

Will Code Remain a Relevant User Interface for End-User Programming with Generative AI Models?

Advait Sarkar

advait@microsoft.com

Microsoft Research and University of Cambridge
Cambridge, United Kingdom
University College London
London, United Kingdom

Abstract

The research field of end-user programming has largely been concerned with helping non-experts learn to code sufficiently well in order to achieve their tasks. Generative AI stands to obviate this entirely by allowing users to generate code from naturalistic language prompts. In this essay, we explore the extent to which “traditional” programming languages remain relevant for non-expert end-user programmers in a world with generative AI. We posit the “generative shift hypothesis”: that generative AI will create qualitative and quantitative expansions in the traditional scope of end-user programming. We outline some reasons that traditional programming languages may still be relevant and useful for end-user programmers. We speculate whether each of these reasons might be fundamental and enduring, or whether they may disappear with further improvements and innovations in generative AI. Finally, we articulate a set of implications for end-user programming research, including the possibility of needing to revisit many well-established core concepts, such as Ko’s learning barriers and Blackwell’s attention investment model.

CCS Concepts: • **Human-centered computing** → **HCI theory, concepts and models**; **Natural language interfaces**; • **Computing methodologies** → *Natural language processing*; *Neural networks*; • **Social and professional topics** → *User characteristics*.

Keywords: generative shift hypothesis, prompt engineering, end-user software customization, attention investment model, learning barriers, self-efficacy, live programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. *Onward! '23, October 25–27, 2023, Cascais, Portugal*

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0388-1/23/10...\$15.00

<https://doi.org/10.1145/3622758.3622882>

ACM Reference Format:

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 (Onward! '23)*, October 25–27, 2023, Cascais, Portugal. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3622758.3622882>

1 The Status Quo for End-user Programming Research

End-user programming (EUP) is the activity of writing a program for one’s own use; one is both the programmer and the end-user of the program [53]. This can be contrasted with software development, in which the programmers are typically not going to be people using the system. There is a huge variety of end-user programmers (EUPs) and end-user programming activities, such as writing spreadsheet formulas to analyse data, or a script to automate a daily workflow, or even programming as a hobby for personal creative satisfaction [1].

Unlike professional software developers, EUPs typically have very little or no formal training in programming or computing. This gap in expertise is the primary challenge faced by EUPs trying to achieve their task (though there are other differences between EUPs and professionals, such as their motivations for programming, strategies for learning, debugging, etc. which add further complexities).

End-user programming research thus aims to bridge this gap. So far, its aim can be described as improving the ability of EUPs to use *formal systems*. The term “formal systems” is a shorthand to mean a predictable and deterministic interface which may include a programming language (e.g., spreadsheet formulas, or block programming in Scratch [86]) and other interface elements (e.g., the spreadsheet grid).

EUP research helps users with formal systems in three ways. The first approach is to help users *learn* how to use formal systems. This concern is shared by computer science education research (CSER) [28], and includes research on novice programming, the design of programming tutorial aids, multiple representations systems [105], etc.

The second approach is to reduce or scaffold the expertise required to use formal systems. This is the motivation behind

many visual programming languages, such as Scratch [86], which uses blocks and slots of various shapes to anticipate and prevent syntactic and type errors, since a block of a certain shape can only fit into a corresponding slot.

The third approach aims to reduce the reliance on coding, such as programming by example (PBE) [62] and programming by demonstration (PBD) [20]. Thus, by showing examples of input and output, or manually demonstrating a particular process, users can define intended behaviour (code generation is often facilitated by a method such as program synthesis [70]). While the aim is to reduce the reliance on coding, in practice many PBE/PBD systems do not entirely avoid interaction with the generated code. For instance, Flash Fill [32], a commercially implemented PBE system for string manipulation formulas in spreadsheets, was initially deployed without showing the generated programs to the user, but was later extended to show the user its generated formulas (in some cases) for explanation, verification, and debugging.

In summary of the status quo: EUP research aims to help users avail of formal programming systems by improving their ability to learn, to reduce the expertise requirements of formal systems, or by offering alternatives to writing code.

2 The Scope of the Term “Generative AI” for End-User Programming

The term “generative AI” is extremely broad, encompassing many types of system with different capabilities, sometimes referring to a core algorithm (such as the transformer [113]), a particular instantiated model (such as GPT-4 [79]; these are also sometimes referred to as “foundation models” [12]), or a productised system which may be comprised of an ensemble of multiple models together with prompt engineering, safety heuristics, and user interface affordances (such as ChatGPT¹). In this respect “generative AI” has taken on a similarly pluralistic nature as terms such as “machine learning” and “artificial intelligence” which can span an immense range of tools and techniques [67].

The additional descriptor “generative” is a reference to the fact that these models can *generate* information artefacts (such as images or text), by modelling a high-dimensional space from its input data from which new points can be sampled. This is as opposed to discriminative models, which can classify, label, score, or transform their input based on training examples, but do not explicitly model the input space [58]. It must be noted that while the term “generative AI” has only come into widespread usage in 2023 and is used to refer to contemporary generative models, research into generative models long predates this [26, 87], and “latent variable models” were proposed as early as the turn of the 20th century [15].

To focus the scope of the discussion in this paper, it is worth defining “generative AI” in terms of the aspects which are of greatest interest to EUPs, as well as the aspects which differentiate “generative AI” from previous generations of AI tools which have also been applied in the EUP context.

First, we are talking about *tools* which are directly used by EUPs. An EUP does not interact with an algorithm, or a specific model, but a tool which may consist of multiple models that are part of a broader system of heuristics, prompt engineering, and user interface elements. From the perspective of the end-user experience, there are many aspects of these heuristics and interface elements which impact their ability to use the underlying model.²

Second, we are talking about tools which are used in a *programming* context. There are many ways to define programming, but a particularly useful and influential definition in end-user programming research is that programming is any activity exhibiting “the property that the user is not directly manipulating observable things, but specifying behaviour to occur at some future time” [11]. Generative AI tools can be applied in a wide variety of contexts, and many of them relate to the direct production of artefacts (text, images, etc.) which are not interpreted as specifying behaviour to occur in the future – we exclude these from consideration.

Finally, we are talking about tools built on *contemporary* machine learning techniques as of this writing in 2023. Despite the fact that generative models have a long history, generative AI models, such as large language models [13] and image generation models [85] have recently seen a significant step-change in capabilities. The reason for this is a combination of advances in hardware (such as GPU clusters for training), algorithms (such as the transformer architecture), and the availability of Internet-scale datasets [94]. This has enabled models to achieve human-level performance for the first time in a wide variety of benchmarks including code generation, speech recognition, image generation, even passing the bar exam [46]. This is the latest development in a period typically dated to begin in 2016 that has been described as the “third summer” of AI,³ following a common periodisation of AI research as measured by “rapid scientific advances, broad commercialisation, and exuberance” [47]. A relatively stable term of art accepted and advocated within the AI research community that encapsulates the advances of the third summer is “deep learning” [60], which is broad enough to encompass a variety of approaches developed in

²There are also elements that impact usability but have little or no connection with the underlying model, such as the text in dialogue boxes and the placement of menu buttons, which are not of concern here.

³2016 is the year that Deepmind’s AlphaGo program [100] beat grandmaster Lee Sedol, though some scholars point as early as 2012, referring to the so-called “ImageNet moment” referring to the solution of the ImageNet challenge by deep convolutional networks [56], while others look as recently as 2018, referring to the BERT model which similarly swept natural language processing benchmarks and is referred to as “NLP’s ImageNet moment” [24]. A detailed historiography of the periodisation of AI research is out of scope.

