Requirements Are All You Need: The Final Frontier for End-User Software Engineering

DIANA ROBINSON, Computer Science and Technology, Cambridge University, Cambridge, United Kingdom of Great Britain and Northern Ireland
CHRISTIAN CABRERA, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom of Great Britain and Northern Ireland
ANDREW D. GORDON, Cogna, London, United Kingdom of Great Britain and Northern Ireland and The University of Edinburgh School of Informatics, Edinburgh, United Kingdom of Great Britain and Northern Ireland
NEIL D. LAWRENCE, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom of Great Britain and Northern Ireland
LARS MENNEN, Cogna, London, United Kingdom of Great Britain and Northern Ireland

What if end-users could own the software development lifecycle from conception to deployment using only requirements expressed in language, images, video or audio? We explore this idea, building on the capabilities that Generative AI brings to software generation and maintenance techniques. How could designing software in this way better serve end-users? What are the implications of this process for the future of end-user software engineering and the software development lifecycle? We discuss the research needed to bridge the gap between where we are today and these imagined systems of the future.

CCS Concepts: • **Software and its engineering** → **Designing software**; *Automatic programming*; **Software evolution**;

Additional Key Words and Phrases: End-User Software Engineering, End-User Programming, Large Language Models

ACM Reference format:
Diana Robinson, Christian Cabrera, Andrew D. Gordon, Neil D. Lawrence, and Lars Mennen. 2025. Requirements Are All You Need: The Final Frontier for End-User Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 141 (May 2025), 22 pages.
https://doi.org/10.1145/3708524

Authors' Contact Information: Diana Robinson (corresponding author), Computer Science and Technology, Cambridge University, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: dmpr3@cam.ac.uk; Christian Cabrera, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: chc79@cam.ac.uk; Andrew D. Gordon, Cogna, London, United Kingdom of Great Britain and Northern Ireland and The University of Edinburgh School of Informatics, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: adg@cogna.co; Neil D. Lawrence, Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: ndl21@cam.ac.uk; Lars Mennen, Cogna, London, United Kingdom of Great Britain and Northern Ireland; e-mail; lars@cogna.co.

This work is licensed under Creative Commons Attribution International 4.0.

© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/5-ART141
https://doi.org/10.1145/3708524

1 Introduction

What if end-users could own the whole software development lifecycle from conception through deployment using only *natural requirements*, that is, a mix of natural language, pictures (such as sketches of user interfaces), audio or even a video demonstration?

Though this may sound futuristic, it is not actually a new vision but an idea that has captured the imagination of computer scientists throughout history, including Grace Hopper and Jim Gray.

In the 1950s, compiler pioneer Grace Hopper was head of *automatic programming* for the first commercial computer, the UNIVAC of Remington Rand [51]. She decided that 'data processors ought to be able to write their programs in English, and the computers would translate them into machine code' [33]. The manual of her language FLOW-MATIC describes its 'instruction code' as being an 'English language description of application requirements' [79]. FLOW-MATIC had a major influence on COBOL [81], which was perhaps the most widely used programming language of the 1970s and remains in use today.

By the turn of the century, Jim Gray, in his 1998 Turing Award lecture, described automatic programming as the 'Holy Grail of programming languages and systems for the last 45 years'. He included automatic programming as one of a set of fundamental research questions for information technology [37]. His definition goes far beyond the highly constrained English language requirements allowed by FLOW-MATIC and its widely used successor.

Automatic Programming (from Gray's Turing Award Lecture [37]): An Imitation Game for a Programming Staff

Devise a specification language or UI

(1) That is easy for people to express designs (1,000 × easier),
(2) That computers can compile, and
(3) That can describe all applications (is complete).

The system should reason about application, asking questions about exception cases and incomplete specification. But it should not be onerous to use.

Around that time, researchers identified and studied *end-user programmers*, professionals in some area other than software engineering who nonetheless develop skills in programming to get work done in their domain of expertise [6, 69], such as spreadsheet users. The field of **End-User Software Engineering (EUSE)** [11, 52] aims to help end-user programmers develop programs more systematically and with high quality. A core challenge is that end-users who write and develop programs tend to have no training in nor any particular interest in software engineering.

Coming to the present day, **Large Language Models (LLMs)** such as OpenAI's GPT-4 [71] have demonstrated impressive advances in abstracting the world in a way that renders a computer's planning and reasoning capabilities more accessible to humans, alleviating the usual requirements for strict logic specification. This capability has been helpful in code generation, which can support several areas of software engineering [2, 15, 30, 72]. GitHub's Copilot, launched in mid-2021, empowers professional developers by using an LLM to generate snippets of code from textual prompts, to explain code, to diagnose errors and to repair them [106].

Consider *whole apps*, that is, pieces of deployable code such as a native desktop or phone app, or a modern Web site. Since the launch of GPT-4 in 2023, the generation of whole apps from simple natural language requirements has become an active research area. Although today's LLMs struggle to generate whole apps reliably in a single pass, several research projects aim to synthesise whole

apps by chaining calls to LLMs repeatedly and employing LLMs to play different roles such as requirements engineer, software architect, software engineer or software tester (see for example [44, 98, 103]). Commercial products include Cogna,[1] and Devin,[2] while there are several open source projects such as AutoGPT,[3] GPT-Engineer,[4] GPT Pilot,[5] and Devika.[6]

The following states our key hypothesis. The title of the paper is a corollary.

Natural Requirements Hypothesis:

Given Generative AI, we can solve the research challenge of automatic programming by taking the specification language to be *natural requirements*, that is, natural language plus drawings, demonstrations, and examples.
A corollary: requirements are all you need for automatic programming.

Gray [37] described automatic programming as a Turing-style imitation game for a programming staff [99]: replace a software development team with a computer that is 'better and requires no more time than dealing with a typical human staff' [37]. By definition, automatic programming solves the problems of end-user programmers since they can simply work with the automatic programmer to build apps to meet their needs.

Given our hypothesis, our vision for software engineering by 2030 is that end-users will build and deploy whole apps just from natural requirements. We call this *requirements-driven EUSE*. New human–AI interfaces will help elicit requirements from the end-user and assist them in attending to quality and deployment issues for LLM-generated apps. These interfaces will transform end-users' ability to build software, despite their indifference to engineering. Today, an end-user would need a professional engineering team to build a custom app just from requirements. By 2030, relying instead on AI-powered automatic programming, they can build dramatically more custom apps than now. We expect the area to begin with relatively simple apps confined to a lower risk set of domains and gradually to scale up, using expert human oversight to complement AI, especially if defects would be consequential.

In this article, we explore the limits and possibilities of this idea of not only expressing requirements but using them to direct software testing and adaption to deployment changes. We discuss the recent technical breakthroughs that have paved the way and the future work needed to realise this vision in a research agenda encompassing requirements elicitation, testing and maintenance and deployment.

Our title reflects our optimism for the future of requirements-driven EUSE. Still, there are two recurring challenges that will need care and attention:

—Our human users are not intrinsically interested in expressing requirements, even if natural language may be an easier medium for them than conventional programming languages.
—Our machines that interpret natural language are not intrinsically reliable, even if they often are startlingly good at generating code, and much better than previous technologies.

In Section 2, we outline a case study and walk through the three tenets of our research agenda in relation to it; in Section 3, we propose a software development lifecycle for requirements-driven

[1] *Cogna: Hyper-customised software defined by you, delivered by AI*, February 2024.
[2] *Cognition AI: Introducing Devin, the first AI software engineer*, March 2024.
[3] *AutoGPT: build and use AI agents*, April 2023.
[4] *GPT-Engineer*, June 2023.
[5] *GPT Pilot*, December 2023.
[6] *Devika is an Agentic AI Software Engineer*, March 2024.

EUSE and then explain each aspect of the research agenda in terms of challenges and research directions, and in Section 4, we discuss risks and mitigation strategies. Section 5 concludes the article.

2 Case Study

We start with a case study to illustrate the tenets of our research agenda which we detail in the next section. Here are the requirements for an app to track the currency requirements[7] of an airline's roster of pilots.

Requirements for a Pilot Currency App:

Here is a minimum list of the US FAA currency requirements for pilots:

—Pilots need to complete a flight review every 24 months, to demonstrate flying ability.
—They must perform at least three takeoffs and landings within the last 90 days to carry passengers.

Build app to:

(1) Allow pilots to log their completed flight reviews.
(2) Access flight logs showing takeoffs and landings.
(3) Show dashboard of pilots.
(4) Warn of any pilots whose currency requirements have expired.
(5) Book pilots in for flight reviews whose requirements will expire in the next month.

