ML approaches
   - ethics of software development and encapsulated in software solutions

- heavy focus on human-system interaction
- new demands on embedded systems
- distributed engineering
- automated repair algorithms
- "no code" solutions are gaining traction in industry
- notion of "family of systems" architectures
- human factors in software engineering

4. What are the key technical challenges and breakthrough ideas in software engineering? (For example, in areas such as architecture, assurance, acquisition, model-based software engineering, autonomy, DevSecOps, and AI)?
   - Trends toward new systems will lead to new software development challenges in
     - automation of software development and deployment
     - development and sustainment of AI systems
     - primacy of data
     - formal methods and proof engineering
     - human-centered engineering approaches
     - hierarchical certification approaches
     - adoption challenges and need for suitable context
     - preparing the workforce of tomorrow and moving toward a tiered workforce in software development
     - new programming languages that automate complex/common functions
     - proof engineering
5. What other types of advances aren't on our radar yet—but should be?
   - hyper automation and new software development languages
   - hyper agility and need for tools and methods to accelerate that agility
   - hyper computation and need to validation and verification, modeling, and simulation
   - social views of software engineering including workforce perceptions and accessibility to more people
   - causal inference

Discuss ideas for how to generate a research roadmap: how to energize and coordinate all of the needed constituencies to achieve the desired future.

- There is a systematic process for envisioning the future…we practice for the future (it doesn't just happen).
- Bring futures thinking to the National Agenda for Software Engineering.
- There is no single future "out there" to be predicted. There are many alternative futures to be anticipated and pre-experienced to some degree.
- Consider horizons of change, including the following:
  - expected future
    - where we are headed
    - the future if everything continues as it has
    - the result of conditions and trends (momentum)
  - alternative futures
    - what might happen instead
    - the set of plausible futures if something less likely or unexpected happens
    - the result of events and issues (contingencies)
  - preferred future(s)
    - what we want to happen
    - either the expected or any of the alternative futures that is preferable
  - Think about WILDCARDS.
  - The software engineering tools and approaches must enable agility to respond to more frequent and sometimes rapid shifts in the future.
  - The future comes from the commercial side and the importance of understanding the vectors of industry.
  - It is important to consider tech transition and understand what can be consumed.

Appendix A-2: Voice of the Customer Workshop

Workshop Date and Goal
May 20, 2020. Obtain perspective on future-leaning software engineering trends and challenges from the point of view of DoD and industry collaborators.

Prior to the workshop, participants were asked to share their thoughts on one or more of six questions, in short lightning talks (about 5 minutes each). A summary of the responses is listed below.

1. Based on what you are hearing from our customers, what do you think will be examples of software systems or software missions of the future? (For example, consider quantum, hyper sonics, autonomous, intelligent systems, and so forth.)
   - no code/low code
   - families of architecture models, different architectures, and new quality attributes
   - insatiable quest for data to satisfy data-to-decision
2. How will software systems of the future be implemented?
   - mission engineering with development of systems
   - speed of the warfighter
3. What are the key software challenges our customers are having or foresee in the future?
   - trust and assurance of systems
   - legacy systems, legacy systems, legacy systems
   - code analysis and ML automated architectural analysis/refactoring
4. What role do our customers want to see from the SEI as software becomes more ubiquitous in their systems?
   - independence
   - knowledge of topics with real world data
   - help people think when they don't have time to do so
5. What other types of advances aren't on our radar yet—but should be?
   - data visualization
   - integrating data into decision making
   - industry leading the way
6. Discuss ideas for a research roadmap: How can we energize and coordinate all of the needed constituencies to achieve the desired future?
   - create a research influencer map
   - acquisition as an advantage for warfighter

Appendix A-3: Future Scenarios Workshop: Developing Plausible Alternative Futures

Workshop Date and Goal

May 29, 2020. Find out from thought leaders what role software engineering plays in our nation's security in 2030.

Workshop Objectives

1. Learn how to think about and devise some futures scenarios with Keith Webster, Carnegie Mellon University Dean of University Libraries & Director of Emerging and Integrative Media Initiatives.
2. Pilot a Zoom futures scenarios workshop with breakout groups with a small number of participants.
3. Learn from the experience and consider applying or adapting this method to other working sessions for the National Agenda for Software Engineering Study.

Workshop Pre-Work

The participants were asked to answer 3 questions before the workshop to kickstart discussion.

1. What are the five to ten key drivers and trends you see impacting the framing questions over the next decade (and beyond)?
2. What questions would you like to be able to answer using the scenarios we'll produce as a result of our work?
3. What decisions do you need to make in the near-term that would benefit from a sense of the long-term?

Workshop Summary

The workshop participants voted and identified the following drivers and trends for the future of software engineering:

- explosion in the amount of software and software complexity in the world and extending into new areas (ubiquity and complexity)
- increase of software in critical roles without appropriate attention into safety, security or society risk
- increasing government and corporate access to personal data and increasing concern about privacy
- AI/ML driving automation and innovation
- access to computing devices, sensors, bandwidth and continuous connectivity
- automated tools for data collection and analysis, for architectural consistency, software quality, vulnerability discovery and repair
- policies gaining traction in many countries to reduce cooperation and collaboration between nations

- deployed software creating increasingly disruptive unintended effects
- changing nature of education and training including content, delivery, and access
- policies gaining traction in many countries to reduce cooperation and collaboration between nations
- deployed software creating increasingly disruptive unintended effects

The workshop participants voted and identified the following uncertainties:

- level of international collaboration
- supply chain – software, talent
- AI/ML (incl access to data and privacy)
- funding and partnership
- data as a strategic asset
- societal consequences of software deployment
- engineering of systems
- access to technology and connectivity

Appendix A-4: DoD Senior Leaders Workshop

Workshop Date and Objectives

June 25, 2020

1. Hear directly from DoD senior leaders (PEOs, PMs, services and intel agencies) in 10-minute lightning talks to understand their challenges and ideas about the future of software engineering.
2. Provide a forum for DoD senior leaders to hear challenges and ideas from other DoD senior leaders about the future of software engineering.

Summary of Workshop Ideas

- Speed, speed, speed. Iterate quickly and often, but with the feedback and discipline needed for assurance.
- There is a need for an organic capability in DoD for software engineering, including the culture, training, career development, and other elements needed to support the build vs. buy paradigm.
- Building in security and ensuring security throughout the lifecycle is a key concern. Topics mentioned included Risk Management Framework (RMF), continuous authority to operate (ATO) that addresses security, developing a "risk of use sticker," Cybersecurity National Action Plan, whitelist interactions, and zero trust architecture.
- Modeling to enable rapid development and deployment should be a focus. There is a need for high fidelity digital models and common standards so systems can integrate with models and with each other.
- The software acquisition system is antiquated and doesn't support speed and iteration. There are "color of money" problems and also cultural problems, such as how we communicate progress, schedule, and cost to stakeholders.
- Better ways are needed for funding foundational or common shared services, which is impossible with the current funding and program element structure.
- System boundaries have evolved or fallen away because everything is connected to everything with massive complexity and non-deterministic behavior—and interacting with and maintaining legacy systems adds to the challenges.
- Resilient operations in denied environments is a must (denied comms, cloud, data access, positioning, navigation, and timing (PNT), etc.).
- The interconnected nature of the software challenges is apparent. Multiple stakeholders emphasized how getting software right requires also getting many other elements right, including workforce, acquisition roles, technical solutions, infrastructure, and data.

Appendix A-5: Software Engineering Grand Challenges and Future Visions Workshop

Workshop Date and Goal
December 1-3, 2020. Outline software engineering's key research challenge areas for the next decade.

Workshop Overview
Society's dependence on software has only accelerated and broadened in recent years, and the software engineering and research communities have continued to focus on specific topics or innovations. However, there is also value in looking further ahead at the wider discipline of software engineering and envisioning the future we can create: Based on where we are today, where will innovation take us in the next 5, 10, or 20 years? And what do we need to do to prepare the future?

To answer these questions and begin to envision the future of software engineering, the SEI, in collaboration with the Defense Advanced Research Projects Agency (DARPA), convened the Software Engineering Grand Challenges and Future Visions Workshop, conducted Tuesday, December 1 through Thursday, December 3, 2020. This workshop aimed to spur a discussion among leading researchers, practitioners, and government stakeholders and promote communication within and among these communities. Its goal was to stimulate new thinking, articulate future needs, and discuss how emerging and/or disruptive software engineering technologies, methods, and tools can help us address those needs.

