Back to the future: What do historical perspectives on programming environments tell us about LLMs?

Tao Dong
Google LLC

taodong@google.com

Luke Church
Computer Laboratory
Cambridge University
luke@church.name

Abstract

Many people, regardless of whether they consider themselves programmers, are interested in the ability of large language models (LLMs) to assist with programming tasks. The rapid rise of LLMs is transforming the socio-technical constraints and incentives under which programming environments are designed and adapted. This change is affecting languages, APIs, tools, and documentation. For instance, the practice of example-centric programming is being enhanced by LLMs, resulting in a shift of the primary programming activity from creating to curating. It may seem like this is all very new, but we see a resurgence of age-old themes in the psychology of programming and developer tools design.

Re-examining these historical themes in light of the growing popularity of LLM-assisted programming leads to open questions about how future programming environments will be designed. In particular, we argue that the implications of LLMs will go far beyond adding features to code editors. Design choices in programming languages and frameworks, which have so far been largely unaffected by LLMs, will be made differently due to changing programmer behaviors and preferences enabled and amplified by LLMs.

1. Introduction

Every few years a technology comes along that gives the feeling of changing everything, creating as venture capitalists might like to call it a ‘disruption’ to society. Looking back over recent years, these have included the invention of the Internet, development of search engines, the adoption of smartphones, all of which have had a profound impact on programming. Other changes seem to have a less enduring effect or a mixed outlook, such as mashups, the semantic web, cryptocurrencies, and the metaverse.

Inside the moment of any of the revolutions it can be difficult to know which way is up. They are often accompanied with a great deal of noise from techno-evangelists proclaiming a glorious future, moral panics of people warning about threats to the end of society and human morality, and techno-apathists saying that there will be no observable effect, we’ve been here before. All three of these groups have proven to be systematically incorrect. Amongst all of this noise, there is a practical need: To understand how these technical advances can be integrated into the almost mundane day to day work of software engineering, like plumbers, unglamorous but very necessary to current society.

Large Language Models are the latest of such technologies, simultaneously hailed as the future and the end of our future. In this paper we’ll leave these grand claims to others and focus on their likely impact on the making of programming environments, including programming language, APIs, tools, and documentation.

So far, the impact of LLMs on programming has been predominantly felt and embraced by IDE developers through the integration of multi-token code completion[1] and AI chat bots[2] in code editors. Their enthusiasm towards LLMs is, however, in stark contrast to the relatively muted reaction to LLMs from technical communities which design and build the foundational layers of a programming environment, including languages, frameworks, compilers, language servers, etc. The IDE might be the primary site of application for LLMs, but will their impact on the design of programming environments stop there?

A study of the history of programming language design teaches us that new capabilities in IDEs and the user behavior and expectations cultivated by such capabilities could have profound implications for the design of the rest of the programming environment. For example, static typing gained widespread acceptance partially because many programmers have developed an appreciation of, if not dependence on, accurate and fast code completion. In the past decade, we've seen static type systems get retrofitted to previously dynamic languages such as JavaScript (through TypeScript[3]) and Python (through Mojo[4]) at least partially for this purpose.

Given the co-evolutive nature of programming environments and programer behavior, will the proliferation of LLM use among programmers lead to a profound impact on the design of languages, frameworks, and other foundational building blocks of programming? How will these impacts affect design choices and decision making? We explore those questions by first revisiting age-old themes of programer behavior and experiences. We then speculate on how those phenomena could be amplified or changed by the adoption of LLMs. Finally, we discuss the implications of changing behaviors for the design of programming environments. It's too early in our understanding of this technology to really have answers; this paper is more an invitation to a conversation than a proposed way forward.

2. Recurring themes about programming in the age of LLMs

In this section, we revisit some long-running themes of human factors in programming and discuss how LLMs shed a new light on them. These are not in any particular order, or have any claim to completeness but rather act as prompts for our readers to think of other pre-existing themes and the new relevance they might have.

2.1. End-user development

End-User Development (EUD) refers to the phenomenon of users, often professionals with domain knowledge, writing programs for themselves to aid their work or everyday tasks (Nardi, 1993). This is usually in contrast to traditional Software Engineering where the programs are written for others, often as products. There have been many computational and notational paradigms EUD including:

- Spreadsheets
- Block-based programming (e.g., Scratch, VEXcode)
- Interface builders
- Diagram-based programming (e.g., Yahoo Pipes)
- Programming by examples
- Scripting (e.g., VBA, AppleScript, SQL queries, etc)

[1] https://github.com/features/copilot
[2] https://developer.android.com/studio/preview/studio-bot
[3] https://www.typescriptlang.org/
[4] https://www.modular.com/mojo

The psychological approach that many of these approaches take to make programming more accessible is turning recall tasks to recognition tasks. For example, block-based programming visualizes grammatical rules through different shapes of "sockets" and "plugs." And the success of interface builders owes to their ability to remove the need to recall code corresponding to visual layout and interactive elements.