2.1 Requirements Elicitation

Though the above requirements seem straightforward, clarifying user intention is often an iterative process that can only be worked out through prototyping and feedback. In our example, if some pilots have retired and an initial version of the app returned hundreds of needless alerts an iteration would prompt the user to clarify *active* pilots in the requirements. There are also requirements that users might not articulate because they are experts in a domain and assume them. Suppose the user had specified the above five requirements for the pilot currency check app, but had not stated that the category and class of aircraft had to match that of the one the pilot next wanted to fly. Anti-requirements, in other words, things we do not want to happen sometimes need to be specified as well. In the example, they may include access privileges to the system, such as pilots not having write access to flight logs and takeoffs.

2.2 Testing

Suppose natural requirements from end-users are the only component dictating the software before it gets built. In that case, there is a broader scope for errors to be introduced into the resulting system. Testing systems in context is often the only way to ensure completeness of the requirements as even experienced software engineers would be unable to fully predict all downstream consequences of the deployed system. To start, this might be done through parallel running the existing process alongside the app and comparing whether they have achieved the same results. Where there are differences, the user can decompose these into requirements for the system to generate tests. In the aviation example, perhaps a list of pilots with expired currency requirements is returned which the user can compare to a known list. Then they can drill into the details of the cases to see where

[7] *What is Pilot Recurrency Training?*, 2024.

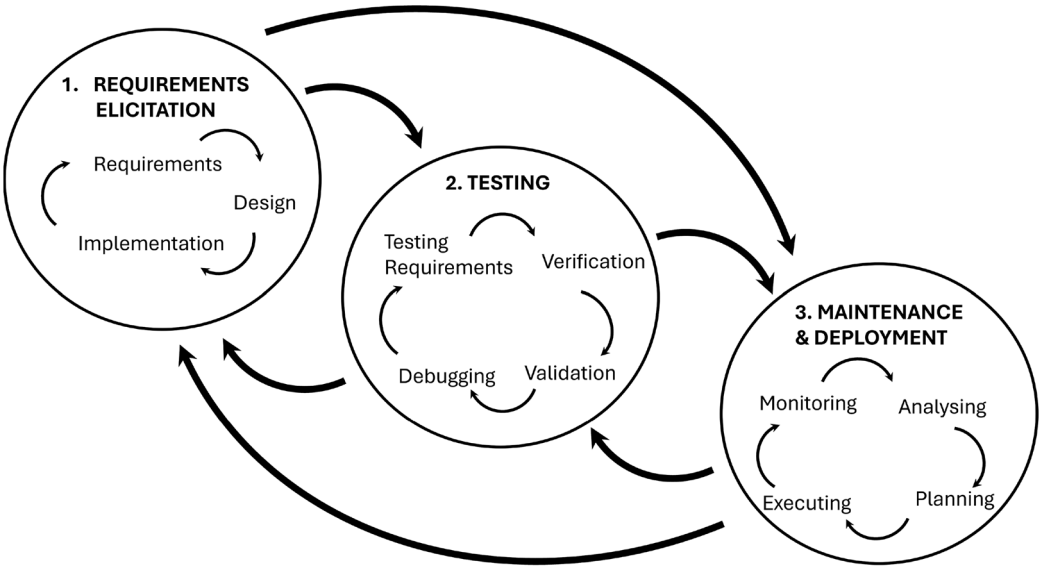

Fig. 1. Diagram of the requirements-driven EUSE lifecycle.

the app has misinterpreted and define requirements for the software to generate unit tests and flag where the original requirements need to be changed.

2.3 Maintenance and Deployment

Dynamic application environments can generate software failures and sub-optimal performance. Applications must use autonomous mechanisms to self-maintain and self-heal. These mechanisms must also enable transparent maintenance where end-users understand the failures and their causes but need not intervene. An example of a dynamic requirement in the aviation case is when the **Federal Aviation Administration (FAA)** changes its regulations. The FAA Web site could be an input to the requirements to be monitored for updates and for the current app to be updated and notify the user. Autonomous mechanisms for fixing apps that might not need to be continuously communicated to the user can include dependency management or autoscaling. These kinds of changes are out of the users' expertise and would require autonomous fixes which need to be bounded in the amount of change they could make to the app without needing to bring the user back into the loop.

3 Research Agenda

Burnett and Scaffidi [13] lay out five steps in the EUSE lifecycle: requirements and design, implementation, verification and validation, debugging and code reuse. Figure 1 shows our revised version of this lifecycle composed of three stages. We combine requirements and design with implementation in an iterative loop because, with automation of implementation from requirements, the design and elicitation phase can include rapid prototyping to clarify user intentions. We group verification and validation with debugging as we envision a process of requirements-driven testing that is interpretable to users with automated tests designed from higher-level specifications. Finally, we propose a new final step of maintenance and deployment that systems will use to self-adapt. This step includes tasks for monitoring and analysing software and environment changes and planning and executing autonomous maintenance and deployment decisions.

The rest of this section lays out a research agenda for each stage of the lifecycle, discussing the relevant challenges and state of the art, followed by a series of suggestions for targeted research investigations.

3.1 Requirements Elicitation: Assist End-Users in Expressing Their Requirements

Requirements elicitation has traditionally involved intermediaries between the user and the software. These might be people, such as UX researchers and software engineers, who translate the requirements of the user, or EUSE tools, which enable users to take actions in a particular programming language or environment. Both give rise to constraints around how and what the user can express. In the new paradigm we are suggesting, designing whole apps from requirements allows the user to communicate their requirements directly. By accepting natural requirements as the specification language for automatic programming, we intend to empower the end-user to express the full nuance and complexity of their goals. Still, the new intermediary in the form of the LLM interface opens up new opportunities and challenges, and the user still needs to learn how to express requirements in a way that can be interpreted successfully. Our vision exemplifies Sarkar's hypothesis that Generative AI enables a 'radical widening in scope and capability of end-user programming' [82].

Two challenges arise from this increased freedom to express requirements directly into software. The first is how to provide the required structure for users to succeed. Uncovering and defining requirements is hard to automate, as users and even software engineers often end up specifying requirements that are impractical, unnecessarily complex or infeasible. The challenge is to guide users to express their requirements comprehensively and clearly enough to be built into software. The second challenge is helping the user to understand the possibilities and limitations of the software they can build. In other words, what areas of their workflow could they automate? Past work in EUSE by Ko et al. [54] called this a selection barrier: 'finding what programming interfaces are available and which can be used to achieve a particular behaviour'. Program synthesis alleviates the first part as it should match execution to requirements. Still, helping the user discover which behaviours are possible remains a challenge.

There are two strands of work in EUSE that are directly relevant to the new paradigm we are proposing: programming by example and natural language programming.

Following years of research on programming by example [25], FlashFill, released in Excel 2013 [39], enabled widespread use. FlashFill automates string processing for end-users by generalising from one or more examples. Building on this success, Generative AI shows tremendous capabilities to transform end-user programming: a demo of Google's multi-modal model, Gemini Pro [96], shows extraction of code to automate a web browsing task given just a screen recording of the user demonstrating the task.[8]

The second area of research is natural language adopted for end-user programming. For instance, a concept spreadsheet, GridBook, demonstrated data analysis via natural language within the grid itself [89]. Following the success of GitHub Copilot since 2021 in empowering developers with code autocomplete, researchers adapted code-generating LLMs to generate spreadsheet calculations [62], with eventual impact in Excel Copilot. These developments show promise but their scope is limited to spreadsheet calculations. Requirements elicitation for whole app synthesis opens up new possibilities for what end-users can build but also new challenges in an open environment beyond the spreadsheet. Alongside natural language, there is a great opportunity for end-users to draw images or make short videos to express requirements.

[8] *Paige Bailey: Mind officially blown*, 22 February 2024.

Recent research explores the possibilities of translating natural language requirements into software applications across domains from creative endeavours to safety-critical contexts. For example, Vaithilingham et al. explore new capabilities that LLMs bring to design processes by developing an imagined scenario of an LLM-powered chat-based dialogue to elicit requirements for a video game [100]. Fakih et al. [29] built a 'user-guided iterative pipeline' that integrates LLMs for programming in Industrial Control Systems. Their pipeline begins with an iterative loop between the user and the LLM to come up with a model that informs the subsequent code generation for Programmable Logic Controllers, which are used in industrial infrastructure applications. Wang et al. [101] design a process for translating high-level requirements specified in natural language into formal specifications for network configurations.