Participants
The workshop participants were drawn from the following communities:

- academic researchers whose work is having (or likely will have) a fundamental impact on the way software and software-reliant systems will be developed
- leaders in companies now developing leading-edge software by creating and applying software advances at scale
- thought leaders familiar with the defense mission and threat space who are working to implement acquisition practices that support current and future capability needs

Position Papers and Talks
Participants were asked to contribute a one-to-two-page position paper describing

1. a vision of the types of systems that will need to be developed in the future (5, 10, or 20 years out)
2. major open problems and "grand challenges" software engineering must address to make those systems technically and/or economically feasible

Contributors were also free to identify associated research areas or themes. Selected participants provided short lightning talks based on their position paper. The organizers and participants were particularly open to visions that aimed to rethink the foundations of software engineering.

The position papers and lightning talks reflected a mix of today's important research areas as well as disruptive technologies likely to have an impact on the field of software engineering but which have not yet commanded wide notice. Topics included artificial intelligence (AI), machine learning (ML), automated testing and cybersecurity tools, cyber-physical systems, socio-technical systems, and formal models.

Focus Questions

The workshop comprised three half-day sessions of facilitated discussions designed to address areas of need and potential impact without precluding interesting avenues of inquiry. Areas of discussion focused on the following:

- What are the major open problems (and grand challenges) that software engineering must address?
- What software research (whether in research labs, university, industry, government, or some mix of these) is needed to invent solutions for those problems?
- What role should collaboration between industry and academia play in developing and adopting solutions? What role should the government play?
- What can incentivize strategic collaborations among government, academia, and industry?

Expected Outcomes: Grand Challenges and Visions

The three-day workshop was designed to produce grand challenges and vision statements, scenarios that describe

- an important class or classes of future software and software-reliant systems
- software engineering research methods, tools, and practices that are needed to make those systems feasible

These outcomes are intended to serve as inputs to ongoing efforts to define new research programs and initiatives that can shape the future of software engineering and a research roadmap that will help us close the gap between today's capabilities and the futures we envision.

Seeding the Discussion

To set the stage for the discussion, three keynotes were presented, and all participants were asked to contribute short lightning talks to introduce key concepts. Keynote addresses were provided by Sandeep Neema, Program Manager at DARPA's Information Innovation Office; Sol Greenspan, Program Director in the National Science Foundation's (NSF) Directorate for Computer & Information Science and Engineering (CISE); and Christopher Ré, Associate Professor of Computer Science at Stanford Artificial Intelligence (AI) Lab. These keynotes described the software engineering research landscape from the point of view of DARPA, NSF, and the Stanford AI Lab.

Keynote Summary One—Sandeep Neema, DARPA: Software and Defense

In his keynote, Neema stressed the importance of software to the DoD. He observed that the growth and complexity of software are astounding, and noted that every generation of a given system is more complex by an order of magnitude (for example, the F-35). Neema underscored the reality that almost 90 percent of system functionality is now realized through software. "Mission success depends on high-quality software," he said, also noting that the DoD is in the business of software. Drilling down, Neema noted that the field of software V&V has been a prolific area of research: The results have been impressive and a portfolio of V&V techniques is now available. However, these advances haven't netted the desired results. "Given innovation in tools and methods for V&V," said Neema, "one would hope software quality has improved significantly, but recent results indicate things are no better than a decade ago."

Neema noted the number of software vulnerabilities has grown over time in Microsoft and Android systems, and the time from discovery to patch is significant. Neema observed that things are not much different in the realm of weapons programs: Despite advances in software analysis and verification, software quality is not much better. "Moore's law may be dead," added Neema, "but systems are still advancing at a fast pace and becoming more sophisticated and technologically advanced." Military systems are evolving rapidly, human-machine partnership are becoming important, and hybrid domains have emerged. "We need to straddle core tech, engineering, and mission applications and build a pipeline that can continuously deliver new capabilities," said Neema.

He then identified four broad research areas:

- proficient AI
- advantage in cyber operations
- resilient adaptable and secure systems
- confidence in the information domain

In his concluding remarks, Neema noted challenges in the areas of resiliency, rapid evolution in the face of nimble threats, bolt-on security, cyber-physical systems and distributed IoT, safe data sharing, and sustained configuration integrity. His ideas about addressing these challenges included targeted modeling and analysis and rich toolchains.

Finally, Neema cited some specific areas of interest to DARPA:

- verification-friendly systems engineering
- intersection of AI, ML, and software engineering
- software comprehension and maintenance
- process engineering

Keynote Summary Two—Sol Greenspan, National Science Foundation (NSF): Tracing the Evolution of Software Engineering through Past Workshops

Greenspan framed his remarks in the context of two previous workshops conducted in 2001 and 2010 to place the current workshop in context. The 2001 workshop, "New Visions for Software Design & Productivity: Research & Applications" (sponsored by Vanderbilt University, the Networking and Information Technology Research and Development (NITRD) Program, the National Science Foundation, and others) examined the question "What can we do well?" Greenspan noted that 2001 workshop found requirements remained a problem; embedded and networked systems were considered important; there was a need to identify what programming environments were needed; testbeds were needed that simulate operational environments; and there was a tension between informal and formal methods and questions about the nature of the balance between the two and whether that balance should change. Participants also questioned whether software development should be more fluid (Agile), which was hard to achieve in DoD and government settings because of the budgeting and oversight process.

The consensus of the 2001 workshop was there had been productivity gains in lines of code per person, and new ideas and tools had emerged in the form of middleware, GUI generators, APIs, MDE, and application frameworks. Participants then asked, "Why can't we declare victory?" To address then-emerging challenges, participants at the 2001 workshop recommended the following:

- specification and management of complex requirements (especially for embedded and networked systems)
- better software development environments with domains-specific capabilities for validation
- testbeds that simulate operational situations (environment conditions, user interactions)

The 2010 "Workshop on the Future of Software Engineering Research" was also sponsored by NITRD and was collocated with the "ACM SIGSOFT Eighteenth International Symposium on the Foundations of Software Engineering." In that workshop, participants examined how we:

- help people produce and use software
- build the complex systems of the future
- create dependable software-intensive systems
- invest in research to improve software decision-making, evolution and economics
- invest in research to improve software research methodology

As with the 2001 workshop, these questions continue to have resonance.

Recommendations from the 2010 workshop included developing social connections for software engineering stakeholders; democratizing and broadening participation in production and use of software (e.g., end-user programming); addressing societal grand challenge problems of unprecedented complexity and scale (e.g., the SEI ultra-large systems report); exploiting emerging technology and platform opportunities (e.g., app stores); automating software evolution; strengthening empirical research foundations; and incorporating social science research.

Though certain questions and challenges persist, there have been some changes since 2010. For instance, the amount of data involved has increased significantly. Software engineering has become a big data science. Also, we have witnessed the emergence of issues concerning of the naturalness of software code; static and dynamic testing and analysis; heightened demand for reliability, robustness, and resilience; and integration of computing, sensing, communications, and new devices (e.g., 5G, IoT, and nano). In light of these new issues, the field of software engineering should be thinking about the following:

- Software-reliant systems will need to contain more knowledge about the world.
- Our environments will need to have embedded software engineering capabilities and infrastructure.
- Systems will need to be held accountable to laws, compliance rules, and societal norms.

Keynote Summary Three—Christopher Ré, Stanford AI Lab: Software 2.0

Ré opened his remarks by stating that Software 2.0 is eating Software 1.0. As an example of the power of Software 2.0, Ré cited Google's translation tool, which shrank from 500 thousand lines of code (LoC) to 500 lines of dataflow. Ré noted that Software 2.0 includes AI applications, and that the software engineering and/or design element is really changing (again citing the shift from LoC to data flow code). This shift has already has had an impact on products we use today, including Spotlight, Safari, and assistants. Core pieces of the software are rewritten in this way. What's more, this shift is not restricted to large companies.

Ré observed that something is changing in what we're building. When ML is used in a core way to build apps, it changes what you do as a software engineer. Models have become commodities. Engineers are not always writing new code but, rather, servicing and understanding new models. Weakly and naturally supervised systems are big changes for Software 2.0.

Ré discussed Overton, a data system for monitoring and improving ML products. Ré wanted to change the conversation about ML systems. He cited the "Transformer model." The model didn't matter. Rather, what mattered was having tools to understand workflow.

"We wanted low-margin parts of the job automated," he said. "We can automate huge parts of the stack and concentrate on what the user actually needs." This, according to Ré, would prevent the phenomenon of "new-model-itis." There's been a shift from building models to support model building to monitoring quality and improving supervision.