¹<https://openai.com/blog/chatgpt> (last accessed 23 June 2023)

recent years while being specific enough to exclude older generative approaches.

Putting these considerations together, the following definition is adopted for the remainder of the paper:

By generative AI, we mean *an end-user tool, applied to programming, whose technical implementation includes a generative model based on deep learning.*

Thus, some concrete examples which fit the definition and scope of generative AI in this paper are: GitHub Copilot⁴, an end-user tool which offers code completion assistance based on the OpenAI Codex large language model; end-user tools for naturalistic language-based programming in spreadsheets such as those built by Liu et al. [64] (also using Codex); and ChatGPT when it is being used to generate code, debug code, or explore APIs and documentation.

Some concrete examples which do *not* fit the definition and scope of generative AI in this paper are: CodeT5 [116], a code generating large language model (uses deep learning, is programming-oriented, but is not an end-user tool); early programming-by-demonstration systems such as Allen Cypher’s Eager [19] (is programming oriented, is an end-user tool, but does not use deep learning); and ChatGPT when it is being used to generate a short fiction story (uses deep learning, is an end-user tool, but is not supporting a programming activity).

3 Motivations for Applying Generative AI to End-User Programming

Why are the capabilities of generative AI important or beneficial for EUP, and how does it differ from existing approaches? As we have seen, EUP research is concerned with helping users achieve programming tasks by improving learning, reducing expertise requirements, and reducing reliance on coding. AI approaches prior to deep learning have been applied in various ways to each of these aims, such as generating and recommending tutorials [51, 122], detecting errors and suggesting fixes [36], and the generation of code from demonstrations [20].

In several EUP activities, generative AI creates improvements in *degree* over previous approaches. That is, it improves performance in terms of scope and accuracy. For example, let us consider five categories of EUP activity in which generative AI creates an improvement in degree: authoring, debugging, reuse, comprehension, and learning.

Authoring. The principal activity of EUPs is authoring, that is, actually writing code. Quantitative evidence for the improvement in degree afforded by generative AI is given by tracing the performance of various approaches in coding benchmark tests [4, 42, 112], where it is now possible to

solve a much wider variety of programming tasks using natural language descriptions of the problem, and success rates for programming tasks described as being representative of “entry-level programmer” skill now regularly approaches 80%.

Debugging. Another key activity of EUPs is in debugging: detecting and fixing errors in existing code. Here again, quantitative studies and benchmarks of error detection and fixing show a significant improvement in degree in comparison to previous approaches [27, 44].

Reuse. Often, EUPs do not seek to directly author code starting from a blank slate, rather, they seek to reuse code available on the Web, code they have previously written, or code written by colleagues or stored in institutional repositories [59, 103]. There are several key challenges in the reuse activity, principally, locating relevant code, evaluating its suitability for reuse, and making necessary modifications to make the code fit their current context. Studies have found that even with interactive support for code reuse, EUPs can struggle to adapt code due to a lack of formal programming expertise [59]. Generative AI tools excel at facilitating reuse and adaptation of code, and mark a significant improvement in degree in comparison to previous approaches [94].

Comprehension. EUPs often seek to understand code for reasons other than debugging. For example, users receiving an unfamiliar spreadsheet from a colleague may review formulas in the spreadsheet to understand the underlying data sources for a particular value in their spreadsheet which is relevant to a decision they need to make [95, 104]. Studies have shown that generative AI has significantly improved capabilities in code explanation and summarisation for students and non-experts (though challenges remain) [44, 68].

Learning. A wide variety of AI approaches to facilitate learning for EUPs have been previously explored, from suggesting relevant tutorials [51, 122], to proactive suggestions for improving the likelihood of EUPs to write test cases to improve the robustness of their programs [120], and overcoming conceptual learning barriers [41]. However, a key challenge remains in adapting tutorials and learning materials to an individual’s problem context (what exactly are they trying to solve), their existing knowledge, and their learning style [14, 91]. Studies of spreadsheet learning show that learning from colleagues or “hallway experts” is common because it is an ideal learning situation: the colleague understands the user’s problem context and can tailor an explanation precisely to the learner’s needs [92]. This “holy grail” of personalised learning is another area where generative AI has shown to provide a significant improvement in degree, where explanations can be generated on an ad hoc basis for arbitrary code with arbitrary amounts of detail [23, 69], and it is likely that such explanations can be tailored with much greater precision to a particular learner in a particular instance.

⁴<https://github.com/features/copilot> (last accessed 25 April 2023)

Besides improvements in degree, generative AI also creates improvements in *kind*, that is to say, it can broadly shift EUP activities in novel directions, or enable kinds of assistance that were not possible with previous approaches. For example, let us consider the cases of one-off automations and exploratory programming.

One-off automations. Recall our working definition of programming as “specifying behaviour to occur at some future time”. This may suggest that EUPs write programs much as software developers do, as reusable assets which can be used repeatedly over time on different input data. Certainly many end-user automations are like this (consider the formulas in organisational spreadsheets which are often highly “templated”, reused broadly and longitudinally across the organisation, sometimes for many years [35]). However, a lot of EUP is *not* like this, where instead the objective is to develop ad-hoc data processing or cleaning scripts which are used exactly once for a specific transformation and then discarded. In such one-off automations a lot of interesting end-user programming behaviour is observed, such as accepting errors or brittleness in the program, and transforming the data through a patchwork of manual and automated steps [81, 91]. Generative AI technologies are capable of transforming, cleaning, and augmenting data *directly*, which would eliminate the need for writing a reusable script for one-off automations entirely [38, 115]. This would shift the scope of EUP activities away from writing such automations, but it would also create new challenges for EUPs, for instance in verifying that such transformations have been done correctly, and in making the occasional transition from a fundamentally opaque transformation performed by a generative AI model to a more repeatable and well-understood script.

Exploratory programming. Finally, many EUP activities involve an exploratory aspect; where the programming objective is not known *a priori* but discovered through trial and error and experimentation [48]. This is not just true of artistic applications of programming such as the generation of digital art or live coding music [17], where the exploration may not converge to a formally “correct” solution (rather the exploration converges to a state of subjectively assessed completion), but is also true of situations such as exploratory data analysis where the rough forms of acceptable solutions may be known beforehand, but the precise procedures are still to be determined [76]. Here, generative AI not only increases the rapidity of such exploration but also enables forms of assistance that were not possible with previous approaches, such as the generation of alternative narratives to trigger nonlinear “leaps” in the user’s thinking (though this is yet to be investigated in an EUP setting) [90, 101].

4 The Potential Intensification of End-User Programming: The Generative Shift Hypothesis

Generative AI can be applied to generate code in a traditional programming language based on natural (or naturalistic) language prompts. A brief overview of large language models for code generation is given in Sarkar et al. [94]. This technology has already been commercialised in a number of code editors and extensions, such as GitHub Copilot.⁵

As a method for enabling people to program without writing code directly, generative AI can be viewed as an evolution of previous methods such as PBE, PBD, and older syntax-directed or machine-learning based code autocompletion.

In other ways, as explained in Section 3, the maturation of generative AI may herald a revolution in capabilities, in particular for EUPs. This is what we will refer to as the *generative shift hypothesis*: a radical widening in scope and capability of EUP due in particular to increasing use of generative models.