We see an opportunity to build on research and techniques from **Human Computer Interaction (HCI)** and human factors in this new paradigm. We can leverage existing strategies for understanding user goals and preferences and redefine them in light of decreased constraints around time and resources when users themselves can be directly involved in creating software. LLMs might provide an opportunity to apply HCI methods to make requirements engineering more nuanced. There is enormous potential to scale techniques that are currently time- and human-labour-intensive, such as ethnographic work and interviewing, to develop a deeper understanding of a user's context and needs. For example, a way to structure the process of requirements elicitation with users might come from semi-structured interviews in which an interview protocol is devised in advance but amended based on responses from the interviewee. LLMs could be trained in this process to guide users through expressing and exploring their preferences.

As we move from a paradigm that involves many skilled humans in the pipeline to one in which LLMs are mediating the interaction between end-users and the software produced, it is important to consider the domains in which this sort of experimentation would be responsible. We believe the level of trust we can put in systems based on the current state of the art of LLMs is low, due to risks like hallucination. Hence, the outputs of requirements-driven EUSE tools should be contained to particular domains in the way that Khlaaf suggests in her work on **Operational Design Domains (ODDs)** [49].

3.1.1 Focus on Software Applications in Shaw's Good-Enough Realm. Figure 2 shows Shaw's plot of different categories of software applications in the space given by a couple of axes: the consequences of failure versus the degree of human oversight [84]. Her 'Good-enough Realm' is the bottom left quadrant, where the consequences of failure are low and the degree of human oversight is high. In this quadrant, formal verification is not essential. The Good-enough Realm includes the categories of appointment scheduling, automatic payments, and tax return software. For these kinds of software, the cost of formal verification is too high and the risk of not doing it is acceptable.

Her 'Correctness Realm' is the top right quadrant, where the consequences of failure are high and the degree of human oversight is low. In this quadrant, formal verification is essential. The Correctness Realm includes the categories of nuclear power plant control, missile guidance and medical device control, and other examples of safety-critical applications. For these kinds of software, the cost of formal verification is justified and the risk of not doing it is unacceptable.

Kang and Shaw argue that Generative AI is best suited for software in the Good-enough Realm, where 'fitness of purpose' is a more appropriate criterion for software correctness than formal verification [48, 84]. Instead of formal specifications that are 'formal, static, and complete', fitness of purpose can be checked with respect to 'software credentials', which are 'heterogeneous, evolving, and incomplete' [48]. Simple examples of software credentials could be textual summaries of the

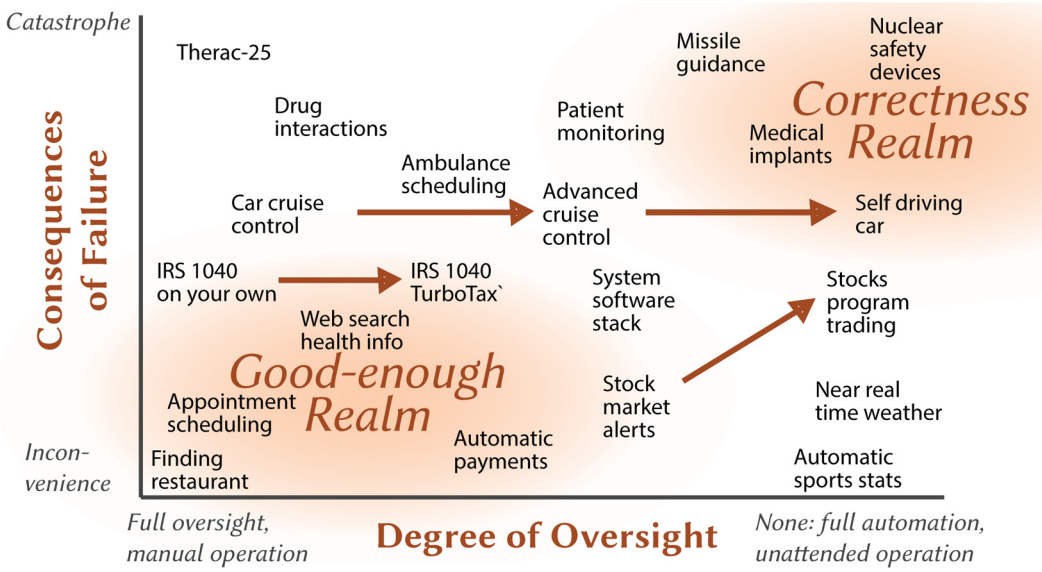

Fig. 2. Shaw's plot of software categories in two dimensions: consequences of failure versus degree of human oversight [84]. Image © Mary Shaw. Used with permission.

behaviour of a piece of code, and also of the meaning of a software test. End-users are likely better able to check textual descriptions than the actual code or test.

A challenge for requirements-driven EUSE, then, is to better understand what sorts of software credentials are appropriate to different categories of software, especially in the Good-enough Realm, and to develop effective methods of suggesting and checking these credentials. For example, a textual explanation of the behaviour of each button in a generated user interface would count as a software credential to be inspected by the end-user. Techniques from CLOVER [92] (discussed further in Section 3.2.2) could be adapted to generate and check these textual explanations.

3.1.2 Develop Automated Ways of Accessing Tacit Knowledge. Beyond getting users to concisely and fully express their requirements, a key challenge lies in finding ways of accessing tacit knowledge. This includes both tacit knowledge of the user for the task they want to build for and of the requirements engineer or UX researcher with expertise in eliciting this form of knowledge. We will look at paths towards addressing both of these research challenges.

What do we miss when we lose the aspect of observing users executing their process and how can we build new automated ways of doing this? Observations are valuable input when generating requirements, as sometimes they reveal tacit knowledge or subconscious priorities that a user would not think to articulate. Methods like ethnography rely on the human ability to empathise as well as an inherent understanding of each other and the reasons we might do something. When we automate these types of processes, this will bring both new opportunities and challenges. How can we build this into a process of direct interaction with software through natural requirements, for example leveraging programming by demonstration or multi-modal requirements elicitation?

One way of accessing this tacit knowledge is via quicker iteration with users for earlier and more granular feedback on how their goals can be built. A specific technique from HCI that lends itself well to this new process of building software from requirements is the Wizard of Oz technique [76]. This is a method of developing low-cost, quickly produced prototypes of interfaces while manipulating them in the background to mimic a completed product. We think this could be adapted

using LLMs to iterate on quickly mocked up prototypes with users. Often seeing a tangible product can unearth elements of the requirements that were assumed subconsciously but not articulated. It can also highlight anti-requirements, things that must not happen in the software but are sometimes harder to articulate.

A potential way forward in the data science space is suggested by Guo [40] using the analogy of an expert data scientist looking over one's shoulder, demonstrating things and making suggestions. Their ideas for implementing this are adaptive user interfaces for data science tasks, such as modifying the Javascript of Web sites to create new ways of downloading and visualising datasets in real time [40]. In this scenario, the LLM would be learning from the user's web browsing to understand the data science tasks and problem solving they were engaged in [40]. By enabling the LLM to learn from the process of the data science analysis, it acts as an observer which helps to uncover tacit knowledge.

The external viewpoint of an automated observer could be a useful way of understanding user intent. Still, it would need to be carefully designed for timing of suggestions [87] and to preserve user agency. We can draw upon work from human factors to design workflows for producing software directly from requirements in ways that do not distract the user from their core goals or lead to loss of productivity [87].

3.1.3 Build Requirements Datasets for LLM-Based Code Synthesis and Evaluation. Unlike human requirements engineers, LLM-based interfaces for requirements capture will be able to learn from a much wider code base for implementation options once they have requirements. Difficult challenges lie in building datasets that translate user needs into code to do the initial training or fine-tuning of the models that generate software as well as new datasets to evaluate the generated code.

Though the large code base of examples that LLMs can learn from is a benefit, it comes with new risks as well. For example, Kang and Shaw [48] fear that the creativity of human developers could atrophy if developers become over-reliant on AI-generated solutions. LLMs could further reinforce this feedback loop by suggesting standard solutions and then training on them to continually deploy, reducing the diversity of the generated software [48]. This problem is known as model collapse when an LLM generates content that 'converges to excessively uniform behaviours' [41]. Guo et al. studied this phenomenon in a range of natural language tasks, training subsequent models on data generated by past models and finding that linguistic diversity decreased [41]. Shumailov et al. found that over time models trained on LLM-generated data would skew from the underlying learning task [86]. They identified maintaining provenance of human-generated data to be an important priority [86].

To compound this problem, if non-technical users are building software, they are unlikely to imagine novel software solutions themselves. The challenge therefore is in providing feedback mechanisms in terms of user intentions that steer the models to generate novel solutions or scaffolding LLM suggestions to the users in ways that help them learn essential principles of generating code that can help them to generate ideas and come to new suggestions. There is an opportunity to open up software development to a much wider set of people when leveraging the possibility that LLMs can provide assistance to lower the barrier to entry. But how do we enable these new users to imagine new ways of building software?