Ré noted a number of challenges in this new environment:

- Many decisions are required, all of which need to be right.
- ML products have complicated pipelines; seams exist between products; and it's hard to share code and/or ideas.
- ML products require fine-grained monitoring and involve rare queries and/or complex disambiguation.
- Quality is rapidly improving.
- Horrible errors are easy to make.

In short, he said, we are entering the age of ML as an engineering discipline.

Lightning Talk Summaries

Software and Missions to Space—As NASA's missions become increasingly dependent on correctly functioning software, code is growing and becoming more complex. Likewise, the safety and reliability of software has become more crucial, but software has also become harder to test and verify. Consequently, the following areas have grown in importance: requirements assessment; software code quality and risk assessment; automated software architecture analysis; research correlated to selected metrics on defects to assess code quality; and automation of software engineering and assurance.

Workforce—What will the software engineering workforce of the future look like? A need exists to need to uniformly train the workforce through a cross-matrix curriculum. Presently, approaches to training differ. There is disparity in the quality and consistency of training. We all train differently, yet expect to collaborate effectively. One possible approach could take inspiration from maneuver warfare, in which distributed forces are placed in different, vulnerable areas to remove a "center of gravity." Eliminating cultural inhibitors to change, including software training practices, was one suggestion.

Scalable Assurance and Cyber-Physical Systems—The scalable assurance challenges of cyber-physical systems (CPS) include the kinetic effect of CPS, which often are safety-critical systems, such as airplanes, requiring strong assurance. However, strong assurance has not been practical because of multi-criticality (assurance levels, timing, real-time mixed-trust computation) and artifact size (too large for strong verification techniques). Challenges also exist regarding the problem of cognitive design overload in large systems. To address these problems of scale, work is needed in the following research areas:

- multi-criticality: real-time mixed criticality with temporal protection
- artifact size: minimize verified components, add enforcers to guard critical properties, protect the enforcers, and enforce critical aspects
- cognitive design overload: model-based engineering

"Multi-Everything"—Systems need to be "multi-everything." ML-enabled software needs to be able to analyze code to ensure confidence and meet requirements and use cases. We need easy migration between cloud services and post-deployment monitoring that's not expensive. We also need to make data a first-class citizen: proprietary data formats are problematic. There is a further need for a variable-trust data environment, which requires security without latency, and lifecycle management is necessary for ML-produced software.

The Needs of the Scientific Community—Performance, portability, and productivity are keys to the future of computing. High-performance computing has changed qualitatively and quantitatively. The software community has seen the advent of many-core and multicore systems, heterogeneous computing, machine learning, and an enormous increase in the diversity of the hardware ecosystem in both edge and high-end computing. Great diversity in capabilities exists. The demand for software in scientific computing has been growing. It outstrips supply and the gap has grown worse over time—a key challenge the scientific ecosystem requires the field to address. Hardware–software codesign, more intelligent ways of producing software for simulation and analysis, and more intelligence in composing scientific workflows are needed.

The Demands of International Development Projects—International conversations are taking place about software. Language needs to be appropriate and culturally inclusive, and the same holds true for software. A need exists for commonality of language and architecture: Software is global conversation. Coding for key systems is taking place around the world. With this in mind, all partners should share a common language, ethical and legal frameworks, and a commonality of commercial off-the-shelf (COTS) and military off-the-shelf (MOTS). Challenges exist related to the notion of the "data lake." How can various international partners working on a project absorb the information they need from the lake? There is also a need to get individuals with the right skills and accreditations into Department of Defense (DoD) in the global marketplace, as well as a need for automated testing, certification, and verification and validation.

Software Warning Lights—Software warning lights are behavior guards in the application domain intended to protect the public's health, safety, and welfare. Such warning systems are common in network operation centers, but they've yet to see use at the application level. We need this kind of cyber-anomalous detection system for the warfighter—a warning light that allows the warfighter to see anomalies in software in real time.

Societal-Scale Systems and Social Media—Societal-scale systems, such as social media, present concerning unintended consequences. These unintended consequences are a national security threat because they dramatically increase the potential for social manipulation. We need to adopt a behavioral science view to inform systems that determine what is true. The notion of social epistemology presents knowledge creation as a social process. This isn't working in our societal-scale systems. We need methods, for instance, to funnel free speech in such systems through testing, critique, review, and trusted forums to create new knowledge. Such a process is critical to determining what's true. In the current state of affairs, social media is focused on engagement, not knowledge. Consequently, we see the rise of so-called "alternative facts," conspiracy theories, et cetera as unintended side effects.

An Army Software Factory—Hardware-intensive systems have become software-intensive systems, and traditional methods are too slow, too expensive, and of little value. The defense community would be well served by building organic competencies that meet or exceed skills in the private sector. A move to in-house development for the DoD would enable it to serve as its own integrator, but such a move presents many challenges.

Cyber-Resilient Software Ecosystems—Cyber resilience on platforms is a problem, and there are larger challenges in ecosystems, including monocultures, security fragmentation, and a "libertarian" design ethic. Not much has been done to manage and/or patch monocultures. Tools are needed to make it easier to deliver implementation diversity. Key stakeholders tend to assume that someone else is taking responsibility for overall design but, when application developers follow an ungoverned path, they can inadvertently increase the attack surface of the apps they develop. We don't build guards and monitors into applications. But, we need these things for a resilient ecosystem.

Keys for Faster Iterative Requirements Engineering—We can build better and more ethical software. The field is not taking advantage of AI. Some software development is being done with AI, but it's clumsy and requires manual tinkering with AI tools. One approach to these challenges lies in the concept of keys: the few controllable variables of a model that control decisions about non-dependent clashing links in chains of reason that link inputs to desired goals. If keys work, they could lead to a dramatic reduction in modeling effort, more AI-assisted software engineering, better software engineering, and ethical software engineering for AI. Keys have the potential to enable developers to very quickly reason about a system.

Self-Supervised Systems—Entity disambiguation systems are built with supervised training, making them brittle and expensive. This challenge is being addressed with fully self-supervised systems, which are easier to maintain and extend. With such systems, engineers focus on monitoring the model. This approach has been deployed in industry, and this fundamental shift to self-supervised systems is not going away. However, while self-supervised systems make things easier for engineers, they do pose challenges. For instance, how does the system handle what's not in training data? How do you monitor the model and unit test? How does the system correct errors on the fly once it's deployed?

Bridging the Gap between Formal Methods and System Assurance—Component specifications for safety could offer a way to bridge the gap between formal methods (math) versus system assurance (a condition in which it's acceptably rare that the system will cause harm). A number of challenges to achieving this goal exist. For instance, can formal methods be used on real-world systems? Yes, but it's expensive. Proofs need to be handwritten by experts. As for system assurance, the challenge lies in

determining how you know when analysis is complete. Questions also arise concerning how repeatable analysis is. The keystone that holds all this together is a rich component specification on which system assurance tooling can be automated and built.

Certifying Adaptive Dynamic Computing Environments— Presently, certifying adaptive dynamic computing environments presents a challenge. "The right answer delivered too late is the wrong answer." Employing dynamic resource management to support multiple missions simultaneously requires adaptive computing resource management. However, certifying such as system to ensure reliability is not a solved problem and is becoming more complex with AI and other advanced technologies.

The Future of Air Travel—Autonomy will be important. There will be many autonomous vehicles in the air as well as on the ground, which will necessitate integration across air space. These developments present challenges in safety certification, affordability, training, system definition and requirements, and full integration of systems.

New System Categories—New system categories have emerged: systems at ultra-large, societal-scale (here defined to include systems such as connected and autonomous vehicles, transactive energy distribution, and low-altitude air traffic control); human-AI-machine teams (such as medical assistive robotics and enhanced reality environments); and multi-technology fusion (such as the Microsoft Premonition program integrating IoT technologies, metagenomics, ecological and epidemiological simulations, and cloud computing for global monitoring of the biome). Software technology is focused on gluing together heterogenous systems at ever-bigger scales. The resulting challenges include developing platforms for system integration, platforms for model integration, and platforms for integrating AI and ML components. Corollary research areas include a formal foundation for system integration, a formal foundation for model integration, and compositionality for AI and ML components.