The generative shift hypothesis posits the following qualitative and quantitative shifts:

- EUP will be applied *more intensively* to existing tasks, with more sophisticated and deeper automation being applied to scenarios which are already sites of EUP (such as spreadsheets).
- EUP will be applied in *more contexts* and to more tasks than before, which were previously not sites for the widespread application of EUP. This includes scripting of tasks across applications and across data sources (this is related to the enterprise concept of “robotic process automation”).
- EUP will be applied *more frequently*. The vastly reduced costs of generating code from naturalistic utterances will shift the attention investment [10] balance, making it more practical to attempt to automate tasks more often.

It is not within the scope of this essay to gather evidence for or against the generative shift hypothesis, and assess whether, and how fast, it might happen. Rather, we take the position of assuming it will happen, and attempting to analyse the role of formal systems in such a future.

Effectively guiding a generative model to produce the desired outcome is an area of active research (e.g., [65, 106, 118]). Current commercial tools can generate code from naturalistic language “prompts”, or auto-complete partially written code, or a combination of both. For non-programmers, the main mode of interaction is via language. This opens the possibility for EUP to become an activity that does not rely on learning or engaging with a formal programming system at all.

⁵<https://github.com/features/copilot> (last accessed 25 April 2023)

Generative AI therefore raises a critical, perhaps existential question for EUP research: what is the role of the current research agenda which aims to help users learn formal systems, or reduce the expertise requirements of formal systems? In a world where generative models are highly performant at translating naturalistic language into code, is there still a role for learning and interaction with formal systems for EUPs? Put simply: *will code remain a relevant user interface for end-user programming?*

5 Does Code Still Matter? Evaluating the Value Propositions of Formal Systems

To answer whether code may become irrelevant for EUPs, it may help to enumerate some current reasons that using a formal system, or combining a formal system with generative AI, might still result in a superior user experience than interacting purely through informal language. This section will discuss the following, possibly incomplete, list of reasons why code still matters: limited abilities, control, agency, awareness of the possible, explanation and interpretation, debugging, trust, and the illusion of informality. We will also consider the extent to which each value proposition might endure or become less relevant, as generative AI improves.

Limited abilities. The first reason is simply that generative AI can fail to correctly “translate” a natural language prompt into a suitable and correct program. It is difficult to draw clear boundaries around what these models can and cannot do. For one, they are opaque and non-deterministic, and can produce widely divergent responses to the same query. For another, the response is heavily influenced by the prompt, the training data, hyperparameters, and any other heuristics or processing applied to the input or output in a particular system. Finally, the state of the art is rapidly evolving. Thus, it would not be appropriate to make general statements such as “*generative AI cannot currently solve problem X*” because it makes a universal claim about the capabilities of every system, with their own ensemble of models, heuristics, training data, etc. Rather, every system’s response is fundamentally unique and varies depending upon the prompt, heuristics, and hyperparameters.

With these caveats in mind, it is worth looking at a small set of examples of generative AI failure, only to illustrate the kinds of problems that are not *typically* solved by end-user tools as of this writing in 2023.

The first example (Listing 1) comes from a paper exploring the limitations of GitHub copilot for CS1 problems [22], where copilot generates an incorrect function for calculating the average length of strings in a list, by generating nonsensical arithmetic. The second example (Listing 2) comes from Khatri et al. [50] and shows another simple example where the generated code satisfies one of the requested conditionals but ignores the other.

```

1 # Given a list of strings and a string s,
2 # return the average length of all strings
3 # containing s.
4
5 def averageContainingStringS(strings, s):
6     count = 0
7     for string in strings:
8         if s in string:
9             count += 1
10    return len(s) * count / len(strings)

```

Listing 1. Python code generation failure example from Denny et al. [22].

```

1 Prompt: "Select all rows where the entry in column
2 'gamma' is less than 40 and select all rows where
3 the entry in column 'gamma' is more than 53"
4
5 Response:
6 Table.SelectRows("#Table1", each [gamma]<40)

```

Listing 2. Code generation failure example from Khatri et al. [50].

```

1 Prompt: "How many super bowls has New Orleans won"
2
3 Response:
4 df[df['Host City'] == 'New Orleans']['Winner'].
   count()

```

Listing 3. Python code generation failure example from Liu et al. [64].

The final example (Listing 3) is drawn from a paper applying code generation models to data analysis problems in spreadsheets, where the full problem context can be seen [64]. In this case, the code has incorrectly selected the “Host City” column to compute a count of relevant strings, whereas it should have selected the “Winner” column. Liu et al. [64] characterise several types of failure modes with examples, such as the generation of non-executable code, selection of incorrect input columns, incorrect output formats (e.g., the generated code overwrites a column instead of generating a new one), or generating raw data output rather than a calculation.

As these examples suggest, there are situations where generating the required code is either beyond the capabilities of generative AI, or for some reason or another incorrect code is generated even if it is within current capabilities. To cope with such situations, EUPs might need to engage with a formal system.

On the other hand, at the time of writing in early 2023, we are in a transitional moment with generative AI technology. It is clear that generative models will continue improving due

to the current strategy of increasing the scaling of parameters and training data.⁶

Therefore, the problem of limited abilities, by definition, may seem to ameliorate as generative models improve. However, as the capabilities for automation improve, so might the demand for automation; it would be a fallacy to assume that EUP demands are stable over time and that today's EUP tasks are representative of future ones. Transport planners often engage in road-widening schemes to alleviate traffic congestion, only to find that the increased capacity is immediately consumed by increased demand [71]. In the 19th century, economist William Jevons observed that improvements in the efficiency of coal engines paradoxically resulted in an increased demand for coal [2]. These are examples of “rebound effects”, where gains in efficiency, capacity, and capability, are offset by changes in human behaviour. Similarly, the improvement of generative AI in addressing basic tasks may result in end-user demand for even more sophisticated and nuanced automation, which again shifts the goalposts for AI. On the other hand, EUPs already find engaging with formal systems a challenge; it is therefore difficult to see how engagement with formal systems might be a suitable fallback in the nuanced cases where generative models of the future fail.

Control. The second reason is to give direct and nuanced control over the behaviour of the system. Composing a program using a formal system allows users to directly express both “what” to compute and “how” to compute it. It allows EUPs to directly implement desired operations, as well as avoid undesired ones (e.g., avoiding a very slow API call, or avoiding a lookup of private data if it is unnecessary).

Formal systems offer direct and nuanced control, but varying degrees of control can also be achieved using informal systems. For instance, faceted natural language prompts which specify input and output types, or decompose a problem into smaller steps, offer increased control while retaining the relative informality of natural language [64, 74]. As generative models improve in performance, providing control without resorting to formal systems will be an important target for the EUP research agenda.

Agency. Related to control, the third reason is agency [18], a term from cognitive neuroscience referring to the feeling of being an active agent able to effect change in the world. A sense of agency improves the EUP experience, a lack of agency worsens it. Learning to code in a formal system has been associated with a sense of agency, and the related concept of computer self-efficacy [7].

It is important to note that while control and agency are related, they are distinct constructs. Control refers to the perception of influence over the external environment, whereas

agency focuses on the perception of being an intentional actor who can initiate and execute actions. Both control and agency play crucial roles in human experience, motivation, and well-being.

Though they are often interdependent, it is possible to have agency without (some types of) control. For example, “*agency is influenced by control specified at different hierarchical levels. [...] when higher-level control is exercised (i.e., goal-level control) lower level control processes (i.e., perceptuo-motor control) have no influence on sense of agency*” [63]. Conversely, it is also possible to have (some types of) control without agency. For instance, studies of assisted mouse cursor pointing (an example of a lower-level perceptuo-motor process) have shown that “*beyond a certain level of assistance users experienced a detectable loss in their sense of agency*”, even when the user had full goal-level control (i.e., the final destination of the cursor) [18].