Answers to these challenges could leverage work on metacognition [95], creativity [107], and education [77] with LLMs. For example, some research looks at boosting creativity within LLMs, such as Mehrotra et al. who examine whether associative thinking, which can help humans think creatively by linking seemingly unrelated concepts, is a useful prompting strategy to get LLMs to provide more creative outputs [107]. Further research is needed on fostering creativity in end-user software engineers while using LLMs, to avoid complacency and over-reliance [83, 87] and to

generate novel solutions. We need to design LLM interfaces and workflows involving LLMs in ways that lead to productive diversity in software produced and maintain a human-generated component that feeds back into the datasets.

A difficult problem will be to develop both the datasets that translate from end-user requirements to code implementation for LLMs to train on and also the benchmark datasets on which to evaluate new models or systems.

LLMs provide an opportunity to create feedback loops between system specifications and users such that users can describe what they are seeking and the system can feed back its interpretation, potentially in the form of rapid prototyping. This feedback between the user and the system can generate new datasets to train LLMs for requirements elicitation. As a way of building these datasets, conversations between UX researchers and requirements engineers with their users or clients need to be curated for initial expression of requirements and stages of refinement up until the final specification before implementation. By training on these datasets, LLMs can begin to generalise the process of refinement necessary to go from user need to final specification. Once we have this, we then need to maintain a human-generated element of the data in order to not verge into model collapse. This could mean putting higher weights on human-generated data to train the models on, similar to techniques for sampling from imbalanced datasets by oversampling the minority class.

We also need to create new benchmark datasets to measure the success of LLMs in translating from natural requirements to code. The HumanEval [22] and **Mostly Basic Programming Problems (MBPP)** [4] benchmarks both consist of a set of function specifications in natural language accompanied by a set of unit tests. We need to go further in order to assess the competence of LLM-based requirements elicitation tools. End-user software engineers with no experience in programming will not describe their programming problems in ways that align with the implementation as a programmer would. Instead, they will describe them in terms that make sense within their domain of work. We will need datasets that consist of examples from a wide variety of domains so that models can generalise about code implementations that satisfy user needs.

3.2 Testing: Generate Tests from Requirements that Are Meaningful to End-Users

How can an end-user tell whether their natural language requirements specify a whole app that meets their needs? After all, since the 1970s, critics of natural language specifications have pointed out the potential for ambiguity, even nonsense [27]. Even if there is no ambiguity, how can the end-user tell whether the delivered app actually meets their requirements? LLMs are prone to hallucinate faulty code.

The human ability to audit or evaluate the responses from Generative AI is rapidly becoming an important life skill. A workshop for internal contributors to Microsoft Copilot described tools to aid humans in this task as *co-audit tools* [36], and listed some general principles. Still, different kinds of content are evaluated differently, and we can start by considering how humans already evaluate whole apps during development.

A professional software engineering team writes a suite of tests, such as input/output examples, to answer these questions. During requirements elicitation and code development, the team can:

(1) test its understanding of potentially mistaken, ambiguous or contradictory requirements, leading to progressive repair of the requirements, and
(2) test whether the code of the app meets the requirements, leading to progressive repair of the code.

Helping end-users write tests for their programs is a longstanding challenge in EUSE. End-user programming, such as formulas in spreadsheets, has been extremely error-prone. For example,

Panko [75] found there is a high probability of error affecting the bottom lines of any substantial spreadsheet. Burnett concurs: 'Considering software quality is necessary, because there is ample evidence that the programs end-users create are filled with expensive errors'. [10] Despite the prevalence of consequential errors, it has been hard to get end users to write tests. We face the characteristic challenge for EUSE that 'the user probably has little expertise or even interest in software engineering' [10].

So, how can we help end-users follow good practice and generate a comprehensive test suite to perform the steps above: (1) to test and repair requirements and (2) to test and repair code?

3.2.1 Automatically Generate Tests for the User. Future research can build on previous findings from testing and debugging tools for spreadsheets and other forms of end-user computing. These tools make details visible [11], entice the user [80], utilise natural problem solving interactions like asking why questions [53], provide user interfaces to inspect formulas [31] and help users think about details in context [24] (perhaps via simulation or prototyping of a narrow slice of the requirements).

Ideally, we can auto-generate tests from the requirements using LLMs [30, 72]. For example, CodeT [20] improves code generation performance by generating tests automatically from natural language descriptions to effectively select appropriate code from amongst the many possibilities suggested by an LLM. Reflexion [85] achieved state of the art performance on HumanEval using a prompt chaining method that generates code, applies tests, reflects on error messages produced and then repairs the errors discovered. Still, a sign of ambiguous requirements is that the pieces of code behave differently on the tests. Another work, ClarifyGPT [67] exploits this observation by detecting whether a given requirement is ambiguous by generating tests and also multiple pieces of code for the requirement and then applies the test to check that pieces of code behave the same on the tests. If so, ClarifyGPT prompts an LLM to generate targeted clarifying questions for the users to resolve the ambiguity.

Still, when synthesised tests fail and cannot be automatically repaired, we need methods to communicate the situation to the end-user, and seek their help. Information from the test failure needs to be intelligible to the user in terms of the high-level requirements, as opposed to low-level details of the code. The end-user needs to be able to interpret the outcome of running tests, so they can help resolve the situation. Hence, when a test fails, the user can help determine whether the test is correct and the code faulty, or vice versa.

One hypothesis for future research is that if testing can be meaningfully aligned with a user's end goals, they might be more inclined to engage with it. Testing an app versus specifying its purpose in requirements involves different ways of thinking. Trying to both lay the groundwork for an app as well as question it within the same set of requirements would likely fail. Thus, we propose a separate requirements elicitation process for debugging where the end-user can suggest a set of requirements for testing in the form of how their software should behave, what must happen and what must not happen, and then automatic unit tests are designed from this through program synthesis. In this way, testing might be more meaningful to users as they would be able to relate it to their goals and system understanding.

3.2.2 Generate Formal Specifications and Explain Them to End-Users. Formal verification of code versus a formal specification can be seen as a form of testing. Recent work [66] using the verified programming language Dafny [60] considers how to generate functions equipped with formal pre-conditions and post-conditions from natural language descriptions, as well as loop invariants to enable verification. Results are very encouraging: the method generates human-validated pre-conditions and post-conditions with verified code bodies for 58% of a chosen subset of the MBPP dataset. CLOVER [92], a parallel line of work, also generates verified Dafny code from

requirements. A core concept is to maintain consistency between three representations of intended functionality: requirements (docstrings), code and formal annotations (including pre-conditions and post-conditions). CLOVER stands for Closed-Loop Verifiable Code Generation. It provides bidirectional mappings between the three representations and applies the maps iteratively to achieve consistency.

Meanwhile, there is impressive progress on using AI to develop new mathematics, such as AlphaProof, which achieved a silver medal standard in Maths Olympiad tests.[9] AlphaProof generates proofs in the Lean theorem prover [26]. We can expect that improvements in automated formal verification can be applied to LLM-generated code.

On this basis, we hypothesise that even for apps in the good-enough quadrant, formal specifications cheaply generated by these automated methods will be useful because they can be explained more easily to the end-user than the code itself. A formal specification amounts to a succinct summary of the functional behaviour of a piece of code. Hence, it may be easier or more effective to generate a textual explanation of the behaviour of a piece of code via its formal specification than to generate the text directly from the code.

3.2.3 Apply Language-Based Security to LLM-Generated Code. We should consider LLM-generated code as adversarial and employ ideas from language-based security to limit the potential harm, and the need for testing.

LLM-generated code is intrinsically untrustworthy. If the underlying training data is poisoned, it is possible that the LLM can generate malicious code. For example, Hubinger et al. [45] demonstrate the possibility of an LLM that generates code that is malicious, but only when certain conditions are met, and otherwise generates benign code. Still, the fundamental reason to distrust code from an LLM is the probabilistic nature of token generation, which may hallucinate faulty code.

In computer security, the *trusted computing base* is a 'small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security' [56]. Code within the trusted computing base needs to be carefully audited and verified by humans to avoid security vulnerabilities. A basic principle of computer security is to focus human attention on evaluating correctness of the trusted computing base and to pay less attention to untrusted code, since we can apply security mechanisms to prevent harm by the untrusted code.

For example, in the early days of the web, Java applets were confined to a sandbox, a limited environment where applets could do computation, and display visuals, but could not access the local file system or the network, and could not interact with other applets, to prevent malicious code from harming the user's computer [35]. More recently, sandboxing has been used to protect from potentially malicious or faulty code generated by LLMs. For example, the Code Interpreter plugin for ChatGPT[10] generates Python code for tasks such as solving logic problems or analysing uploaded spreadsheets and runs the untrusted Python in a temporary sandbox with no web access.