Software Engineering Research in Production—Partnerships need to be established to solve large-scale computing problems, but it's hard to make these partnerships happen. Technical innovation is needed in the areas of data anonymization and privatization, observability, secure computation-like data analysis, and measurement frameworks. Furthermore, cultural changes are needed: Reviewers need to accept industrial case studies, and industry and government need "matchmaking" with academia. Policy changes on non-disclosure and data use agreements are needed to shrink the divide among industry, government, and academia to establish stronger partnerships so research can better represent what's going on.

The Role of Digital Twins—We are in an age of software transformation. The future will bring autonomous transportation, human augmentation (biotech), and smart infrastructure (nanotech). These developments will have an impact on software, which will need to adopt a cross-discipline focus, require a low and/or reduced context ("How can we make building blocks?"), and demand safety and security assurance-test-driven development. One possible solution to these challenges is to integrate digital twin with model-based engineering.

Workshop Summary: General Discussion Topics

While the workshop discussions focused on a number of specific technical areas, overall they provided interesting insight into the state of research and practice. In this section, these insights are collected into broad groups.

Tension between Speed and Trusted Capability

Many of the discussions focused on the increasing tension between the speed of new capability deployment and the need for "exquisite engineering" (i.e., the engineering effort needed to develop and field the high-quality systems that we trust with many aspects of our modern lives and livelihoods). The need for trust in systems is greater than ever—not only for CPS, but also for societal-scale systems. In the words of one workshop participant, "We live in software." This statement reflects the degree to which many aspects of our day-to-day lives are now supported by, controlled by, or influenced by software systems. Given the plethora of such systems and the difficulty of understanding emergent behaviors, there is a renewed focus on building these systems to be correct and auditable by construction.

Participants also highlighted the need to make the software producing workforce more productive and efficient to meet the engineering demand for evermore software systems whose behaviors require evermore confidence. Some potential advancements in this direction include greater emphasis on the following items:

- platforms and/or environments that help raise the quality of software by construction (via automation and machine teaming)
- democratizing development
- tools that leverage the heterogeneity of the workforce
- low-code and/or no-code solutions

Assured Composition

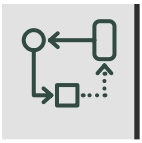
Several participants commented on an emerging development paradigm, "assured composition," that aims to deal with the issues noted above. Assured composition reflects a need to better understand emergent behaviors that result from chance or design in these systems.

Research at Scale

A related emerging research paradigm, "research at scale," reflects a need to better understand phenomena in systems at scale that are usually not easily accessible and/or available to the research community. These large-scale systems pose new challenges and require new approaches for researchers to contribute meaningful results that can impact the state of the practice.

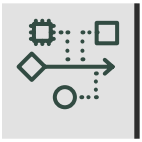
Focus Themes

Five major themes emerged over the course of the workshop, Illustrated in Figure 4. By design, these were not decided ahead of time but rather emerged out of participant discussions. Some of these themes were raised and discussed explicitly; others were created to summarize several related conversations that happened over the course of the event.



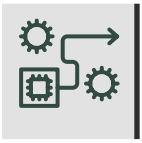
Assuring Continuously Evolving Systems

Provide evidence and arguments that a system will behave as intended, considering both desired functionality and quality attributes, as it evolves continuously to incorporate new capability and dynamically self-adapts its operating configuration at runtime.



AI-Augmented Software Development

Augment each stage of software development with AI to orchestrate continuous systems evolution, positioning for constant high speed change.



Engineering of AI-Enabled Software

Develop empirically validated practices to support development and sustainment of next generation AI-Enabled software. Provide tools, verification methods, techniques and practice to apply sound software engineering principles to AI Engineering.

Designing in Ethics in Software, Systems, and Societal-Scale Systems

Build and evolve societal-scale software systems that enable transparency and mitigate risks of unethical influence on individuals, unrestrained social manipulation, or disruption of social epistemology.

Composable Software Systems

Provide a scientific and engineering basis for designing, building, analyzing, and assuring heterogeneous and composable software systems. Provide languages, tools, environments, and techniques to support these activities.

Figure 4: Summary of Major Themes Emerging from the Workshop

References

URLs are valid as of the publication date of this document.

[Abhari 2012]
Abhari, A. J. et al. Scaffold: *Quantum Programming Language.* TR-934-12. Princeton University. 2012. **https://www.cs.princeton.edu/research/techreps/TR-934-12**

[Adee 2020]
Adee, S. What Are Deepfakes and How Are They Created? *IEEE Spectrum.* April 29, 2020. **https://spectrum.ieee.org/what-is-deepfake**

[Ahmad 2021]
Ahmad, Norita; Laplante, Phil; & Defranco, Joanna. Life, IoT, and the Pursuit of Happiness. *Computing Edge.* August 2021.

[Alur 1992]
Alur, R. & Dill, D. The theory of timed automata. In: de Bakker J.W., Huizing C., de Roever W.P., Rozenberg G. Real-Time: Theory in Practice. REX 1991. *Lecture Notes in Computer Science.* Volume 600. 1992. **https://doi.org/10.1007/BFb0031987**

[Alur 1993]
Alur, R.; Courcoubetis, C.; Henzinger, T.A.; & Ho, P.-H. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems.* Grossman, R.; Nerode, A.; Ravn, A.; & Rischel, H. [editors]. Springer. Pages 209–229. 1993.

[Amazon 2021]
Amazon. Amazon Braket: Explore and experiment with quantum computing. *Amazon Web Services, Inc.* June 3, 2021 [accessed]. **https://aws.amazon.com/braket/**

[Amershi 2019]
Amershi, S. et al. Software Engineering for Machine Learning: A Case Study. Pages 291-300. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* Montreal, QC, Canada. 25–31 May 2019. **https://ieeexplore.ieee.org/document/8804457**

[Bader 2021]
Bader, Johannes; Kim, Sonia; Luan, Frank; Chandra, Satish; & Meijer, Erik. AI in Software Engineering at Facebook. *IEEE Software.* July/August 2021.

[Bass 2012]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice.* 3rd Ed. 2012. Addison-Wesley. ISBN-13: ?978-0321815736.

[Benveniste 2015]
Benveniste, Albert et al. *Contracts for Systems Design: Theory.* RR-8759. Inria Rennes Bretagne Atlantique; INRIA. 2015. **https://hal.inria.fr/hal-01178467**

[Bitdefender 2020]
Bitdefender. Mid-Year Threat Landscape Report 2020. *Bitdefender.* 2020. **https://www.bitdefender.com/files/News/CaseStudies/study/366/Bitdefender-Mid-Year-Threat-Landscape-Report-2020.pdf**

[Bliss 2020]
Bliss, Nadya et al. An Agenda for Disinformation Research. *Computing Community Consortium.* 2020. **https://arxiv.org/ftp/arxiv/papers/2012/2012.08572.pdf**

[Bojkic 2020]
Bojkić, Marko; Pržulj, Đorđe; Stefanović, Miroslav; & Ristic, Sonja. Usage of Dependency Injection within Different Frameworks. Presented at *19th International Symposium INFOTEH-JAHORINAAt*-Jahorina. March 2020.

[Bosch 2020]
Bosch, J.; Crnkovic, I.; & Olsson, H.H. Engineering AI systems: A research agenda. *Cornell University ArXiv.org*. June 3, 2020. **https://arxiv.org/abs/2001.07522**

[Bourque 2014]
Bourque, P. & Fairley, R.E., eds. *Guide to the Software Engineering Body of Knowledge, version 3.0.* IEEE Computer Society. 2014. **www.swebok.org**.

[Broy 2018]
Broy, Manfred. Yesterday, Today, Tomorrow: 50 Years of Software Engineering. *IEEE Software.* September/October 2018.

[Carleton 2020]
Carleton, Anita; Harper, Erin; Menzies, Tim; Xie, Tao; Eldh, Sigrid; and Lyu, Michael. Expert Perspectives on AI. *IEEE Software*. Volume 37. Number 4. Pages 87–94. July–August 2020, **https://ieeexplore.ieee.org/document/9121622**.

[Carleton 2020]
Carleton, Anita; Harper, Erin; Menzies, Tim; Xie, Tao; Eldh, Sigrid; &Lyu, Michael. The AI Effect: Working at the Intersection of AI and SE. IEEE Software. July/August 2020.