An interesting question posed by the generative shift for interface design, therefore, is what types of control over the generation process are necessary or sufficient for creating a sense of agency. Direct control via a formal system can create a sense of agency, but just as there are alternative ways of offering control, so there are alternative ways of fostering agency. At one extreme, the sense of agency can be manipulated using placebo controls (such as the notorious pedestrian traffic light buttons that serve only to pacify impatient pedestrians and have no effect on the scheduled changes of the lights [72]). Moreover, user agency is a complex, multi-level phenomenon: the overall agency of an EUP might increase with the increased automation capabilities offered by generative AI, even if they find it difficult to control the behaviour of the AI in particular instances due to the lack of a formal system. As with control, fostering and maintaining an appropriate sense of agency will be an important target for the EUP research agenda.

Awareness of the possible. The fourth reason that learning and using a formal system is still useful is the “awareness of the possible” [91]; experience with a formal system gives users a thought language within which to define problems and envision potential solutions. Knowing *that* something can be automated is as important as knowing *how* to automate it; a user with a powerful generative AI that can automate some task is still at a disadvantage if they do not conceive of that task as something which can be automated.

This benefit of formal systems may be surpassed by improved generative AI as well as other interaction design techniques. Generative models may be able to suggest potential suitable automations and different strategies for implementing them, educating the user along the way. For example, in the field of data analysis, the aim of Intelligent Discovery Assistants (IDAs) has been to support users to form a strategy for achieving some analysis goal [98]; similarly research into exploratory programming [48] aims to support users in shaping their goals through experimentation. These fields

⁶This trend is almost certainly not going to continue indefinitely; we are approaching limits of computation and data availability, and others have pointed out various limitations to the so-called “scaling hypothesis” [45, 89, 99, 114], but a discussion of this is out of scope.

have produced general interaction techniques for visualising and interacting with a possibility space which may be applied to interaction with generative models, without requiring end-users to engage with code. As with the previous points, facilitating this awareness with generative models will also be an important target for EUP research.

Explanation, interpretation, and debugging. Research has shown that explaining the behaviour of AI-driven EUP systems is desirable for many reasons [57]; allowing EUPs to engage with the formal system(s) in which a generative model might emit scripts is one way (albeit not the only way) to provide explanation and interpretability.

Debugging is closely related to explanation and interpretation. Engagement with formal systems is a potentially superior way for EUPs to find and fix errors, than engaging with a generative model purely through natural language. For example, if the script generated by a generative model contains an off-by-one error, viewing the generated code might make it significantly easier to identify and correct it.

Formal systems offer a predictable route towards explanation, interpretation, and debugging. Yet each of these already presents a formidable challenge for EUPs, indeed much EUP research focuses on helping EUPs carry out each of these activities by scaffolding formal systems to overcome their limitations (e.g., the “idea garden” [41]). The design challenge is likely to increase further as generative models improve at synthesizing more sophisticated programs. We probably want different approaches to explanation which do not rely on engagement with formal systems; EUP and interactive machine learning research has already been developing various techniques that may be suitable starting points [54, 57].

Trust. Formal systems allow EUPs to verify the behaviour of the generated code in terms of well-understood and well-trusted procedures. For instance, when a spreadsheet user sees that a sum is calculated with the spreadsheet SUM function, they trust that it is correctly computing the sum because it is a deterministic algorithm written and checked by a trusted party (e.g., Microsoft or Google). Inspecting the formal output of a generative model may therefore be an important route towards trusting the result. It is worth noting here, though, that code “correctness” is not as important to EUPs as it is to professional software developers; research has shown that EUPs commonly use incorrect or incomplete code if it helps them partially automate their workflow, and are happy to manually correct errors, or accept errors which they see as irrelevant to the task at hand [81].

Expressing a computation in terms of trusted and verified operations (such as SUM) is an important route for establishing trust in the correctness of generated code. This strength of formal systems is unlikely to disappear simply by improving the performance of current models. Generative models can be asked to explain code in natural language, but they are prone to hallucination, as their output represents statistically likely completions rather than true and verifiable statements

[80]. In the future, it may be possible to constrain explanations of a script to refer to provably correct and verifiable procedures. To pursue this strategy and develop usable EUP systems along these lines will require a collaborative effort between researchers skilled in generative AI, programming languages, and human-centric end-user programming.

Illusion of informality. The eighth and final reason is slightly unlike the others, and is rather a meta-level observation about the nature of using generative models. The observation is that the disappearance of formality may be an illusion; generative models still require high levels of craft expertise to use effectively, and the shift to “prompt engineering” hasn’t eliminated programming at all, but simply shifted it into a higher level of abstraction. This is a perspective harboured by some (though not all) professional software developers who use generative AI assistance in their work [94].

As generative models get better, will “prompt engineering” remain a form of skilled craft practice that is distinct from ordinary communication skills, or will generative models reach a point where they can interpret truly arbitrary natural language at or exceeding human proficiency? On one hand, Miller and others have pointed out that a major hurdle for AI to interpret the meaning of a query is understanding the context of that query [73, 89], with the implication that future models, which will become better at including or inferring this context, will allow for more informal querying styles. On the other hand, the imprecision of natural language, particularly when it comes to discourse on matters of logic, mathematics, philosophy, and science, has long been seen as a major drawback and spurred many attempts to design more logically “perfect” languages [25, 78]. Indeed, the program of analytic philosophy which was born out of such concerns eventually gave rise to our modern programming languages [3], and it is interesting that programming via generative models brings us back, full circle, to natural language. However, the trend of improving generative models seems to imply that rather than “prompt engineering” remaining just like programming but at a higher level of abstraction, the application of language formality for precision, brevity, etc. in future systems will become more social and discretionary. Language for instructing generative models may develop much as scientific or legal language develops as a way for scientists and lawyers to communicate more effectively: organically, through power relations enacted by individuals, groups, and institutions [29].

6 Limitations and Challenges Posed by Generative AI in End-User Programming

There are many limitations and challenges of applying generative AI in end-user programming. Many of the challenges and ethical issues around trust, verification, privacy, bias,

credit, and accountability are not unique to end-user programming: they apply to all applications of AI to software development, and at the broadest level they apply to all applications of AI.

The focus of this paper is on the role of formal systems in the future of end-user programming, and a detailed discussion of ethical challenges would diverge from this focus. These have been extensively reviewed in recent research [102, 109, 117]. Nonetheless, it is worth briefly acknowledging these challenges to highlight that the generative shift is not an uncontestedly positive improvement for end-user programmers.

Problems for CS education. There is concern that generative AI may undermine the learning of novice programmers [6, 23]. Besides issues of academic integrity and misconduct, educators are apprehensive that over-reliance on generative AI may hinder the development of novice programmers' metacognition and encourage what is considered "bad habits" in code.

Errors. Generative AI systems may not always produce high-quality outputs, and the generated outputs may contain errors or inconsistencies that are hard to detect or correct by end-user programmers [94].

Trust and transparency. Generative AI systems may not be able to explain how they generate their outputs, or why they choose certain outputs over others [89]. This may make it difficult for end-user programmers to trust or verify the generated code, or to understand its logic and functionality. They may not provide any documentation or comments for the generated code, or provide incorrect documentation, or may not allow the end-user programmer to modify or customise the code.

Privacy and security. Code generated by generative AI systems may collect, store, or use personal or sensitive data from end-users or other sources, such as health records, biometric data, or financial data. This may raise concerns about the protection of data privacy and security, and the potential for data breaches, misuse, or abuse. They may access or expose confidential or proprietary information from end-users or their clients [34].