To go beyond sandboxing, Java introduced the idea of permission-based access control [35]. In the simplest form, an untrusted app requests a set of permissions, such as permissions to access local files or databases, and so on, and the user can grant or deny these permissions. Permission-based access control is the basis of the security model of modern mobile OS, such as Android [5]. For example, a malicious torch app might request permission to switch on the camera flashlight, but also permission to access the user's contacts. In theory, the user will be suspicious of the torch app requesting access to the user's contacts and will deny that permission. In practice, relying on the end-user to make security decisions is ineffective. The authors of a critical review of Android

[9] *New York Times: Move Over, Mathematicians, Here Comes AlphaProof*, 25 July 2024.
[10] *OpenAI: ChatGPT plugins*, 23 March 2023.

security conclude that: 'Permission dialogs have issues similar to warning messages: They fail to lead to the desired effect, as users tend to click through them, misunderstand their purpose, and hence do not benefit from them' [1].

These Java mechanisms are examples of *language-based security*, the idea that the programming language itself can enforce security properties. There is a rich literature on language-based security and we believe there are many opportunities to apply techniques from this research to LLM-generated code. Some of these will be technical analyses of information flow in apps, and some of these will be techniques to help human end-users to understand and act on security properties of the generated code. In particular, the idea of a trusted computing base helps limit the amount of code that needs to be rigorously tested. Static analysis can reduce the amount of LLM-generated code in the trusted computing base and hence reduce the need for testing.

Bastion [70] is an ongoing research project to limit the potential harms of agentic AI by using language-based security techniques, including capability access constraints, information flow constraints and causal constraints enforced via the F* type system [93].

3.3 Maintenance and Deployment: Respond to Dynamic Requirements and Environments

The deployment and maintenance of applications are the next steps we envision in the requirements-driven EUSE lifecycle. The deployment phase installs the application in the end-user's environment, either cloud or local machine. Maintenance tasks are responsible for monitoring and optimising the application once deployed. These complex decision-making processes involve different variables at the requirements, software architecture, infrastructure and environment levels. When considering EUSE, Lieberman advocates for automatic application deployment and maintenance [61]. Dynamic requirements and environments challenge such automation because applications must guarantee optimal functioning despite changing goals, variable data, unexpected failures, and security threats, among other variables that emerge from the real world [16].

The state of the art of research on autonomous deployment relies on continuous delivery, DevOps and MLOps tools [74]. These are successful engineering solutions for automatically executing deployment pipelines. However, creating such pipelines is mostly a manual task in which experts define how and where to install each piece of software based on end-user technical and budget constraints. Autonomous maintenance requires systems that autonomously adapt to end-users' requirements and environment changes. The field of *autonomic computing* [55] aims to build systems that respond to variable environments without human intervention. It proposes to equip systems with sensors for environment monitoring, decision functions to reason about perceived changes and actors to modify systems' structure, behaviour and environment variables. These elements constitute feedback loops that accumulate knowledge from systems' internal and external interactions. Autonomous agents use this knowledge to make adaptive decisions [14, 102]. However, current autonomous agents present limitations to modern software systems trends (e.g., data-driven systems). They are based on processes that optimise variables that are easy to quantify (e.g., applications response time) [57]. Modern systems have demanding high-level requirements such as accountability, trustworthiness, transparency and fairness [23, 88]. The optimisation processes rely on black-box models, the decisions of which are hard to explain in complex systems [18, 19]. This lack of explainability impacts the end-users' understanding or interpretation of systems.

The limitations outlined above challenge the sustainability of software systems in different dimensions. Individual sustainability is impacted by sophisticated software systems that are difficult to use (e.g., manual pipelines) and end-users do not fully understand (e.g., lack of interpretability). Software misalignment with high-level requirements and goals negatively impacts the sustainability of systems from a social perspective as the systems' utility is limited. Dynamic and complex

environments challenge software sustainability from an economic, technical and environmental perspective as these environments demand considerable resources to develop complex systems. Addressing these limitations to offer sustainable software systems that self-maintain while keeping under end-users' control is an important research area that requires our immediate focus [15].

We envisage the following research challenges from the EUSE perspective:

3.3.1 Automate Software Deployment Plans. Integrating capabilities from LLMs into the deployment and maintenance of synthesised applications has the potential to support the automatic creation of deployment plans and autonomous responses from applications to changing requirements, variable data, failures, and unexpected behaviour. The EUSE community must address challenging research questions before this potential is realised. These challenges emerge from the nature of LLMs, which are non-deterministic, prone to hallucinations, and operate as black boxes [30, 72].

Deployment plans must be consistent throughout the application lifecycle, but LLMs can generate different plans for the same application because of their non-deterministic responses. The following research questions arise: To what extent do non-deterministic responses impact the application deployment plan? How do we validate deployment plans before execution? How should we handle stochasticity when executing deployment plans? And how can current tools (e.g., continuous delivery, DevOps, MLOps) integrate deployment plans generated by LLMs?

A research direction to address the above challenges is to develop a domain-specific LLM specialising in software engineering [15, 47]. The software engineering community has historically created a rich body of knowledge based on empirical evidence available now on the Internet. Using these data to fine-tune an LLM can enable the generation of deployment plans according to software architectural patterns and practices defined by experts. Such a specialised LLM would create more accurate deployment plans, but hallucinations can still appear. We envision using design artefacts as knowledge to drive the responses of the LLM. For example, end-user technological constraints can be expressed as factual knowledge that LLMs responses must respect. Prompt engineering or Retrieval-Augmented Generation [21] are promising research directions [15] to include external data and make LLMs' outputs more consistent.

3.3.2 Maintain Software Components Synthesised by LLMs. LLM hallucinations can cause application misbehaviour by injecting errors into software components, deployment plans and maintenance decisions. There is already progress on automatically resolving a benchmark set of GitHub issues [46], including maintenance tasks. However, LLMs' non-deterministic behaviour challenges deploying and maintaining AI-generated software systems. The holistic deployment and maintenance of applications still have the following open questions: How can we handle hallucinations to generate software components and deployment and maintenance plans? How can users identify faulty or outdated components generated by LLMs? How can designers identify causal relationships between requirements and components to enable preventive or corrective maintenance? How can we support alignment analysis between users' requirements and applications using causal relationships?

Using non-deterministic AI models (e.g., LLMs) to generate software components impacts its safe and reliable deployment, especially in safety-critical contexts that require performance guarantees. Communities and end-users must know this new reality and consider the risks of automatically developing applications relying on LLMs. Addressing these risks requires a shift towards development methodologies prioritising planning phases (e.g., requirements definition) against production phases. However, current organisations profess an agile culture where working software is the measure of progress [90]. An initial step towards these shifts is developing new software development methodologies based on current methods for designing critical data-driven systems [58, 59].

In addition, we need dynamic design architectures that represent the applications' state through time and enable emulations of possible software behaviour and maintenance tasks' impact.

A promising research direction is the work on **Data-Oriented Architectures (DOAs)** [16], which can support causal analysis of systems data. DOAs advocate for exposing data using paradigms like dataflow computing, which provides a computational graph of the system by design. We can then treat this computational graph as a complete graphical causal model where practitioners can apply causal inference techniques to dataflow systems [73].

3.3.3 Develop Novel Interfaces for Improving Systems Interpretability. Deployment and maintenance processes must be automated, while remaining transparent and interpretable to end-users. LLMs offer new interfaces between applications and end-users. However, questions regarding the explainability of AI-based applications remain open: How can we explain deployment and maintenance decisions made by LLMs to end-users? Can the interaction between end-users, LLMs and other entities in the synthesis process (e.g., external data sources) improve such explainability?

Researchers are exploring using LLMs to implement question-answering systems [105] which have been demonstrated in past research in EUSE to be promising interfaces for humans to interrogate outputs of computer programs [53]. Exploring this idea in the context of the maintenance of generated applications is a promising research direction. One idea would be to develop a question-answering system of systems that combines domain-specific LLMs with software systems data (i.e., DOAs [16]) to answer end-users questions regarding the applications' behaviour, supporting debugging and tasks, and guiding users in using and understanding software. Techniques from HCI could support the development of system monitoring tools along these lines, as discussed in Section 3.1.

4 Discussion of Risks and Mitigations

The research agenda in this position article focuses on the technical challenges that emerge at different stages of the novel EUSE lifecycle depicted in Figure 1. The goal is to convert natural requirements into whole software applications by exploiting generative AI. While realising this goal, researchers must also consider and mitigate potential risks so as to ensure the creation of safe and sustainable software that benefits society. These risks will require interdisciplinary research efforts. In the following, we discuss potential risks and mitigations.