[Carley 2018]
Carley, K.M.; Cervone, G.; Agarwal, N.; & Liu, H. Social Cyber-Security. In *Social, Cultural, and Behavioral Modeling*. Thomson, R.; Dancy, C.; Hyder, A.; & Bisgin, H. [editors]. Springer. Pages 389–394. 2020. **https://link.springer.com/chapter/10.1007/978-3-319-93372-6_42**

[CBS 2020]
CBS News. Toyota "Unintended Acceleration" Has Killed 89. *CBS News.* May 25, 2010. **https://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/**

[Centola 2018]
Centola, Damon. *How Behavior Spreads: The Science of Complex Contagion.* Princeton University Press. 2018. ISBN: 978-0-691-17531-7.

[Chaki 2011]
Chaki, S.; Gurfinkel, A.; & Strichman, O. Time-bounded analysis of real-time systems. Pages 72-80. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD) 2011*. Austin, Texas. October 30–November 2, 2011.
https://ieeexplore.ieee.org/document/6148914

[Charette 2005]
Charette, Robert N. Why software fails. *IEEE spectrum.* Pages 42-49. Volume 42. Number 9. 2005.

[Costa 2017]
Costa, F.; Morris, K.; Kon, F.; and Clarke, P. Model-Driven Domain-Specific Middleware. Pages 1961-1971. In *Proceedings of 37th International Conference on Distributed Computing Systems (ICDCS).* 2017. doi: 10.1109/ICDCS.2017.197.

[Council 2021]
Council, Jared. Where Business Is Using AI. *The Wall Street Journal.* March 10, 2021.

[Czerwinski 2021]
Czerwinski, Mary; Hernandez, Javier; & McDuff, Daniel. Building an AI that Feels: AI Systems with Emotional Intelligence Could Learn Faster and Be More Helpful. *IEEE Spectrum.* May 2021.

[Dao 2021]
Dao, Nhu-Ngoc; Quoc, Viet-Pham; Dinh-Thuan; & Dustdar, Schahram. The Sky is the Edge. *Computing Edge.* August 2021.

[Dean 2021]
Dean, Brian. Social Network Usage & Growth Statistics: How Many People Use Social Media in 2021? *Backlinko.* April 26, 2021. **https://backlinko.com/social-media-users**

[Debray 2020]
Debray, Vidroba & Miller, Seneca. Overcoming Challenges With Continuous Integration and Deployment Pipelines. *IEEE Software.* May/June 2020.

[Demi 2021]
Demi, Selina; Colomo-Palacious, R.; & Sánchez-Gordón, M. Software Engineering Applications Enabled by Blockchain Technology: A Systematic Mapping Study. *Applied Sciences*. Volume 11. No. 7. March 25, 2021. **https://www.mdpi.com/2076-3417/11/7/2960**

[Denney 2018]
Denney, Ewen & Pai, Ganesh. Tool support for assurance case development. *Automated Software Engineering.* Volume 25. Number 3. September 2018. Pages 435–499. **https://link.springer.com/article/10.1007/s10515-017-0230-5**

[DIB 2019]
Defense Innovation Board. *Software is Never Done: Refactoring the Acquisition Code for Competitive Advantage.* May 2019. **https://media.defense.gov/2019/Apr/30/2002124828/-1/-1/0/SOFTWAREISNEVERDONE_REFACTORINGTHEACQUISITIONCODEFORCOMPETITIVEADVANTAGE_FINAL.SWAP.REPORT.PDF**

[Divine 2020]
Divine, John. Tech Stocks: Is 2020 the Year Software Eats the World? *U.S. News and World Report.* May 20, 2020. **https://money.usnews.com/investing/stock-market-news/articles/tech-stocks-is-2020-the-year-software-eats-the-world**

[DoD 2018a]
Office of the Deputy Assistant Secretary of Defense. *DoD Digital Engineering Strategy.* Department of Defense. June 2018. **https://fas.org/man/eprint/digeng-2018.pdf**

[DoD 2018b]
Department of Defense. *Summary of the 2018 National Defense Strategy of the United States of America:* Sharpening the American Military's Competitive Edge. U.S. Department of Defense. 2018. **https://dod.defense.gov/Portals/1/Documents/pubs/2018-National-Defense-Strategy-Summary.pdf**

[DSB 2016]
Defense Science Board. *Summer Study on Autonomy.* June 2016. **https://www.hsdl.org/?view&did=794641**

[DSB 2018]
Defense Science Board. *Design and Acquisition of Software for Defense Systems.* Defense Science Board, U.S. Department of Defense. Office of the Under Secretary of Defense for Research and Engineering. February 2018. **https://dsb.cto.mil/reports/2010s/DSB_SWA_Report_FINALdelivered2-21-2018.pdf**

[Eaton 2021]
Eaton, C. & Volz, D. U.S. Pipeline Cyberattack Forces Closure. *The Wall Street Journal,* May 8. 2021. **https://www.wsj.com/articles/cyberattack-forces-closure-of-largest-u-s-refined-fuel-pipeline-11620479737**

[Ebert 2018]
Ebert, Christof. 50 years of software engineering: Progress and perils. *IEEE Software.* Pages 94-101. Volume 35. Number 5. 2018

[Feiler 2006]
Feiler, Peter H. et al. *Ultra-Large-Scale Systems: The Software Challenge of the Future.* Software Engineering Institute. 2006. ISBN 0-9786956-0-7. **https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30519**

[GAO 2019]
United States Government Accountability Office. *Agencies Need to Develop Modernization Plans for Critical Legacy Systems.* GAO-19-471. U.S. Government Accountability Office. June 11, 2019. **https://www.gao.gov/assets/700/699616.pdf**

[Gay 2006]
Gay, S. J. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science.* Volume 16. Number 4. Pages 581–600. July 24, 2006.

[Ge 2018]
Ge, N. Dieumegard, A.; Jenn, E.; and Voisin, L. Corrrect-by-contruction Specification to Verified Code. *Software: Evolution and Process.* Wiley. June 25, 2018. **https://onlinelibrary.wiley.com/doi/10.1002/smr.1959**

[Gil 2019]
Gil, Yolanda & Selman, Bart. *A 20-Year Community Roadmap for Artificial Intelligence Research in the US.* Computing Community Consortium. August 2019. **https://cra.org/ccc/wp-content/uploads/sites/2/2019/08/Community-Roadmap-for-AI-Research.pdf**

[Gillberg 2020]
Gillberg, A. & Holst, G. *The Impact of Reactive Programming on Code Complexity and Readability: A Case Study.* Institute of Computer and Systems Sciences. Mid Sweden University, Östersund, Sweden. 2020.

[Halon 2019]
Halon, Eytan. SpaceIL reveals preliminary reasons behind Beresheet crash. *The Jerusalem Post.* April 18, 2019. **https://www.jpost.com/Israel-News/SpaceIL-reveals-preliminary-reasons-behind-Beresheet-crash-587117**

[Hamilton 2018]
Hamilton, Margaret. What the Errors Tell Us. *IEEE Software.* September/October 2018.

[Hammond 2019]
Hammond, Ray. Megatrends of the 21st Century. *Allianz Partners.* June 2019. **https://www.allianz-partners.com/content/dam/onemarketing/awp/azpartnerscom/reports/futorology/Allianz-Partners-Megatrends-of-the-21st-Century-ENG.pdf**

[Harman 2015]
Harman, M.; Jia, Y.; & Zhang, Y. Achievements, Open Problems and Challenges for Search Based Software Testing. Pages 1–12. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST).* Graz, Austria. April 13–17, 2015. **https://ieeexplore.ieee.org/document/7102580**

[Hassan 2008]
Hassan, A.E. The road ahead for mining software repositories. Pages 48–57. In *Proceedings of the 2008 Frontiers of Software Maintenance.* Beijing, China. September 30–October 2, 2008. **https://ieeexplore.ieee.org/document/4659248**

[Hemon 2020]
Hemon, Aymeric; Fitzgerald, Barbara Lyonnet; & Rowe, Frantz. Innovative Practices for Knowledge Sharing in Large-Scale DevOps. *IEEE Software.* May/June 2020.