Though EUD tools lowered the barriers to programming, they're often considered having low ceilings—easy to get started with but hard to achieve advanced results or fine-grained control. As Myers et al. (2000) commented,

> *"The most successful current systems seem to be either low threshold and low ceiling, or high threshold and high ceiling."*

LLMs offer the potential of enabling the psychological design maneuver of turning recognition into recall without the attendant limits to the ceiling of what can be achieved. This might be done through their surprising capability to generate working code based on natural language descriptions. The primary task in programming is shifting towards recognizing whether a piece of LLM-generated code can satisfy the requirements and identifying chunks of code that require further customization and refinement.

The implications for EUD can be profound. First, the threshold of previously high-threshold, high-ceiling tools are coming down, making low-threshold, low-ceiling tools on the market less attractive. A lowered threshold can help more end-users gain the confidence and desire to write programs, motivating more software systems to offer EUD support, either via a natural language style interaction, a general purpose language whose interaction is intermediated through an LLM, or a mixture of the two.

Second, one of the perceived utility of visual programming tools might need to be re-evaluated, since a key benefit they provide—turning recall tasks into recognition tasks—may be obtainable through conversing with LLMs in a natural language. Even before the introduction of LLMs this trend was already emerging, for example, with declarative UI programming techniques making UI builders less popular. The adoption of LLMs might further decrease the utility of UI builders in their current form. With the introduction of multimodal LLMs with capabilities to generate code based on visual input (e.g., UI sketches), the design of UI builders must evolve to complement users' multimodal interactions with LLMs.

2.2. Example-centric programming and opportunistic programming

Example-centric programming (ECP) describes a pragmatic approach towards writing code by retrieving and modifying examples found on the Internet (Brandt et al., 2010). The approach was popularized by the widespread use of online Q&A forums for programming such as StackOverflow and open-source code hosting platforms such as SourceForge and eventually Github. This approach is closely related to opportunistic programming (Brandt et al., 2009), a term used to describe how online information foraging, just-in-time learning, and sometimes coding without understanding are practiced in the production of computer programs.

As widespread as ECP has become, researchers have noted that doing so is not without its challenges. Systems have been designed and implemented to address two common challenges when practicing ECP: 1) constructing effective queries are difficult, especially when specific attributes of user's programming environments need to be considered; and 2) sifting through search results for relevant code is tedious and slow. In response, some ECP systems embed a custom search facility in the IDE to help the user construct search queries and clean up results (Brandt et al., 2010).

LLMs significantly lowered these two barriers to practicing ECP through code completion and generation. Prompts—equivalent to queries in previous systems—are constructed with natural language and snippets of code the user has written and additional context supplied by the IDE. In some cases an entire codebase can be included as contextual information in the prompt.[5] The output of those LLMs are well-formatted code that can be iteratively adjusted through tweaking the prompt. Early research shows that using LLM-based assistance could boost productivity by as much as 55.8 percent in select tasks (Peng et al., 2023). As a result, ECP is likely to become even more popular among programmers, especially when learning a new programming language or using an unfamiliar library. We can expect more code to be written opportunistically based on examples suggested by an LLM.

The full understanding of the task being performed of selecting the relevant material out of the example, the equivalent of information foraging within an LLM, is not well characterized. It is also not currently clear how other problems with ECP, such as the associated security challenges, might carry across to code generated via LLMs, and whether the tools used for managing emergent properties of statistical tools such as 'de-biasing' could be effectively applied to address them.

2.3. Live programming

Traditional programming systems require an explicit interaction between changing the content of a program and observing the effects of its execution. Technical steps such as compilation often take place after the user requests it. This adds up to creating an episodic user experience where the user makes a change and then waits to see what the effect of that change is.

Live programming provides an alternative to this. Initially characterized by Tanimoto into 4 levels (Tanimoto, 1990), at the higher levels it describes an experience where parts of the interface and associated notations react rapidly after the user has made a change to the program.