In some cases, AI-based algorithms for code synthesis will generate software whose behaviour incorporates biases against certain demographic groups. Moreover, a systematic literature review revealed that research on end-user programming is predominantly conducted by and evaluated with people from **Western, Educated, Industrialised, Rich, and Democratic (WEIRD)** societies [34, 43]. The research we advocate must consider these findings and involve a broader set of people; the broader perspective will likely help more people, from WEIRD societies or not. Similarly, mounting evidence shows that some usability issues affect women and men differently [12, 68]. Can we design automated requirements elicitation to include effective cognitive walkthroughs with a diverse range of personas to improve usability overall? The *Systems Engineering* approach [42] prioritises problems over solutions to build systems that effectively align with their stakeholders' intents. Such prioritisation can mitigate the risk of producing biased software by holistically considering software stakeholders in inclusive requirements definition processes. In addition, *algorithmic audit* is a 'method of repeatedly querying an algorithm and observing its output to analyse the algorithm's opaque inner workings and possible external impact' [65]. Suppliers of algorithms should assist in external algorithmic audits and help automate the administration of these.

Responsible AI research has led to large companies issuing guidelines and checklists [3] while researchers examine their applicability in practice [78]. We will need guidelines for responsible

requirements-driven EUSE. The non-deterministic nature of Generative AI can inject threats into synthesised applications with critical requirements. Minimising these threats is key towards the realisation of our research vision. Based in part on hazard analysis of Codex [50], Khlaff [108] proposes the ODD model, originally developed to categorise automated driving systems [97], to define concrete operational envelopes for AI-based systems. We should define a hierarchy of ODDs for requirements-driven EUSE to help audit and contain the operational risks. We consider current Generative AI technologies to require significant changes and progress to be used reliably and responsibly in automatically implementing safety-critical systems. As discussed earlier, we consider that state-of-the-art LLMs for code generation to suit the requirements of applications in Shaw's 'Good-enough Realm' [84] for now.

Adopting AI in our society is changing people's activities and tasks in their jobs and daily activities [28]. End-users and software engineers are not indifferent to such changes. In particular, the novel EUSE lifecycle in Figure 1 requires defining and developing new skills at different stages. For example, end-users must be able to interact with LLMs to define their intents clearly. Designing effective interfaces between humans and machines partially addresses this challenge. However, education and training efforts are required to empower people with the knowledge and skills they need to effectively exploit the benefits of the new tools and identify their limitations. We identify two exciting research areas that must cooperate to mitigate this risk. The first research area focuses on redefining competencies frameworks in the digital context [7], while the second explores enabling new education and training strategies based on AI [104].

Systems said to be powered by AI are often assisted by human labour, in practice. Examples include mobile phone apps, Web sites and even automated retail checkout systems, where human gig workers pick up on those image classification and other tasks that AI cannot autonomously decide. Anthropologist Mary Gray and computer scientist Siddharth Suri call these humans 'ghost workers' because they are invisible to the end-users of the services they provide [38]. They argue that this work force is deliberately concealed and is often underpaid and overworked. Systems that empower end-users with requirements-driven EUSE for the short-term—and quite possibly the longer-term—will need assistance from human UX researchers, requirements engineers and software engineers to help with requirements elicitation and effective delivery of reliable systems. For the ethical development of this application of AI, providers should be explicit about the role of human workers in the system and ensure that they are fairly compensated and treated with dignity.

Generative AI relies on energy-hungry platforms and infrastructure for data and model management, including GPU computing [32, 91]. Hence, there are fears that generative AI has a 'clean-energy problem'.[11] In parallel to the research agenda proposed in this article, research is needed on more efficient forms of generative AI to mitigate the threat to sustainability from individual, social, economic and environmental perspectives. Possible solutions may lie in methods like quantisation which reduces the precision of parameters of LLMs so that they can be run on cheaper hardware and reduce energy consumption [9, 64]. Another active area of research that addresses this area of environmental sustainability is small language models which can be more efficiently and affordably deployed and have been applied in several domains including coding [63]. Ideas around edge-computing architectures [17, 94] and federated learning [8] propose using the potential of every day and cheaper devices in distributed learning processes to mitigate the sustainability risk. Technical advances in these areas will enable the development of more sustainable tools for code synthesis.

[11] *The Economist: Generative AI has a clean-energy problem*, April 2024.

5 Conclusions

Breakthroughs in LLMs have enabled a paradigm shift in how users control the software development lifecycle. We propose a requirements-driven EUSE research agenda with three key areas of focus and concrete directions within these (see summary below). These areas constitute a novel EUSE lifecycle that drives our narrative on the research directions to make our vision a reality. In this article, we first contextualise the research areas using a case study and then provide a set of research questions that can trigger diverse research projects in this domain together with promising avenues for exploration.

Requirements-Driven EUSE Research Agenda

Requirements elicitation: assist end users in expressing their requirements

(1) Focus on software applications in Shaw's good-enough realm.
(2) Develop automated ways of accessing tacit knowledge.
(3) Build requirements datasets for LLM-based code synthesis and evaluation.

Testing: generate tests from requirements that are meaningful to end users

(1) Automatically generate tests for the user.
(2) Generate formal specifications and explain them to end users.
(3) Apply language-based security to LLM-generated code.

Maintenance and deployment: respond to dynamic requirements and environments

(1) Automate software deployment plans.
(2) Maintain software components synthesised by LLMs
(3) Develop novel interfaces for improving systems interpretability.

Our approach of *requirements-driven EUSE* is one in which end-users can conceptualise, test and deploy software entirely from requirements. We have looked at the limits to how far requirements can dictate system behaviour including non-functional requirements which must be automated and communicated clearly to the user. We argue that for some classes of application, beginning in Shaw's Good-enough Realm, this approach can solve Gray's research challenge of Automatic Programming [37].

Acknowledgements

We are grateful to Brendan Murphy and to the anonymous reviewers for their insightful comments on drafts of this article. Discussions at the *ACM International Workshop on Software Engineering in 2030* were also helpful.

References

[1] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. 2016. Sok: Lessons learned from Android security research for appified software platforms. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP '16)*. IEEE, 433–451.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys* 51, 4, Article 81 (Jul. 2018), 37 pages. DOI: https://doi.org/10.1145/3212695

[3] Saleema Amershi, Daniel S. Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi T. Iqbal, Paul N. Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 3. Retrieved from https://www.microsoft.com/en-us/research/publication/guidelines-for-human-ai-interaction/

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv:2108.07732.

[5] David Barrera, H. Güneş Kayacik, Paul C. Van Oorschot, and Anil Somayaji. 2010. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 73–84.

[6] Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, and Antonio Piccinno. 2019. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software* 149 (2019), 101–137. DOI: https://doi.org/10.1016/j.jss.2018.11.041

[7] Viviana Bastidas, Kwadwo Oti-Sarpong, Timea Nochta, Li Wan, Junqing Tang, and Jennifer Schooling. 2024. Leadership of urban digital innovation for public value: A competency framework. *IET Smart Cities* 6, 3 (2024), 237–252. DOI: https://doi.org/10.1049/smc2.12063

[8] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. In *Proceedings of the Machine Learning and Systems*. A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1, 374–388. Retrieved from https://proceedings.mlsys.org/paper/2019/file/bd686fd640be98efaae0091fa301e613-Paper.pdf

[9] Catherine Breslin. 2024. September roundup. Retrieved September 09, 2024 from https://aixinsights.substack.com/p/september-roundup

[10] Margaret Burnett. 2009. What is end-user software engineering and why does it Matter?. In *End-User Development*. Volkmar Pipek, Mary Beth Rosson, Boris de Ruyter, and Volker Wulf (Eds.), Springer, Berlin, 15–28.

[11] Margaret Burnett, Curtis Cook, and Gregg Rothermel. 2004. End-user software engineering. *Communications of the ACM* 47, 9 (Sep. 2004), 53–58.

[12] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. 2016. GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers* 28, 6 (2016), 760–787.

[13] M. M. Burnett and C. Scaffidi. 2014. End-user development. In *The Encyclopedia of Human-Computer Interaction (2nd. ed)*. Interaction Design Foundation - IxDF. Retrieved December 3, 2024 from https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/end-user-development

[14] Christian Cabrera and Siobhán Clarke. 2021. A reinforcement learning-based service model for the Internet of things. In *Service-Oriented Computing*. Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik (Eds.), Springer International Publishing, Cham, 790–799.