Misinformation. Generative AI systems may generate outputs that are false, misleading, or deceptive, either intentionally or unintentionally. Generative AI models can produce outputs that are coherent and convincing, but not necessarily accurate or factual. They can also invent references and sources that do not exist, or contain biases from the training data. This can mislead or confuse end-user programmers who rely on generated code for their own use. For example, a generative AI tool may generate a code snippet that performs a data analysis, but the results may be incorrect or skewed by the underlying data or model assumptions. This may create risks of misinformation and manipulation, and undermine the trustworthiness and credibility of information sources.

Bias and discrimination. Generative AI systems may generate outputs that are unfair, biased, or discriminatory, either intentionally or unintentionally. They may generate code that reflects or amplifies existing social biases or stereotypes, such as gender, race, or ethnicity [8]. This may result in harms or injustices to individuals or groups, such as exclusion, marginalisation, or oppression.

Accountability. Generative AI systems may generate outputs that have significant impacts on end-users or other parties, such as legal, financial, or health outcomes. They may generate code that violates laws, regulations, or ethical norms, or that causes harms or damages to end-users or their stakeholders. This may raise questions about the accountability and responsibility for the outputs and their consequences, and the allocation of liability and compensation in case of harms or damages.

Attribution and ownership. Generative AI models can create outputs that are similar or identical to existing human-created artefacts, such as code, images, text, etc. This raises questions about the attribution and ownership of the generated outputs, and the potential for plagiarism, infringement, or misuse of intellectual property [90]. End-user programmers may not be aware of the original sources of the generated code, or the legal and ethical implications of using it for their own purposes.

In response to these challenges, researchers have proposed several complementary solutions [33, 34]:

Ethical design. Generative AI systems should be designed and developed with ethical principles and values in mind, such as fairness, transparency, explainability, privacy, security, and human dignity. Tools should incorporate mechanisms for data protection, output verification, bias mitigation, and user feedback.

Critical use. Generative AI systems should be used critically, and deployed with ethical standards and guidelines in mind, such as professional codes of conduct, industry best practices, and regulatory frameworks. End-user programmers need to be aware of the potential risks and challenges of using such tools.

Governance. Generative AI systems should be subject to ethical oversight and governance by various stakeholders, such as developers, providers, users, regulators, auditors, and ethicists. Tools should be monitored and evaluated by independent third parties for its performance, quality, safety, reliability, and accountability.

7 Implications for End-User Programming Research

We've explored how the generative shift results in a renewed research agenda around the role of formal systems in end-user programming, with a particular focus on control, agency, explanation and debugging, and awareness of the possible.

This contrasts with two of the three main foci of prior generations of EUP research (described in Section 1): improving the ability of EUPs to learn formal systems, and to reduce the expertise requirements of formal systems. What might a research agenda for end-user programming look like in the wake of the generative shift?

Learning to fish. The importance of learning how to code for end-user programmers in a world where generative AI can provide answers is a complex issue that parallels the allegory of teaching a man to fish: “Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime”. While the availability of generative AI systems that can provide direct answers may initially seem to diminish the need for coding skills, the underlying implications reveal a more nuanced perspective. Coding goes beyond the mere act of obtaining answers; it fosters self-sufficiency and cultivates a deeper understanding of the underlying processes and algorithms involved. By learning how to code, end-user programmers gain the ability to create their own solutions, adapt existing ones, and address unique challenges effectively. This empowerment leads to a greater sense of control and creativity, enabling them to solve problems beyond the scope of what AI systems can provide. It has been argued that coding skills facilitate critical thinking, problem-solving, and logical reasoning, which are valuable assets in various domains [121]. While generative AI can offer immediate solutions, relying solely on it can result in dependency and limit the potential for innovation. Therefore, while generative AI presents opportunities for efficiency and convenience, the significance of learning how to code persists, promoting independence, adaptability, and a deeper understanding of computational principles.

But is this really true of end-user programming? Is self-sufficiency really an important optimisation goal, if generative AI is consistently capable of solving end-user tasks? That is, do we still need to teach the man to fish if he can simply receive fish on demand? It is possible that the answer is “no”; as much previous research has shown, in many end-user programming tasks the user may be content to use “incorrect” programs as well as manual methods in order to make progress towards a task [10, 81]. Unlike in a CS education or professional programming setting, the *task comes first*, and any educational or critical thinking outcomes are viewed as secondary. There are also objections to the universalising viewpoints of computational thinking, and evidence that learning to code does not in itself develop computational thinking skills [21].

A related issue has been explored by Potthast et al. [84] in the context of information retrieval: “the dilemma of the direct answer”. The question is to what extent search engine technology is responsible for synthesising information on the web to address the user’s particular query. They identify various trade-offs both on the user side (e.g., the cognitive workload of analysing retrieved documents versus

the accuracy of the answer) and system-side (e.g., the organisation of information for automated processing versus human reading). In our current context, we might think of this as “the dilemma of direct programming”. An emergent research agenda for end-user programming might therefore be to ask what user-side and system-side tradeoffs emerge as a consequence of the generative shift.

Repositioning BERTology. Generative AI-based research must be carefully designed and evaluated in order to make meaningful contributions to the scientific literature. An emerging pattern in research is to simply try a generative model and see what it can do, a kind of research sometimes referred to as “BERTology” [88]. This approach is passive: implicitly positioning EUP researchers outside the development process. Moreover, the results are unlikely to be robust to further iteration and development of generative AI (already the term “BERTology” shows signs of ageing, as its name refers to a family of models that is no longer considered the state of the art). Rather, experimenting with generative AI and developing prompting strategies is a *craft practice* that should be seen as a necessary part of a research investigation, with its own forms of rigour, that feeds into broader research questions with more enduring value. Part of this rigour will involve identifying a clear research question or hypothesis that is independent of the “abilities” of models, the careful selection of a generative model that is well-suited to the problem at hand, and ensuring that the work is reproducible and transparent by detailing the models, heuristics, hyperparameters, and prompt development methodology.

End-user software customisation. A closely related set of concerns to end-user programming is end-user software customisation [66]. This encompasses a wide range of activities such as changing the default settings in an application, writing macros, and editing software source code [75], all with the aim of tailoring software to user needs. End-user customisation is viewed as challenging, with many expertise and motivational barriers, and consequently quite rare; in fact the tendency of people to accept “default” options is a widely documented and deployed phenomenon in behavioural science in domains ranging across organ donation, retirement savings, browser and search engine choices, and wireless encryption [37, 49]. At the same time, researchers have acknowledged the empowering value of end-user software customization, and proposed various potential solutions for increasing the customizability of software [5, 52, 83].

The generative shift may change the landscape of end-user software customization substantially. Not only in reducing the barriers to scripting and automation, but also in the spectrum of potential automations, and the user’s relationship with and attitude to automation and scripting. Petricek theorises that customizable software consists of a set of “substrates” [83], which are programming notations which trade off expertise requirements versus the scope of change they enable. For instance, in Excel, formulas have low expertise

requirements and allow for a low scope of change to Excel functionality, whereas VBA has comparatively higher expertise requirements but enables a greater scope of change. A hypothetically “ideal” substrate can take on a smooth gradient, so that the same programming notation or environment can be used to make extremely small-scope changes with correspondingly low expertise requirements, as well as broad changes with correspondingly higher expertise requirements. Generative AI may provide such a substrate: by mediating between expressions of user intent in natural language, and a wide variety of underlying technical infrastructures. For instance, a natural language query in a spreadsheet might be satisfied by a combination of changed settings, formulas, and macros. Through the unified interface of naturalistic language, the user may be able to smoothly span a broad scope of changes.