Live programming interaction styles are especially beneficial in some interaction domains. A genre of music has emerged around live coding, where the execution of the program creates the sound effect, and the program is often modified live on stage, often with the audience able to see the changes whilst they dance in an event known as an algorave (Blackwell & Cocker, 2022).

Interactive creation of music is not the only application domain in which liveness is a benefit. Liveness has been applied to game design (Kato & Shimakage, 2020), spreadsheets, data analysis (Church, Marasoiu, et al., 2016), and user interface engineering in general purpose programming languages[6][7] where it sometimes goes by the name 'hot reload'.

There are many challenges with using live systems in production environments, these include the relative instability of the behavior of the systems being created, so as you make small changes the program flickers, to problems that most modern programming languages try to exclude partial or incomplete programs, but during development most programs are 'illegal' most of the time. What the right behavior in these intermediary stages is is a domain specific problem, hence we have not seen many general purpose live programming environments, but rather separate ones for music, for data and for UIs.

[5] https://github.com/mpoon/gpt-repository-loader
[6] https://docs.flutter.dev/tools/hot-reload
[7] https://gaearon.github.io/react-hot-loader/getstarted/

In previous work (Church, Söderberg, et al., 2016) we speculated on how some of the interaction problems with Liveness could be addressed using program synthesis techniques, both having the programming language infrastructure ‘fix up’ the program to the most probable outcome that the user is ‘likely to have meant’, this could be extended to large areas of functional synthesis where significant components are automatically generated.

The primary effect that LLMs have here is to make the synthesis computationally possible and perhaps even plausible. As this means that the task of creating the code to run becomes easier, the focus will shift on to interacting with and understanding the behavior of the program, including how to provide safe environments where the behavior of speculative programs can be understood without damaging side effects.

2.4. Path of least resistance

Path of least resistance is a notion introduced by Myers et al. (2000) in their paper *Past, Present, and Future of User Interface Software Tools*. The main insight is that properties of tools can influence the kinds of programs more likely to be written through making it easier to achieve the desirable outcomes. In UI programming, one of the major advancements in the past decade is the rise of design systems (Churchill, 2019), which make well-designed UI components readily available through UI frameworks and ensure UI designers and developers communicate with the same UI primitives. Design systems made it much easier to create UIs with adequate usability and aesthetics without direct support from designers.

Nonetheless, it remains a challenge to encourage programmers to adapt standard UI components to accommodate users with special needs, international audiences, and multiple types of devices. Model-based UI development was proposed as a potential solution to this challenge. The general concept is that the programmer specifies the UI as an abstract model of user tasks and then has the system create concrete user interfaces based on attributes of the user and the computing environment (Gajos & Weld, 2004; Miñón et al., 2016). However, this model-based approach has yet to gain traction due to the difficulty of constructing models in the first place.

LLM provides a new possibility to offer paths of least resistance in programming through its power to transform code, both newly generated examples and the user’s existing code, through prompt engineering.

2.5. Student and Professional Programmers

As we are suggesting above, the adoption of LLMs within programming is likely to have an effect in a number of different ways depending on what activity is being performed and for what purpose. Programmers are in general not homogeneous, so we wouldn’t expect them all to be affected in the same way. We have already suggested above that end-user developers may have a different experience of LLM-mediated programming to professionals. Similarly the effect of LLMs may be different to students than it is to professionals, reflecting their differing motivations and practices.

For example, we’ve already suggested a number of times above that as program synthesis becomes easier more of the work of programming will become rapid comprehension and curation as opposed to authoring. However, this presents a challenge to the underlying educational philosophy of software engineering which is often dominated by constructionist perspectives—that the act of building something, especially out of logical components such as program statements, provides a strong form of knowledge. Instead, LLMs could shift the intellectual approach in software engineering education to one better characterized as “reverse engineering,”

"discovery," and "evaluation," often more associated with scientific disciplines than engineering ones. Perhaps this brings the working practice of the student, closer to the pragmatic approaches often adopted by industry professionals[8].

LLMs may also affect the aspirations and motivations of both individual students and corporate actors. If there is little to be gained from a systematic study of programming, rather than an opportunistic one, will students still be interested in studying computer science? And if LLMs increase the cross-language mobility of code, will the corporate sponsors of programming languages still be interested in getting their languages into classrooms?

3. Implications for building programming environments

As has happened with previous "disruptions," programming environments will respond to new evolutionary pressures created by the adoption of LLMs as a programming aid. At first this is likely to occur with local adaptations in the places that are easiest to evolve—often the code editors—but we predict these pressures will start to shape the rest of the technology stack a programmer needs to interact with as well.