[15] Christian Cabrera, Andrei Paleyes, and Neil D. Lawrence. 2024. Self-sustaining software systems (S4): Towards improved interpretability and adaptation. In *Proceedings of the 2024 International Workshop New Trends in Software Architecture (SATrends '24)*. ACM, New York, NY, 5. DOI: https://doi.org/10.1145/3643657.3643910

[16] Christian Cabrera, Andrei Paleyes, Pierre Thodoroff, and Neil D. Lawrence. 2023. Real-world machine learning systems: A survey from a data-oriented architecture perspective. arXiv:2302.04810.

[17] Christian Cabrera, Sergej Svorobej, Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. 2023. MAACO: A dynamic service placement model for smart cities. *IEEE Transactions on Services Computing* 16, 1 (2023), 424–437. DOI: https://doi.org/10.1109/TSC.2022.3143029

[18] Matteo Camilli, Raffaela Mirandola, and Patrizia Scandurra. 2023. Enforcing resilience in cyber-physical systems via equilibrium verification at runtime. *ACM Transactions on Autonomous and Adaptive Systems* 18, 3 (Sep. 2023), 32 pages. DOI: https://doi.org/10.1145/3584364

[19] Matteo Camilli, Raffaela Mirandola, and Patrizia Scandurra. 2023. XSA: Explainable self-adaptation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. ACM, New York, NY, Article 189, 5 pages. DOI: https://doi.org/10.1145/3551349.3559552

[20] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code generation with generated tests. arXiv:2207.10397.

[21] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 16 (Mar. 2024), 17754–17762. DOI: https://doi.org/10.1609/aaai.v38i16.29728

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374

[23] Jennifer Cobbe, Michael Veale, and Jatinder Singh. 2023. Understanding accountability in algorithmic supply chains. In *Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency (FAccT '23)*. ACM, New York, NY, 1186–1197. DOI: https://doi.org/10.1145/3593013.3594073

[24] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering end users in debugging trigger-action rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19, Paper 388)*. ACM, New York, NY, 1–13.
[25] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch What I Do: Programming by Demonstration*. MIT press.
[26] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *Automated Deduction - CADE-25*. Amy P. Felty and Aart Middeldorp (Eds.), Lecture Notes in Computer Science, Vol. 9195, Springer, Cham, 378–388.
[27] Edsger W. Dijkstra. 1978. On the foolishness of "natural language programming". In *Program Construction, International Summer School*. Friedrich L. Bauer and Manfred Broy (Eds.), Lecture Notes in Computer Science, Vol. 69, Springer, 51–53. DOI: https://doi.org/10.1007/BFB0014656
[28] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2023. GPTs are GPTs: An early look at the labor market impact potential of large language models. arXiv:2303.10130. Retrieved from https://arxiv.org/abs/2303.10130
[29] Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros Araya, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2024. LLM4PLC: Harnessing large language models for verifiable programming of PLCs in industrial control systems. arXiv:2401.05443.
[30] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. arXiv:2310.03533.
[31] Kasra Ferdowsi, Jack Williams, Ian Drosos, Andrew D. Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar, and Benjamin Zorn. 2023. COLDECO: An end user spreadsheet inspection tool for AI-generated code. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '23)*. IEEE, 82–91. DOI: https://doi.org/10.1109/VL-HCC57772.2023.00017
[32] Stefan Feuerriegel, Jochen Hartmann, Christian Janiesch, and Patrick Zschech. 2024. Generative AI. *Business & Information Systems Engineering* 66 (2024), 111–126.
[33] Lynn Gilbert and Gaylen Moore. 1981. *Particular Passions: Grace Murray Hopper*. Lynn Gilbert. Retrieved from https://www.vassar.edu/stories/2017/assets/images/170706-legacy-of-grace-hopper-hopperpdf.pdf
[34] Harshit Goel, Aayush Kumar, and Sruti Srinivasa Ragavan. 2023. End-user programming is WEIRD: How, why and what to do about it. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '23)*. IEEE, 41–50. DOI: https://doi.org/10.1109/VL-HCC57772.2023.00013
[35] Li Gong. 2009. Java security: A ten year retrospective. In *Proceedings of the 2009 Annual Computer Security Applications Conference*. IEEE, 395–405.
[36] Andrew D. Gordon, Carina Negreanu, José Cambronero, Rasika Chakravarthy, Ian Drosos, Hao Fang, Bhaskar Mitra, Hannah Richardson, Advait Sarkar, Stephanie Simmons, et al. 2023. Co-audit: Tools to help humans double-check AI-generated content. arXiv:2310.01297.
[37] Jim Gray. 2003. What next?: A dozen information-technology research goals. *Journal of the ACM* 50, 1 (2003), 41–57. DOI: https://doi.org/10.1145/602382.602401
[38] Mary L. Gray and Siddharth Suri. 2019. *Ghost Work: How to Stop Silicon Valley from Building a New Global Underclass*. Eamon Dolan Books.
[39] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Communications of the ACM* 55, 8 (2012), 97–105.
[40] Phillip Guo. 2024. *Beyond Just Programming: A Vision of Generative AI as an End-to End Expert Data Science Consultant*. Cogna Tech Talk.
[41] Yanzhu Guo, Guokan Shang, Michalis Vazirgiannis, and Chloé Clavel. 2023. The curious decline of linguistic diversity: Training language models on synthetic text. arXiv:2311.09807.
[42] Reinhard Haberfellner, Olivier De Weck, Ernst Fricke, and Siegfried Vössner. 2019. *Systems Engineering: Fundamentals and Applications*. Springer.
[43] Joseph Henrich, Steven J. Heine, and Ara Norenzayan. 2010. The weirdest people in the world? *Behavioral and Brain Sciences* 33, 2–3 (2010), 61–83.
[44] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. arXiv:2308.00352
[45] Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M. Ziegler, Tim Maxwell, Newton Cheng, et al. 2024. Sleeper agents: Training deceptive LLMs that persist through safety training. arXiv:2401.05566. Retrieved from https://arxiv.org/abs/2401.05566
[46] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-Bench: Can language models resolve real-world github issues?. In *Proceedings of the 12th International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=VTF8yNQM66

[47] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future. arXiv:2408.02479. Retrieved from https://arxiv.org/abs/2408.02479

[48] Eunsuk Kang and Mary Shaw. 2024. tl; dr: Chill, y'all: AI will not devour SE. arXiv:2409.00764. Retrieved from https://arxiv.org/abs/2409.00764.

[49] Heidy Khlaaf. 2023. Toward comprehensive risk assessments and assurance of AI-based systems. Retrieved from https://www.trailofbits.com/documents/Toward_comprehensive_risk_assessments.pdf

[50] Heidy Khlaaf, Pamela Mishkin, Joshua Achiam, Gretchen Krueger, and Miles Brundage. 2022. A hazard analysis framework for code synthesis large language models. arXiv:2207.14157.

[51] Donald E. Knuth and Luis Trabb Pardo. 1980. The early development of programming languages. *A History of Computing in the Twentieth Century*, 197–273. Retrieved from https://archive.org/details/DTIC_ADA032123/page/n1/mode/2up

[52] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys* 43, 3, Article 21 (Apr. 2011), 44 pages. DOI: https://doi.org/10.1145/1922649.1922658

[53] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, 151–158.

[54] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 199–206. DOI: https://doi.org/10.1109/VLHCC.2004.47

[55] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. 2013. *Autonomic Computing: Principles, Design and Implementation*. Springer Science & Business Media.

[56] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. 1992. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10, 4 (1992), 265–310.

[57] William B. Langdon and Mark Harman. 2015. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (2015), 118–135. DOI: https://doi.org/10.1109/TEVC.2013.2281544

[58] Alexander Lavin, Ciarán M. Gilligan-Lee, Alessya Visnjic, Siddha Ganju, Dava Newman, Sujoy Ganguly, Danny Lange, Atílím Güneş Baydin, Amit Sharma, Adam Gibson, et al. 2022. Technology readiness levels for machine learning systems. *Nature Communications* 13, 1 (2022), 6039.

[59] Neil D. Lawrence. 2017. Data readiness levels. arXiv:1705.02245.

[60] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

[61] Henry Lieberman. 2007. End-user software engineering position paper. In *End-User Software Engineering*. Margaret H. Burnett, Gregor Engels, Brad A. Myers, and Gregg Rothermel (Eds.), Dagstuhl Seminar Proceedings (DagSemProc), Vol. 7081, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1. DOI: https://doi.org/10.4230/DagSemProc.07081.15

[62] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. ACM. DOI: https://doi.org/10.1145/3544548.3580817

[63] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D. Lane, and Mengwei Xu. 2024. Small language models: Survey, measurements, and insights. arXiv:2409.15790.