But the implications go further: rather than the current model of feature-rich software applications where a number of use cases have been “captured” in code by expert software developers [77], future applications for creating and manipulating information artefacts may leave a large portion of development and customisation for end users. In such a future, what is the nature of the application? Is it a small set of core features that identifies a nucleus of concerns or type of information artefact that is a useful abstraction for knowledge workers, upon which they build? Or does the application disappear as an organisational principle for knowledge work, transitioning into artefact or process-oriented paradigms?

The cognitive dimensions of notations framework [31] offers a vocabulary for evaluating the design tradeoffs made in programming languages. However, they may also be applied to the design decisions in end-user software customisability. The generative shift enables much greater end-user flexibility in this regard, and raises some interesting implications for the cognitive dimensions framework. For instance, the dimension *viscosity* refers to how difficult it is to make small changes to the program. Consider the spreadsheet, an archetypical end-user programming application. The spreadsheet interface may have its own sources of viscosity, but the spreadsheet also has multiple options for scripting and automation, such as the formula language, definition of custom functions, and macro-style scripting. Each of these may bring their own sources of viscosity. Now, if the spreadsheet interface can be significantly customised through scripting, the user may be in a position to identify and ameliorate sources of viscosity in their own workflow. This leads to the user experience of viscosity in the interface being *multifaceted*, and evolving *dynamically* over time, rather than relatively fixed properties of a notation. A related set of concerns has been explored in depth by Jakubovic et al. [40].

Quantifying end-user programmers. In 2005, Scaffidi et al. estimated the number of end-user programmers using US labour statistics and extrapolating from the number of spreadsheet users [97]. This methodology works for as long

as EUP activity is strongly associated with particular applications (such as spreadsheets), but the generative shift will enable EUP activities across many more applications and platforms, facilitating the breakdown of application boundaries [77] and mitigating the pains of transitioning between tools in the “toolbelt” style of computing [91, 107].

Attention investment. Blackwell’s Attention Investment Model explains the behaviour of programmers in deciding whether to automate something (e.g., write a script) or pursue a manual strategy [10]. Just like a cost-benefit analysis, Blackwell’s model posits that a programmer is likely to pursue automation if the (perceived) *payoff*, in terms of attention units saved by automation, minus the *investment* costs of automation, is greater than the *cost* of following a manual strategy, when accounting for the *risk* that the automation may not work. What happens to this model when the cost of automation is reduced to zero, or near zero? It is possible that rather than a trade-off between the costs and benefits of attention investment into automation, other factors will come to dominate the automation decision, such as agency and trust. Or perhaps there will still a cost-benefit tradeoff in terms of attention units, but these units will be spent (and saved) primarily on new categories of EUP activities that do not currently exist.

Learning barriers. Ko et al. defined six learning barriers for end-user programming systems: design, selection, coordination, use, understanding, and information [55]. After the generative shift, will EUPs face the same learning barriers? For instance, the abstraction matching problem identified by Sarkar et al. [94] does not fit neatly into any of these categories. Perhaps new learning barriers will appear, some will disappear, and some barriers will remain but their nature will change.

Self-efficacy. Wiedenbeck et al. found that computer self-efficacy increases substantially during introductory programming courses [119]. They propose that in order to create self-efficacy, students should practice tracing program execution, program comprehension, and manual writing of code. However, as a result of the generative shift, EUPs may gain self-efficacy not from the direct authoring and comprehension of code, but from effectively applying tools to solve their domain problems. Comprehending the underlying logic and structure of the AI-generated code may require different instructional strategies compared to tracing the logic of manually written programs. Rather than focusing on code directly, EUPs might learn how to explore the outputs and performance characteristics of AI-generated code as a way of critically analysing its behaviour and identifying potential limitations. This approach may help EUPs develop a mental model that incorporates an understanding of the strengths and weaknesses of generative AI and how it aligns with their programming goals.

Naming. Liblit et al. draw attention to (identifier) naming as a central conceptualisation mechanism and cognitive

challenge in programming [61]. They find that programmers employ cognitive strategies when selecting and using names, such as following lexical and morphological conventions to convey role information and using metaphors to facilitate productive inferences. Natural language grammars influence name usage, and debates arise around the polysemy of names and their literal versus metaphorical meanings. The authors posit that the linguistic sophistication of a programming language may change or shift the cognitive burdens of programmers. The generative shift introduces an additional layer of linguistic influence in the prompt language used to control the tools as well as the language the tools may emit (by way of code explanation, example outputs, clarification requests, etc.). While generative AI could alleviate cognitive burdens in code development by generating optimised and readable code, as well as providing concise explanations of code logic, it is unclear whether such code and explanations can replicate the linguistic conventions, metaphors, and domain knowledge necessary for managing complexity, and it is uncertain to what extent such considerations remain central to the activity of programming as less code is written and read directly.

Live programming. Tanimoto presented a highly influential framework for evaluating the degree of liveness in programming environments, based on how quickly and continuously they provide feedback to programmers about the execution of their code [110]. After the generative shift, live systems might need to account for different forms and sources of feedback, besides execution feedback. For example, generative AI could provide feedback to programmers about the quality, readability, or maintainability of their code, or suggest alternative or improved ways of writing their code [39]. Moreover, generative AI could enable programmers to use natural language, gestures, or voice commands to express their intentions or goals, and the programming environment could generate or modify code accordingly. This could change the notion of liveness from being based on code editing and execution, to being based on goal specification and realization. Alternatively, generative AI could enable programmers to delegate some or all of their programming tasks to intelligent agents, and monitor or intervene in their work as needed. This could change the notion of liveness from being based on direct manipulation and control, to being based on supervision and guidance [111].

8 Conclusion

This essay has considered how generative AI might change the landscape of end-user programming and end-user programming research. The research agenda for end-user programming has so far focused on helping people learn and use formal systems (e.g., a programming language). However, generative AI stands to facilitate an intensification and extensification of end-user programming activities across

many more applications and tasks, and users may interact with generative AI primarily through informal systems of natural language. This is the *generative shift hypothesis*.

Under the generative shift hypothesis, we have discussed how several strengths of formal systems, including control, agency, explanation, debugging, trust, may become less relevant or change in nature. We propose that EUP research may need to shift its traditional focus on formal systems to new concerns of the practical use of generative AI.

The key takeaways are:

- The generative shift moves the focus of end-user programming research from improving the learnability and expertise requirements of formal systems, to new issues of control, agency, explanation, debugging, and the awareness of the possible.
- The generative shift affects many core theories and concepts of end-user programming, such as end-user software customisation, the attention investment model, learning barriers, self-efficacy, and live programming. These may need to be revisited and revised as generative AI enables EUP to proceed in new ways and scales.

Petricek encourages us to question the fundamental assumptions of programming languages research and theory [82]. The generative shift is an opportunity to propel EUP research forward, renewing and revitalising the importance of human-centric approaches to instructing computers, rather than unprogrammable AI tools that ultimately take freedom away from the user. As Blackwell calls for in *Moral Codes* [9]: “*If computer users have access to appropriate notations - Moral Codes - they can use simple automation to make their lives less mechanical, rather than more. If computer interfaces are designed as notational spaces, they offer freedom and negotiation, even forms of social organisation, complex assemblies of intelligent decision making and deliberation, respecting the humans creating them, rather than pretending humans were not involved.*”

References

- [1] Saeed Aghaee, Alan F Blackwell, David Stillwell, and Michal Kosinski. 2015. Personality and intrinsic motivational factors in end-user programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 29–36.
- [2] Blake Alcott. 2005. Jevons’ paradox. *Ecological economics* 54, 1 (2005), 9–21.
- [3] Ian Arawjo. 2020. To write code: The cultural fabrication of programming notation and practice. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–15.
- [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 preprint arXiv:2108.07732* (2021).
- [5] Antranig Basman, Luke Church, Clemens Nylandstedt Klokmoose, and Colin BD Clark. 2016. Software and How it Lives On-Embedding Live Programs in the World Around Them.. In *PPIG*. 19.