3.1. Notational Evolution

Previous work in the study of the evolution of notations might give us some pointers as to how these changes might take place. Green and Fetais (2016) and others (Green and Church, forthcoming) have pointed out that notations change along predictable patterns. LLMs within programming can currently be seen as very large "helper devices" (Green & Petre, 1996)—that is external components that sit off to the side of the main notation peripherally interacting with it.

There are a number of other such tools in the evolution of programming environments, some, such as Stack Overflow, have remained largely independent of the editing context, but even so have had a significant influence on development practice and on the design of an API or an IDE. Others such as static analysis tools have tended to be folded into the environment directly, and are now an expected part of programming support.

In both these cases the tools and resources then co-evolve with the programming language and the APIs. The most notable example of this is the introduction of code completion into Visual Studio, where the designers of the C# language explicitly stated that it was designed to be written in IDEs with this feature. This supports the long running argument of Cognitive Dimensions that notations and the environments where notations are used need to be considered as a joint design exercise.

Whilst it's difficult to predict the course of evolution that LLMs will take, it seems likely that it will subsequently influence the design of future tooling and notations, either via integration or independent evolution. In this section, we consider some concrete examples of how this evolution might take place.

3.1. Evolving a programming language or framework

Introducing a new syntax to a programming language or new APIs to a framework often carries risks of increased complexity, insufficient adoption, and higher maintenance cost. Such risks are even higher when the new syntaxes and APIs are not backwards compatible. *Will the benefit of the new feature outweigh its risks and costs?* That's a question programming language and API designers often need to consider. For every feature that sees the light of the day, many others are buried after tradeoffs are evaluated.

[8] https://blog.codinghorror.com/mort-elvis-einstein-and-you/

How will LLMs influence the outcome of evaluating those tradeoffs? One potential consequence is that it could become harder to introduce programmers to the latest language or API changes, once they form a habit of asking LLMs rather than a search engine for example code. As mentioned earlier, LLMs enable example-centric programming at an unprecedented level of efficiency. However, LLMs are biased against, if not completely ignorant of, new changes to a programming language or framework. Without mitigations, users' growing reliance on LLMs in coding can slow down the adoption of new features and prolong the life of deprecated APIs. This will in turn demotivate new language and API changes, especially when the changes in question are to improve user experience rather than filling a mission-critical gap such as fixing a security vulnerability.

The provider of LLM-based coding assistance can potentially mitigate the problem through prompt engineering and retrieval augmented generation.[9] One possibility is injecting information about new features into the user's prompt behind the scene, once the user's original prompt matches the programming language or API in question. For example, a library author reported that by including short usage examples for newly introduced features as part of a prompt, ChatGPT was able to generate code utilizing those new features correctly.[10] Nonetheless, both how effective and how reliable this technique can be remain open questions.

Another possibility is to instruct the LLM to acknowledge what information it doesn't have access to and direct the user to alternate methods of programming assistance. This requires the LLM to disclose its data cutoff date and have the ability to estimate how likely the user's prompt requires more recent information.

It's worth pointing out both types of mitigations require coordination between LLM assistance providers and vendors of programming languages and frameworks. Thus, LLM service providers could become an intermediary between programmers and the languages and APIs they use in a way more prominent than web searches have been. Maintaining a good relationship with major LLM assistance providers might become vital to a programming language's ability to evolve its features without disrupting its users.

3.2. Encouraging good programming practices

Programming language and framework designers want their users to use their products in the right way. Some common concerns include writing tests, supporting accessibility, avoiding security pitfalls, making code readable and well-documented. Making those tasks easier to carry out is a powerful way to get programmers to do them more often, because it's natural for people to gravitate towards "the path of least resistance." As Myers et al. (2000) pointed out, *"Our tools have a profound effect on how we think about problems. It will remain imperative that our tools make 'doing the right thing' easy (in preference to 'doing the wrong thing')."*

Before LLMs became part of programming support, lowering such resistance against good practices was often achieved through scaffolding, verification, and automation. First, scaffolding includes resources such as templates, example code, and saved snippets. They provide reminders and partially implemented solutions for the programmers to follow and build on. Second, verification involves tools such as lints, accessibility checkers, and vulnerability scanners. They ensure flaws are discovered sooner rather than later. Last, automation is often applied to routine tasks such as running tests, generating API docs from source code and comments, and upgrading dependencies.