[Hendrycks 2019]
Hendrycks, Dan; Mazeika, Mantas; Kadavath, Saura; & Song, Dawn. Using Self-Supervised Learning Can Improve Model Robustness and Uncertainty. In *Proceedings of Advances in Neural Information Processing Systems 32 (NeurIPS 2019).* Vancouver, BC, Canada. December 2019. **https://proceedings.neurips.cc/paper/2019/file/a2b15837edac15df90721968986f7f8e-Paper.pdf**

[Holland 2020]
Holland, Charles & Tanenbaum, Jacob. *Emerging Technologies 2020: Six Areas of Opportunity.* Carnegie Mellon University, Software Engineering Institute. December 2020. **https://resources.sei.cmu.edu/asset_files/WhitePaper/2020_019_001_651791.pdf**

[Horneman 2019]
Horneman, Angela; Mellinger, Andrew; & Ozkaya, Ipek, *AI Engineering: 11 Foundational Practices.* Carnegie Mellon University, Software Engineering Institute. 2019. **https://resources.sei.cmu.edu/asset_files/WhitePaper/2019_019_001_634648.pdf**

[Ivers 2020]
Ivers, James; Ozkaya, Ipek; Nord, Robert L.; & Seifried, Chris. Next generation automated software evolution refactoring at scale. Pages 1521–1524. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Association for Computing Machinery, New York, New York. 2020. **https://dl.acm.org/doi/pdf/10.1145/3368089.3417042**

[Jansen 2019]
Jansen, Slinger; Cusumano, Michael; & Popp, Karl Michael. Managing Software Platforms and Ecosystems. *IEEE Software.* May/June 2019.

[Jensen 2021]
Jensen, Jakob Jul. Applying a Smart Ecosystem Mindset to Rethink Your Products. *Computing Edge.* June 2021. **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9269897**

[Jones 2019]
Jones, Tyson; Brown, Anna; Bush, Ian; & Benjamin, Simon C. QuEST and High Performance Simulation of Quantum Computers. *Scientific Reports.* Volume 9. Number 10736. July 24, 2019. Pages 1–11. **https://www.nature.com/articles/s41598-019-47174-9.pdf**

[Kazman 2020]
Kazman, Rick & Pasquale, Liliana. Software Engineering in Society. *IEEE Software.* January/February 2020.

[Kim 2019]
Kim, Miryung. *Re-engineering Software Engineering for a Data-centric World.* Presented at The 34th IEEE/ACM International Conference on Automated Software Engineering. November 2019. **http://web.cs.ucla.edu/~miryung/ASE2019-Keynote-MiryungKim-SE4DA.pdf**

[Kirbas 2021]
Kirbas, Serkan et al. On the Introduction of Automatic Program Repair in Bloomberg. *IEEE Software.* April 25, 2021. **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9395171**

[Klieber 2016]
Klieber, William & Snavely, Will. Automated Code Repair Based on Inferred Specifications. Pages 130–137. In *2016 IEEE Cybersecurity Development (SecDev 2016).* Boston, Massachusetts. November 3–4, 2016. **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7839806**

[Krasner 2021]
Krasner, Herb. The Cost of Poor Software Quality in the US: A 2020 Report. *Consortium for Information & Software Quality.* January 1, 2021. **CPSQ-2020-report.pdf** (it-cisq.org)

[Krishnan 2020]
Krishnan, Mahesh. *Evolution in the Automation of CI/CD.* Presented at the 35th IEEE/ACM International Conference on Automated Software Engineering. September 2020. **https://youtu.be/4f7sCrEtM30**

[Kwiatkowska 2019]
Kwiatkowska, Marta. Safety and Robustness for Deep Learning with Provable Guarantees. Page 2. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Tallinn, Estonia. August 2019. **https://dl.acm.org/doi/10.1145/3338906.3342812**

[Lago 2015]
Lago, Patricia; Malavolta, Ivano; Muccini, Henry; Pelliccione, Patrizio; & Tang, Antony. The Road Ahead for Architectural Languages. *IEEE Software.* Volume 32. Number 1. January–February 2015. Pages 98–105. **https://ieeexplore.ieee.org/abstract/document/6756703**

[Lanowitz 2021]
Lanowitz, Theresa. Ransomware and Energy and Utilities. *AT&T Blog.* June 3, 2021. **https://cybersecurity.att.com/blogs/security-essentials/ransomware-and-energy-and-utilities**

[Law 2021]
Law, Kincho & Lynch, Jerome. Smart City: Technologies and Challenges. *Computing Edge.* June 2021.

[Le Goeus 2019]
Le Goues, Claire; Pradel, Michael; & Roychoudhury, Abhik. 2019. Automated Program Repair. *Communications of the ACM.* Volume 62. Number 12. Pages 56–65. November 21, 2019. **https://cacm.acm.org/magazines/2019/12/241055-automated-program-repair/fulltext**

[Le Goeus 2021]
Le Goeus, Claire; Pradel, Michael; Roychoudhury, Abnik; & Chanra, Stish. Automatic Code Repair. *IEEE Software.* July/August 2021.

[Levitt 2019]
Levitt, Mark. Software Development—by 2040. *Trixta.* January 25, 2019. **https://medium.com/trixta/software-development-by-2040-14e39f9153cb**

[Lewis 2021]
Lewis, G.; Bellomo, S.; & Ozkaya, I. Characterizing and Detecting Mismatch in Machine-Learning-Enabled Systems. In *1st International Workshop on AI Engineering—Software Engineering for AI collocated with ICSE 2021.* Virtual workshop. May 30, 2021. **https://arxiv.org/pdf/2103.14101.pdf**

[Lira 2019]
Lira, C.; Mello, B.; & Prazeres, C. Pages 1243–1251. Reactive Microservices for the Internet of Things: A Case Study in Fog Computing. In *SAC '19: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. April 2019.
https://doi.org/10.1145/3297280.329740210.1145/3297280.3297402.

[Mahe 2020]
Mahe, Nolwen; Adams, Bram; Marsan, Josianne; Templier, Mathieu; & Bissonnette, Sylvie. Migrating a Software Factory to Design Thinking: Paying Attention to People and Mind-Sets. *IEEE Software*. March/April 2020.

[Maksimov 2019]
Maksimov, Mike; Kokaly, Sahar; & Chechik, Marsha. A Survey of Tool-Supported Assurance Case Assessment Techniques. *ACM Computing Surveys (CSUR)*. Volume 52. Number 5. October 2019. Pages 1–34.
https://dl.acm.org/doi/pdf/10.1145/3342481

[Marr 2020]
Marr, Bernard. The Top 10 Breakthrough Technologies for 2020. *Forbes*. February 26, 2020. **https://www.forbes.com/sites/bernardmarr/2020/02/26/mit-names-top-10-breakthrough-technologies-for-2020/?sh=279ddb71d482**

[Martonosi 2018]
Martonosi, Margaret & Roettele, Martin. Next Steps in Quantum Computing: Computer Science's Role. *Computing Community Consortium Catalyst*. November 2018. **https://cra.org/ccc/wp-content/uploads/sites/2/2018/11/Next-Steps-in-Quantum-Computing.pdf**

[McFadden 2021]
McFadden, Christopher. What's the Biggest Software Package by Lines of Code? *Interesting Engineering*. July 15, 2021.
https://interestingengineering.com/whats-the-biggest-software-package-by-lines-of-code

[McFall-Johnsen 2020]
McFall-Johnsen, Morgan. Catastrophic software errors doomed Boeing's airplanes and nearly destroyed its NASA spaceship. Experts blame the leadership's 'lack of engineering culture.' *Business Insider*. February 29, 2020.
https://www.businessinsider.com/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2

[Military.Com 2015]
Military.Com. Software Glitch Causes F-35 to Incorrectly Detect Targets in Formation. *Military.Com*. March 24, 2015.
https://www.military.com/defensetech/2015/03/24/software-glitch-causes-f-35-to-incorrectly-detect-targets-in-formation

[MIT 2021]
MIT Technology Review. 10 Breakthrough Technologies 2021. *MIT Technology Review*. February 24, 2021. **https://www.technologyreview.com/2021/02/24/1014369/10-breakthrough-technologies-2021/**

[Mitchell 2010]
Mitchell, Robert L. Toyota's Lesson: Software Can Be Unsafe at Any Speed *Computerworld*. February 5, 2010.
https://www.computerworld.com/article/2467572/toyota-s-lesson--software-can-be-unsafe-at-any-speed.html

[Moore 2016]
Moore, Andrew; O'Reilly, Tim; Nielsen, Paul; & Fall, Kevin. Four Thought Leaders on Where the Industry is Headed. *IEEE Software 33:1*. January–February 2016. Pages 36–39. **https://doi.org/10.1109/MS.2016.1**

[Morrison 2018]
Morrison, Patrick; Pandita, Rahul; Xiao, Xusheng; Chillarege, Ram; & Williams, Laurie A. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*. Volume 23. Number 3. June 2018. Pages 1383–1421.
https://link.springer.com/article/10.1007/s10664-017-9541-1

[Muccini 2018]
Muccini, Henry; Bosch, Jan; & van der Hoek, Andre. Collaborative Modeling in Software Engineering. *IEEE Software*. November/December 2018.