- [6] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.
- [7] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. 2006. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 231–240.
- [8] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big?. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*. 610–623.
- [9] Alan Blackwell. 2022. *Chapter 13: Conclusion*. MIT Press. <https://moralcodes.pubpub.org/pub/chapter-13>.
- [10] Alan F Blackwell. 2002. First steps in programming: A rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE, 2–10.
- [11] Alan F Blackwell. 2002. What is programming?. In *PPIG*. Citeseer, 20.
- [12] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).
- [13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [14] 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.
- [15] Li Cai. 2012. Latent variable modeling. *Shanghai Arch. Psychiatry* 24, 2 (April 2012), 118–120.
- [16] George Chalhoub and Advait Sarkar. 2022. "It's Freedom to Put Things Where My Mind Wants": Understanding and Improving the User Experience of Structuring Data in Spreadsheets. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–24.
- [17] Nick Collins, Alex McLean, Julian Rohrerhuber, and Adrian Ward. 2003. Live coding in laptop performance. *Organised sound* 8, 3 (2003), 321–330.
- [18] David Coyle, James Moore, Per Ola Kristensson, Paul Fletcher, and Alan Blackwell. 2012. I did that! Measuring users' experience of agency in their own actions. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 2025–2034.
- [19] Allen Cypher. 1991. Eager: Programming repetitive tasks by example. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 33–39.
- [20] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.
- [21] Peter J Denning. 2017. Remaining trouble spots with computational thinking. *Commun. ACM* 60, 6 (2017), 33–39.
- [22] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.
- [23] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing Education in the Era of Generative AI. *arXiv preprint arXiv:2306.02608* (2023).
- [24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [25] Umberto Eco. 1997. *The search for the perfect language*. Blackwell, London, England.
- [26] B. S. Everitt. 1984. *An Introduction to Latent Variable Models* (1 ed.). Springer Dordrecht. VIII, 108 pages. <https://doi.org/10.1007/978-94-009-5564-6>
- [27] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. *arXiv preprint arXiv:2205.10583* (2022).
- [28] Sally Fincher and Marian Petre. 2004. *Computer science education research*. CRC Press.
- [29] Michel Foucault. 1966. *Les mots et les choses: une archéologie des sciences humaines*. Editions Gallimard.
- [30] Maria I Gorinova, Advait Sarkar, Alan F Blackwell, and Don Syme. 2016. A live, multiple-representation probabilistic programming environment for novices. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 2533–2537.
- [31] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.
- [32] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
- [33] Andrei Hagiú and Julia Wright. 2020. *A Practical Guide to Building Ethical AI*. <https://hbr.org/2020/10/a-practical-guide-to-building-ethical-ai>
- [34] Stefan Harrer. 2023. Attention is not all you need: the complicated case of ethically using large language models in healthcare and medicine. *EBioMedicine* 90 (2023).
- [35] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. 2011. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceedings of the 33rd International Conference on Software Engineering*. 451–460.
- [36] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2015. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering* 20 (2015), 549–575.
- [37] JON M. JACHIMOWICZ, SHANNON DUNCAN, ELKE U. WEBER, and ERIC J. JOHNSON. 2019. When and why defaults influence decisions: a meta-analysis of default effects. *Behavioural Public Policy* 3, 2 (2019), 159–186. <https://doi.org/10.1017/bpp.2018.43>
- [38] Gonzalo Jaimovitch-López, César Ferri, José Hernández-Orallo, Fernando Martínez-Plumed, and María José Ramírez-Quintana. 2022. Can language models automate data wrangling? *Machine Learning* (2022), 1–30.
- [39] Krish Jain and Steven L Tanimoto. 2020. Integrating a Live Programming Role into Games.. In *PPIG*.
- [40] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *arXiv preprint arXiv:2302.10003* (2023).
- [41] Will Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Caulty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Amy Ko. 2015. A principled evaluation for a principled Idea Garden. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 235–243.
- [42] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv preprint arXiv:2306.02907* (2023).
- [43] Nima Joharizadeh, Advait Sarkar, Andrew D Gordon, and Jack Williams. 2020. Gridlets: Reusing spreadsheet grids. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

- [44] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *arXiv preprint arXiv:2304.02195* (2023).
- [45] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).
- [46] Daniel Martin Katz, Michael James Bommarito, Shang Gao, and Pablo Arredondo. 2023. Gpt-4 passes the bar exam. *Available at SSRN 4389233* (2023).
- [47] Henry Kautz. 2022. The third AI summer: AAAI Robert S. Englemore memorial lecture. *AI Magazine* 43, 1 (2022), 105–125.
- [48] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.
- [49] Jay P Kesan and Rajiv C Shah. 2006. Setting software defaults: Perspectives from law, computer science and behavioral economics. *Notre Dame L. Rev.* 82 (2006), 583.
- [50] Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avriila Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, et al. 2023. From Words to Code: Harnessing Data for Program Synthesis from Natural Language. *arXiv preprint arXiv:2305.01598* (2023).
- [51] Aleksandra Klačnja-Milićević, Boban Vesin, Mirjana Ivanović, and Zoran Budimac. 2011. Integration of recommendations and adaptive hypermedia into Java tutoring system. *Computer Science and Information Systems* 8, 1 (2011), 211–224.
- [52] Clemens N Klokose, James R Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 280–290.
- [53] 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 (CSUR)* 43, 3 (2011), 1–44.
- [54] 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*. 151–158.
- [55] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.
- [56] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.
- [57] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th international conference on intelligent user interfaces*. 126–137.
- [58] Julia A Lasserre, Christopher M Bishop, and Thomas P Minka. 2006. Principled hybrids of generative and discriminative models. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 1. IEEE, 87–94.
- [59] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. Tweakit: Supporting end-user programmers who transmogrify code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.
- [60] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
- [61] Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive Perspectives on the Role of Naming in Computer Programs.. In *PPiG*. 11.
- [62] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.
- [63] Hannah Limerick, David Coyle, and James W. Moore. 2014. The experience of agency in human-computer interactions: a review. *Frontiers in Human Neuroscience* 8 (2014). <https://doi.org/10.3389/fnhum.2014.00643>
- [64] 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 (Hamburg, Germany) (CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. <https://doi.org/10.1145/3544548.3580817>
- [65] Vivian Liu and Lydia B Chilton. 2022. Design guidelines for prompt engineering text-to-image generative models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–23.
- [66] Wendy E Mackay. 1991. Triggers and barriers to customizing software. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 153–160.
- [67] Adrian Mackenzie. 2017. *Machine learners: Archaeology of a data practice*. MIT Press.
- [68] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 931–937.
- [69] Stephen MacNeil, Andrew Tran, Juho Leinonen, Paul Denny, Joanne Kim, Arto Hellas, Seth Bernstein, and Sami Sarsa. 2022. Automatically Generating CS Learning Materials with Large Language Models. *arXiv preprint arXiv:2212.05113* (2022).
- [70] Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (1971), 151–165.
- [71] David Metz. 2021. Economic benefits of road widening: Discrepancy between outturn and forecast. *Transportation research part A: policy and practice* 147 (2021), 312–319.
- [72] Donna J Middaugh. 2018. Delusion of Control: Pushing Buttons. *Medsurg Nursing* 27, 6 (2018), 399.
- [73] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence* 267 (2019), 1–38.
- [74] Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hananeh Hajishirzi. 2021. Reframing Instructional Prompts to GPTk’s Language. *arXiv preprint arXiv:2109.07830* (2021).
- [75] Anders Mørch. 1997. Three levels of end-user tailoring: Customization, integration, and extension. *Computers and design in context* 1997 (1997), 61.
- [76] Stephan Morgenthaler. 2009. Exploratory data analysis. *Wiley Interdisciplinary Reviews: Computational Statistics* 1, 1 (2009), 33–44.
- [77] Midas Nouwens and Clemens Nylandstedt Klokose. 2018. The application and its consequences for non-standard knowledge work. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.
- [78] Arika Okrent. 2009. *In the land of invented languages: Esperanto rock stars, Klingon poets, Loglan lovers, and the mad dreamers who tried to build a perfect language*. Random House.
- [79] OpenAI. 2023. GPT-4 Technical Report. [arXiv:2303.08774](https://arxiv.org/abs/2303.08774) [cs.CL]
- [80] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.
- [81] Rahul Pandita, Chris Parnin, Felienne Hermans, and Emerson Murphy-Hill. 2018. No half-measures: A study of manual and tool-assisted end-user programming tasks in Excel. In *2018 IEEE Symposium*