[64] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit LLMs: All large language models are in 1.58 bits. arXiv:2402.17764.

[65] Danaë Metaxa, Joon Sung Park, Ronald E. Robertson, Karrie Karahalios, Christo Wilson, Jeff Hancock, and Christian Sandvig. 2021. Auditing algorithms: Understanding algorithmic systems from the outside in. *Foundations and Trends in Human-Computer Interaction* 14, 4 (2021), 272–344. DOI: https://doi.org/10.1561/1100000083

[66] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 37 (Jul. 2024), 24 pages. DOI: https://doi.org/10.1145/3643763

[67] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. ClarifyGPT: Empowering LLM-based code generation with intention clarification. arXiv:2310.10996.

[68] Emerson Murphy-Hill, Alberto Elizondo, Ambar Murillo, Marian Harbach, Bogdan Vasilescu, Delphine Carlson, and Florian Dessloch. 2024. GenderMag improves discoverability in the field, especially for women. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. IEEE Computer Society, 973–973.

[69] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT press.
[70] Cyrus Omar, Patrick Ferris, and Anil Madhavapeddy. 2024. Modularizing reasoning about AI capabilities via abstract dijkstra monads. In *Proceedings of the 12th ACM SIGPLAN Workshop on Higher-Order Programming with Effects (HOPE'24)*.
[71] OpenAI. 2024. GPT-4 technical report. arXiv:2303.08774.
[72] Ipek Ozkaya. 2023. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software* 40, 3 (2023), 4–8. DOI: https://doi.org/10.1109/MS.2023.3248401
[73] Andrei Paleyes, Siyuan Guo, Bernhard Scholkopf, and Neil D. Lawrence. 2023. Dataflow graphs as complete causal graphs. In *Proceedings of the 2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN '23)*, 7–12. DOI: https://doi.org/10.1109/CAIN58948.2023.00010
[74] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. 2022. Challenges in deploying machine learning: A survey of case studies. *ACM Computing Surveys* (Apr. 2022). DOI: https://doi.org/10.1145/3533378
[75] Raymond R. Panko. 1998. What we know about spreadsheet errors. *Journal of End User Computing* 10, 2 (1998), 15–21.
[76] Martin Porcheron, Joel E. Fischer, and Stuart Reeves. 2021. Pulling back the curtain on the wizards of Oz. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW3 (Jan. 2021), 1–22.
[77] Leo Porter and Daniel Zingaro. 2023. *Learn AI-Assisted Python Programming with GitHub Copilot and ChatGPT*. Manning Publications, New York, NY.
[78] Bogdana Rakova, Jingying Yang, Henriette Cramer, and Rumman Chowdhury. 2021. Where responsible AI meets reality: Practitioner perspectives on enablers for shifting organizational practices. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW1 (2021), 7:1–7:23.
[79] Remington Rand Univac 1958. *FLOW-MATIC Programming System*. Remington Rand Univac. Retrieved from https://bitsavers.trailing-edge.com/pdf/univac/flow-matic/U1518_FLOW-MATIC_Programming_System_1958.pdf
[80] J. R. Ruthruff, A. Phalgune, L. Beckwith, M. Burnett, and C. Cook. 2004. Rewarding "good" behavior: End-user debugging and rewards. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*. IEEE, 115–122.
[81] Jean E. Sammet. 1978. *The Early History of COBOL*. ACM, New York, NY, 199–243. DOI: https://doi.org/10.1145/800025.1198367
[82] Advait Sarkar. 2023. Will code remain a relevant user interface for end-user programming with generative AI models?. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 153–167.
[83] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? arXiv:2208.06213.
[84] Mary Shaw. 2022. Myths and mythconceptions: What does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages* 4, HOPL (2022), 1–44.
[85] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. arXiv:2303.11366.
[86] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. 2023. The curse of recursion: Training on generated data makes models forget. arXiv:2305.17493.
[87] Auste Simkute, Lev Tankelevitch, Viktor Kewenig, Ava Elizabeth Scott, Abigail Sellen, and Sean Rintel. 2024. Ironies of generative AI: Understanding and mitigating productivity loss in human-AI interactions. arXiv:2402.11364.
[88] Jatinder Singh, Jennifer Cobbe, and Chris Norval. 2019. Decision provenance: Harnessing data flow for accountable systems. *IEEE Access* 7 (2019), 6562–6574. DOI: https://doi.org/10.1109/ACCESS.2018.2887201
[89] Sruti Srinivasa Ragavan, Zhitao Hou, Yun Wang, Andrew D. Gordon, Haidong Zhang, and Dongmei Zhang. 2022. Gridbook: Natural language formulas for the spreadsheet grid. In *Proceedings of the 27th International Conference on Intelligent User Interfaces*, 345–368.
[90] Apoorva Srivastava, Sukriti Bhardwaj, and Shipra Saraswat. 2017. SCRUM model for agile methodology. In *Proceedings of the 2017 International Conference on Computing, Communication and Automation (ICCCA '17)*. IEEE, 864–869.
[91] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. arXiv:1906.02243.
[92] Chuyue Sun, Ying Sheng, Oded Padon, and Clark W. Barrett. 2024. Clover: Closed-loop verifiable code generation. In *AI Verification SAIV*. Guy Avni, Mirco Giacobbe, Taylor T. Johnson, Guy Katz, Anna Lukina, Nina Narodytska, and Christian Schilling (Eds.), Lecture Notes in Computer Science, Vol. 14846, Springer, Cham, 134–155. Retrieved from https://doi.org/10.48550/arXiv.1906.02243
[93] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices* 46, 9 (2011), 266–278.

[94] Hadi Tabatabaee Malazi, Saqib Rasool Chaudhry, Aqeel Kazmi, Andrei Palade, Christian Cabrera, Gary White, and Siobhán Clarke. 2022. Dynamic service placement in multi-access edge computing: A systematic literature review. *IEEE Access* 10 (2022), 32639–32688. DOI: https://doi.org/10.1109/ACCESS.2022.3160738

[95] Lev Tankelevitch, Viktor Kewenig, Auste Simkute, Ava Elizabeth Scott, Advait Sarkar, Abigail Sellen, and Sean Rintel. 2024. The metacognitive demands and opportunities of generative AI. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 1–24.

[96] Gemini Team. 2023. Gemini: A family of highly capable multimodal models. arXiv:2312.11805.

[97] Eric Thorn, Shawn C. Kimmel, and Michelle Chaka. 2018. *A Framework for Automated Driving System Testable Cases and Scenarios*. Technical Report DOT HS 812 623. United States Department of Transportation. Retrieved from https://doi.org/10.48550/arXiv.2312.11805

[98] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-driven development. arXiv:2403.08299.

[99] A. M. Turing. 1950. Computing machinery and intelligence. *Mind* LIX, 236 (Oct. 1950), 433–460. DOI: https://doi.org/10.1093/mind/LIX.236.433

[100] Priyan Vaithilingam, Ian Arawjo, and Elena L. Glassman. 2024. Imagining a future of designing with AI: Dynamic grounding, constructive negotiation, and sustainable motivation. arXiv:2402.07342.

[101] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. 2023. Making network configuration human friendly. arxiv:2309.06342.

[102] Danny Weyns. 2019. Software engineering of self-adaptive systems. In *Handbook of Software Engineering*. Sungdeok Cha, Richard N. Taylor, and Kyochul Kang (Eds.), Springer, 399–443.

[103] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2023. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155.

[104] Kumar Yelamarthi, Raju Dandu, Mohan Rao, Venkata Prasanth Yanambaka, and Satish Mahajan. 2024. Exploring the potential of generative AI in shaping engineering education: Opportunities and challenges. *Journal of Engineering Education Transformations* 37, Special Issue 2 (2024), 439–445.

[105] Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. ToolQA: A dataset for LLM question answering with external tools. *Advances in Neural Information Processing Systems* 36 (2023), 50117–50143.

[106] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot'S impact on productivity. *Communications of the ACM* 67, 3 (Feb. 2024), 54–63. DOI: https://doi.org/10.1145/3633453

[107] Mehrotra Pronita, Parab Aishni, and Gulwani Sumit. 2024. Enhancing creativity in large language models through associative thinking strategies. arXiv: 2405.06715. Retrieved from https://doi.org/10.48550/arXiv.2405.06715

[108] Khlaaf Heidy. 2023. *Toward Comprehensive Risk Assessments and Assurance of AI-Based Systems*. Retrieved from https://www.trailofbits.com/documents/Toward_comprehensive_risk_assessments.pdf

Received 5 April 2024; revised 1 October 2024; accepted 15 November 2024