[MSR 2021]
2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). Madrid, Spain. 17–19 May 2021.
https://conf.researchr.org/home/msr-2021

[Murphy 2019]
Murphy, Gail C.; Kersten, Mik; Elves, Robert; & Bryan, Nicole. Enabling Productive Software Development by Improving Information Flow. In *Rethinking Productivity in Software Engineering*. Sadowski, Caitlin & Zimmerman, Thomas [editors]. Apress. Pages 281–292. 2019.

[Murphy 2020]
Murphy, Gail. Is Software Engineering Research Addressing Software Engineering Problems? Presented at *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. September 2020.
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9286117

[Myer 1992]
Meyer, Bertrand. Applying "Design by Contract." *Computer*. Volume 25. Number 10. October 1992. Pages 40–51.

[Nagourney 2018]
Nagourney, Adam; Sanger, David E.; & Barr, Johanna. Hawaii Panics After Alert About Incoming Missile Is Sent in Error. *The New York Times*. January 13, 2018.
https://www.nytimes.com/2018/01/13/us/hawaii-missile.html

[NASEM 2019]
National Academies of Sciences Engineering and Medicine. *Quantum Computing: Progress and Prospects*. The National Academies Press. 2019. ISBN: 978-0-309-47969-1.

[NITRD 2011]
Networking and Information Technology Research and Development Program. *Future of Software Engineering Research.* December 2011. **https://www.nitrd.gov/pubs/FOSER_report_2011.pdf**

[NITRD 2021]
Networking and Information Technology Research and Development Program. *Innovation through NITRD Coordination.* NITRD.gov. August 9, 2021 [accessed]. **https://www.nitrd.gov/**

[Novielli 2019]
Novielli, Nicole & Serebrenik, Alexander. Sentiment and Emotion in Software Engineering. *IEEE Software.* September/October 2019.

[NRC 2010]
National Research Council. *Critical Code: Software Producibility for Defense.* Washington, DC: The National Academies Press. 2010. ISBN: 978-0-309-15948-7. **https://www.nap.edu/catalog/12979/Critical-code-software-producibility-for-defense**

[NSTC 2018]
National Science and Technology Council, Subcommittee on Quantum Information Science. *National Strategic Overview for Quantum Information Science.* September 2018. **https://www.quantum.gov/wp-content/uploads/2020/10/2018_NSTC_National_Strategic_Overview_QIS.pdf**

[NSTC 2020]
National Science and Technology Council, Subcommittee on Networking & Information Technology. *Supplement to the President's FY2021 Budget.* August 14, 2020.
https://www.nitrd.gov/pubs/FY2021-NITRD-Supplement.pdf

[Office of the President 2017]
Office of the President. *National Security Strategy of the United States of America.* December 2017. **https://www.hsdl.org/?view&did=806478**

[Oliveira 2018]
Oliveira, V.P.L.; Souza, E.F.d.; Goues, C.L.; et al. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering.* Volume 23. Number 5. October 2018. Pages 2980–3006.

[Ozkaya 2020]
Ozkaya, I. What Is Really Different in Engineering AI-Enabled Systems? *IEEE Software.* Volume 37. Number 4. July–August 2020. Pages 3–6. **https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9121629**

[Ozkaya 2021]
Ozkaya, Ipek. A Watershed Moment for Search-Based Software Engineering. *IEEE Software.* Volume 38. Number 4. 2021. Pages 3–6.

[Pasztor 2021]
Pasztor, Andy & Tangel, Andrew. Boeing's Other Big Problem: Fixing Its Space Program. *The Wall Street Journal.* January 16, 2021. **https://www.wsj.com/articles/boeings-other-big-problem-fixing-its-space-program-11610773201**

[Patel 2018]
Patel, N. & Patel, K. *Java 9 Dependency Injection: Write Loosely Coupled Code with Spring 5 and Guice.* Packt Publishing Ltd., Apr 26, 2018. **https://www.packtpub.com/product/java-9-dependency-injection/9781788296250**.

[Patel 2021]
Patel, Prachi. Engineering Bias Out of AI. *IEEE Spectrum.* May 2021.

[Pham 2021]
Pham, Thanh. Analyzing The Software Engineer Shortage. *Forbes.* Apr 13, 2021. **https://www.forbes.com/sites/forbestechcouncil/2021/04/13/analyzing-the-software-engineer-shortage/?sh=1a92fd4e321c**

[Pons 2019]
Pons, Lena & Ozkaya, Ipek. *Priority Quality Attributes for Engineering AI-enabled Systems.* Presented at AAAI FSS-19: Artificial Intelligence in Government and Public Sector. November 7–8, 2019. **https://arxiv.org/pdf/1911.02912.pdf**

[Preskill 2018]
Preskill, J. Quantum Computing in the NISQ Era and Beyond. *Quantum.* Volume 2. 2018. Page 79. **https://doi.org/10.22331/q-2018-08-06-79**

[Qiskit 2019]
Qiskit. Qiskit: An Open-source Framework for Quantum Computing. *qiskit.org.* August 19, 2021 [accessed]. **https://qiskit.org/**

[Rahman 2017]
Rahman, Akond; Partho, Asif; Meder, David; & Williams, Laurie A. Pages 20–26. In *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering* (RCoSE '17). Buenos Aires, Argentina. May 2017. **https://dl.acm.org/doi/pdf/10.5555/3105398.3105404**

[Rahman 2019]
Rahman, M.S.; Rivera, E.; Khomh, F.; Guéhéneuc, Y.; & Lehnert, B. *Machine Learning Software Engineering in Practice: An Industrial Case Study.* 2019. ArXiv, abs/1906.07154.

[Ramakrishna 2021]
Ramakrishna, Sudhakar. New Findings From Our Investigation of SUNBURST. *Thwack Solarwinds IT Community.* January 11, 2021. **https://thwack.solarwinds.com/resources/b/community-announcements/posts/new-findings-from-our-investigation-of-sunburst**

[Rauch 2018]
Rauch, J.; Gonzalez, M.; & Shields, J. A. The Constitution of Knowledge. *National Affairs.* Fall 2018. **https://www.nationalaffairs.com/publications/detail/the-constitution-of-knowledge**

[Reinertson 2019]
Reinertson, Donald. *The Principles of Product Development Flow: Second Generation Lean Product Development.* Celeritas Publishing. 2019. ISBN-13: 978-1935401001.

[Rhee 2020]
Rhee, Joseph; Wagschal, Gerry; & Jung Jinsol. How Boeing 737 MAX's flawed flight control system led to 2 crashes that killed 346. *ABC News.* November 27, 2020. **https://abcnews.go.com/US/boeing-737-maxs-flawed-flight-control-system-led/story?id=74321424**

[Richards 2014]
Richards, Lisa. How Software Changed the World. *MAPCON*. March 31, 2014. **https://www.mapcon.com/us-en/how-software-changed-the-world**

[Richards 2019]
Richards, Raymond. Automated Rapid Certification Of Software (ARCOS). *Defense Advanced Research Projects Agency.* May 7, 2021 [accessed]. **https://www.darpa.mil/program/automated-rapid-certification-of-software**

[Roulette 2020]
Roulette, Joey. Boeing's botched Starliner test flirted with 'catastrophic' failure: NASA panel. *Reuters.* February 6, 2020. **https://www.reuters.com/article/us-space-exploration-boeing/boeings-botched-starliner-test-flirted-with-catastrophic-failure-nasa-panel-idUSKBN20106A**

[Ruchkin 2014]
Ruchkin, Ivan, et al. Contract-Based Integration of Cyber-Physical Analyses. In *2014 Proceedings of the International Conference on Embedded Software, EMSOFT.* **https://ieeexplore.ieee.org/document/**

[Sadowski 2018]
Sadowski, Caitlin; Söderberg, Emma; Church, Luke; Sipko, Michal; & Bacchelli, Alberto. Pages 181-190. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18).* Gothenburg, Sweden. May 27–June 3, 2018. **https://dl.acm.org/doi/pdf/10.1145/3183519.3183525**

[Santhanam 2019]
Santhanam, P.; Farchi, Eitan; & Pankratius, Victor. *Engineering Reliable Deep Learning Systems.* Presented at *The AAAI Fall Symposium Series on AI in Government & Public Sector.* Washington D.C. November 7-9, 2019. **https://arxiv.org/ftp/arxiv/papers/1910/1910.12582.pdf**

[Satter 2021]
Satter, Raphael. Up to 1,500 businesses affected by ransomware attack, U.S. firm's CEO says. *Reuters.* July 6, 2021. **https://www.reuters.com/technology/hackers-demand-70-million-liberate-data-held-by-companies-hit-mass-cyberattack-2021-07-05/**

[Schmidt 2006]
Schmidt, D. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer 39:2*. February 2006. Pages 25–31. **https://doi.org/10.1109/MC.2006.58**

[Sculley 2015]
Sculley, D. et al. Hidden Technical Debt in Machine Learning Systems. Pages 2503–2511. In *Proceedings of the 28th International Conference on Neural Information Processing Systems—Volume 2.* Montreal, Quebec, Canada. December 2015. **https://papers.nips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf**

[Seacord 2005]
Seacord, R. *Secure Coding in C and C++*. Addison-Wesley Professional. 2005.