- on visual languages and human-centric computing (VL/HCC). IEEE, 95–103.
- [82] Tomas Petricek. 2016. Programming language theory: Thinking the unthinkable (Work in progress). In *Proceedings of the Annual Conference of the Psychology of Programming Interest Group (PPIG)*.
- [83] Tomas Petricek. 2022. No-code, no thought? substrates for simple programming for all. <https://tomasp.net/blog/2022/no-code-substrates/>
- [84] Martin Potthast, Matthias Hagen, and Benno Stein. 2021. The dilemma of the direct answer. In *Acm sigir forum*, Vol. 54. ACM New York, NY, USA, 1–12.
- [85] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. In *International Conference on Machine Learning*. PMLR, 8821–8831.
- [86] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
- [87] Michael Revow, Christopher KI Williams, and Geoffrey E Hinton. 1996. Using generative models for handwritten digit recognition. *IEEE transactions on pattern analysis and machine intelligence* 18, 6 (1996), 592–606.
- [88] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A primer in BERTology: What we know about how BERT works. *Transactions of the Association for Computational Linguistics* 8 (2021), 842–866.
- [89] Advait Sarkar. 2022. Is explainable AI a race against model complexity?. In *Workshop on Transparency and Explanations in Smart Systems (TeXSS), in conjunction with ACM Intelligent User Interfaces (IUI 2022) (CEUR Workshop Proceedings, 3124)*, 192–199. <http://ceur-ws.org/Vol-3124/paper22.pdf>
- [90] Advait Sarkar. 2023. Exploring Perspectives on the Impact of Artificial Intelligence on the Creativity of Knowledge Work: Beyond Mechanised Plagiarism and Stochastic Parrots. In *Annual Symposium on Human-Computer Interaction for Work 2023 (CHIWORK 2023)*. ACM, Oldenburg, Germany, 17. <https://doi.org/10.1145/3596671.3597650>
- [91] Advait Sarkar. 2023. Should Computers Be Easy To Use? Questioning the Doctrine of Simplicity in User Interface Design. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (Hamburg, Germany) (CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 419, 10 pages. <https://doi.org/10.1145/3544549.3582741>
- [92] Advait Sarkar and Andrew D. Gordon. 2018. How do people learn to use spreadsheets? (Work in progress). In *Proceedings of the 29th Annual Conference of the Psychology of Programming Interest Group (PPIG 2018)*, 28–35.
- [93] Advait Sarkar and Andrew D Gordon. 2018. How do people learn to use spreadsheets?(Work in progress).. In *PPIG*.
- [94] 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?. In *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022)*.
- [95] Advait Sarkar, Sruti Srinivasa Ragavan, Jack Williams, and Andrew D Gordon. 2022. End-user encounters with lambda abstraction in spreadsheets: Apollo's bow or Achilles' heel?. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–11.
- [96] Advait Sarkar, Martin Spott, Alan F Blackwell, and Mateja Jamnik. 2016. Visual discovery and model-driven explanation of time series patterns. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 78–86.
- [97] Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 207–214.
- [98] Floarea Serban, Joaquin Vanschoren, Jörg-Uwe Kietz, and Abraham Bernstein. 2013. A survey of intelligent assistants for data analysis. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 1–35.
- [99] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
- [100] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.
- [101] Nikhil Singh, Guillermo Bernal, Daria Savchenko, and Elena L Glassman. 2022. Where to hide a stolen elephant: Leaps in creative writing with multimodal machine intelligence. *ACM Transactions on Computer-Human Interaction* (2022).
- [102] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askill, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203* (2019).
- [103] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3509–3521.
- [104] Sruti Srinivasa Ragavan, Advait Sarkar, and Andrew D Gordon. 2021. Spreadsheet comprehension: Guesswork, giving up and going back to the author. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–21.
- [105] Alistair Stead and Alan F Blackwell. 2014. Learning syntax as notational expertise when using drawbridge. In *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014)*. 41–52.
- [106] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M Rush. 2022. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. *IEEE transactions on visualization and computer graphics* 29, 1 (2022), 1146–1156.
- [107] Tamara Sumner and Markus Stolze. 1997. *Evolution, not revolution: Participatory design in the toolbelt era*. MIT Press Cambridge, MA.
- [108] Bruno Tafur and Advait Sarkar. 2023. User Perceptions of Automatic Fake News Detection: Can Algorithms Fight Online Misinformation? [arXiv:2304.07926](https://arxiv.org/abs/2304.07926) [cs.HC]
- [109] Alex Tamkin, Miles Brundage, Jack Clark, and Deep Ganguli. 2021. Understanding the capabilities, limitations, and societal impact of large language models. *arXiv preprint arXiv:2102.02503* (2021).
- [110] Steven L Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, 31–34.
- [111] Steven L Tanimoto. 2020. Multiagent live programming systems: Models and prospects for critical applications. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. 90–96.
- [112] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [113] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

- [114] Pablo Villalobos, Jaime Sevilla, Lennart Heim, Tamay Besiroglu, Marius Hobbhahn, and Anson Ho. 2022. Will we run out of data? An analysis of the limits of scaling datasets in Machine Learning. *arXiv preprint arXiv:2211.04325* (2022).
- [115] David Vos, Till Döhmen, and Sebastian Schelter. 2022. Towards Parameter-Efficient Automation of Data Wrangling Tasks with Prefix-Tuning. In *NeurIPS 2022 First Table Representation Workshop*.
- [116] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [117] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, et al. 2021. Ethical and social risks of harm from language models. *arXiv preprint arXiv:2112.04359* (2021).
- [118] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).
- [119] Susan Wiedenbeck, Deborah Labelle, and Vennila NR Kain. 2004. Factors affecting course outcomes in introductory programming.. In *PPIG*. 11.
- [120] Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. 2003. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 305–312.
- [121] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.
- [122] Nuyun Zhang, Gang Huang, Ying Zhang, Ning Jiang, and Hong Mei. 2010. Towards Automated Synthesis of Executable Eclipse Tutorials.. In *SEKE*. 591–598.

Received 2023-04-28; accepted 2023-08-11