[SEI 2019]
Software Engineering Institute. Architecture Analysis and Design Language. *Software Engineering Institute Website*. June 9, 2021 [accessed]. **https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=191439&customel_datapageid_4050=191439**

[SEI 2020]
Software Engineering Institute. Architecting the Future of Software Engineering. *2020 SEI Year in Review*. Pittsburgh, PA: Software Engineering Institute. Carnegie Mellon University. 2021. **http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=736258**

[SEI 2021]
Software Engineering Institute. Artificial Intelligence Engineering. *Software Engineering Institute Website*. June 9, 2021 [accessed]. **https://www.sei.cmu.edu/our-work/artificial-intelligence-engineering/index.cfm**

[Schatsky 2020]
Schatsky, David & Bumb, Sourabh. AI is Helping to Make Better Software. *Deloitte Insights*. January 2020. **https://www2.deloitte.com/content/dam/insights/us/articles/6342_S4S-AI-in-software/DI_S4S%20AI%20in%20software.pdf**

[Shaw 2016]
Shaw, Mary. *Progress Toward an Engineering Discipline of Software*. Presented at The 38th International Conference on Software Engineering. May 2016. **https://2016.icse.cs.txstate.edu/static//downloads/docs-and-slides/mary-shaw-keynote-slides.pdf**

[Shull 2016]
Shull, Forrest; Carleton, Anita; Carriere, Jeromy; Prikladnicki, Rafael; & Zhang, Dongmei. The Future of Software Engineering. *IEEE Software*. January/February 2016.

[Smith 2020]
Smith, Carol. Designing Trustworthy AI for Human-Machine Teaming. SEI Blog. March 9, 2020. **https://insights.sei.cmu.edu/blog/designing-trustworthy-ai-for-human-machine-teaming/**

[Svore 2018]
Svore, K. et al. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. Pages 1–10. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. Vienna, Austria. February 2018. **https://dl.acm.org/doi/pdf/10.1145/3183895.3183901**

[Swimmer 2019]
Simmer, Morton & Vosseler, Rainer. Everything is Software: The Consequences of Software Permeating Our World. *Trend Mirco*. July 31, 2019. **https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/everything-is-software**

[Thomas 1928]
Thomas, W.I. & Thomas, D.S. *The Child in America: Behavior Problems and Programs.* Knopf. 1928.

[Trujilo 2020]
Trujilo, Leonardo; Villanueava, Oar; & Hernandez, Daniel. A Novel Approach for Search-Based Program Repair. *IEEE Software.* July/August 2021.

[Tunggal 2021]
Tunggal, Abi Tyas. The 57 Biggest Data Breaches (Updated for 2021). *UpGuard Blog.* July 6, 2021. **https://www.upguard.com/blog/biggest-data-breaches**

[Turton 2020]
Turton, William. Hackers used a little-known IT vendor to attack U.S. agencies. *Fortune.* December 15, 2020. **https://fortune.com/2020/12/15/solarwinds-hackers-u-s-agencies/**

[Twetman 2021]
Twetman, H.; Paramonova, M.; & Hanley, M. *Social Media Monitoring: A Primer.* NATO Strategic Communications Centre of Excellence. 2021. ISBN: 978-9934-564-91-8. **https://stratcomcoe.org/cuploads/pfiles/social_media_monitoring_a_primer_12-02-2020.pdf**

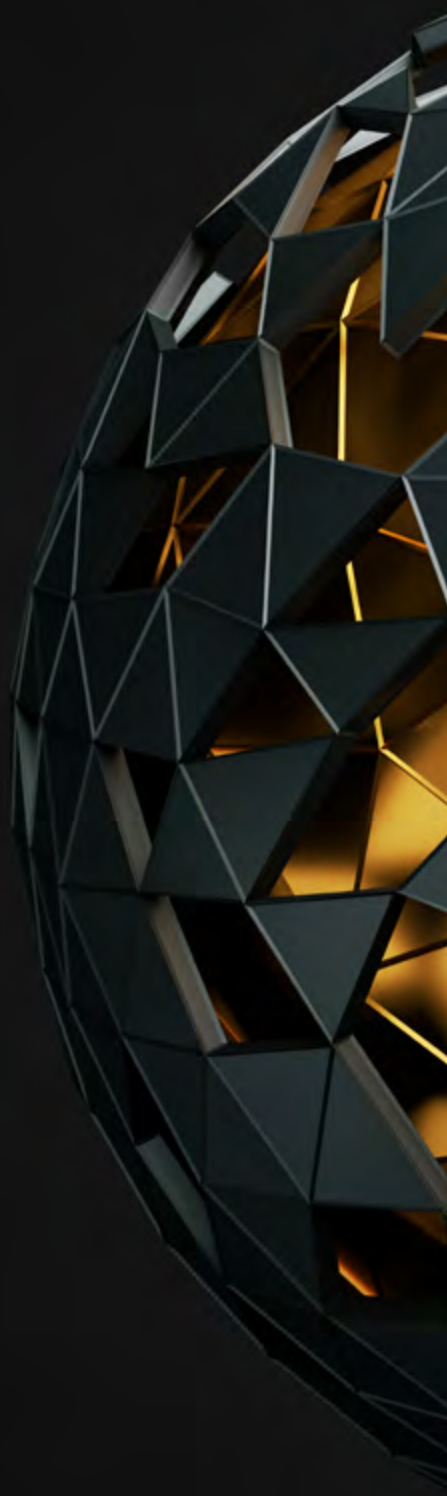
[van Genuchten 2019]
van Genuchten, Michiel & Hatton, Les. Ten Years of "Impact" Columns—The Good, the Bad, and the Ugly. *IEEE Software.* Volume 36. Number 6. 2019. Pages 57–60.

[Voelter 2013]
Voelter, M. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. *dslbook.org.* 2013. **http://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf**

[Waltzman 2017]
Waltzman, Rand. The Weaponization of Information: The Need for Cognitive Security. *RAND Corporation.* 2017. **https://www.rand.org/pubs/testimonies/CT473.html**

[Weyuker 2021]
Weyuker, Elaine. *The View from 40 years in the Research Trenches: From Academia to Industry and Back Again.* Presented at 43rd International Conference on Software Engineering. May 2021. **https://conf.researchr.org/details/icse-2021/icse-2021-keynotes/4/Elaine-Weyuker-s-Keynote-The-View-From-40-Years-in-the-Research-Trenches-From-Aca**

[Wing 2021]
Wing, Jeannette. *Data for Good: Ensuring the Responsible Use of Data to Benefit Society.* Presented at 43rd International Conference on Software Engineering. May 2021. **https://conf.researchr.org/details/icse-2021/icse-2021-keynotes/5/Jeannette-Wing-s-Keynote-Data-for-Good-Ensuring-the-Responsible-Use-of-Data-to-Ben**



About Us

The Software Engineering Institute (SEI) at Carnegie Mellon University is a Federally Funded Research and Development Center (FFRDC)—a nonprofit, public–private partnership that conducts research for the United States government. One of only 10 FFRDCs sponsored by the U.S. Department of Defense (DoD), the SEI conducts R&D in software engineering, systems engineering, cybersecurity, and many other areas of computing, working to introduce private-sector innovations into government.

As the only FFRDC sponsored by the DoD that is also authorized to work with organizations outside of the DoD, the SEI is unique. We work with partners throughout the U.S. government, the private sector, and academia. These partnerships enable us to take innovations from concept to practice, closing the gap between research and use.

Contact Us

Carnegie Mellon University
Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

412.268.5800 | 888.201.4479
sei.cmu.edu | info@sei.cmu.edu