[9] https://www.pinecone.io/learn/retrieval-augmented-generation/
[10] https://www.reddit.com/r/ChatGPT/comments/12xzcpj/teach_new_apis_to_chatgpt_in_one_prompt/

LLMs have the potential to augment all these three approaches. To begin with, LLMs can personalize templates and examples used to scaffold good programming practices based on the project context either provided by the user explicitly or the user's IDE implicitly. Next, LLMs have shown an ability to check whether a program follows specific requirements. For example, it's been documented that ChatGPT can detect accessibility issues in HTML code and make suggestions to fix them.[11] The interactions between traditional lints and LLM-driven checks will be especially interesting to explore, as LLMs are being more deeply integrated into code editors. Finally, LLMs can potentially automate more tasks such as generating inline comments[12] and enumerating test cases[13].

Moreover, LLMs could incentivize better programming practices, because they generate more useful responses when the user includes clearly-written and well-documented code in the prompts. For example, Jones (2023) demonstrated that using descriptive variable and function names and including inline comments improved ChatGPT's ability to reason about the results of executing a program.

3.3. Designing end-user development support

LLMs are likely to fuel the growth of end-user development tools both as standalone software and as extension mechanisms for existing systems. As mentioned earlier, this growth will be driven by two factors. First, LLMs can lower the barriers for end-users to leveraging existing programming interfaces through code generation from natural languages, and hence making such tools no longer reserved to power users. Second, as more users taste the power of customizing their software, they will demand and pay for EUD support in more types of software systems than what's available today. This creates economic incentives for software developers to invest in EUD support.

When designing an EUD tool, one of the central questions is: *how much code will the user have to deal with?* Based on the answer to this question, EUD tools are often classified as no-code or low-code tools. LLMs are likely to make the no-code approach less appealing, as more users become accustomed to consuming LLM-generated code.

LLMs can influence the evolution of low-code tools as well, in particular, how they leverage visual programming techniques. On the one hand, LLMs are likely to make visual programming less necessary for end-users to create programs. On the other hand, LLMs could make it more important to visualize a program—both its structure and its output—so the end-user developer can quickly verify the program's behavior. Related, liveness of the EUD tool will be highly valuable due to the shift of cognitive work from program authoring to program verification.

Another fundamental question LLMs could influence is whether the EUD tool should provide a domain-specific language (DSL) or adopt a popular general-purpose language. A niche DSL, while often simpler in its syntax and higher in its abstraction, can make it harder to leverage LLMs' code generation capabilities than general-purpose languages in wide-spread usage. The tradeoff could gradually tip in favor of the latter as LLMs improve.

[11] https://intopia.digital/articles/using-chatgpt-to-make-chatgpts-experience-more-accessible/
[12] https://tecadmin.net/add-comments-and-refactor-your-code-with-chatgpt/
[13] https://research.aimultiple.com/chatgpt-test-automation/

3.4. Introduction of new programming languages

Up to this point we've mainly discussed the trade-off space around how LLMs, used as a helper device, influence existing ecosystems and working practices. However, new languages will be designed after the introduction of LLMs, and just as C# and the .NET libraries were influenced by the presence of code completion; we can expect any such new language to be influenced by the existence of LLMs.

Experiments in language production are rare and expensive, as are LLMs themselves. It will take some time before the way in which LLMs influence language design becomes clear, but we can speculate around how such a technology might affect when it makes sense to invent a new programming language or to adapt existing ones.

On the one hand, LLMs could make it more costly to create new programming languages in the future. LLMs rely on large corpuses of existing code to be available to train the models, the methods by which to make such training data available for a new language is currently unclear, as is the needed quantity and shape of the training data. The implication of this "data" requirement may amplify the extent to which successful languages are only built by very wealthy organizations.

On the other hand, once the amount of code written in a new programming language reaches a critical mass, which is unknown at this time, LLMs can lower the cost of its continued development and accelerate its adoption. This is because a significant fraction of the cost of a language is associated with supporting developers performing tasks that LLMs already do well, such as curating examples of how to open files, or translating large existing bodies of code from one language to another. The existence of powerful automated support tools for performing these tasks is likely to significantly decrease the switching cost for developers from one language to another.

Related, programmers might feel less "attached" to an existing language and consider that as part of their professional identity. Current languages are designed mostly with the process of creating code in mind, but as we have pointed out above, with LLMs this may become a less significant component of the overall cost of creating software, and instead the focus may be on consuming, curating and verifying the behavior of code generated by LLMs.

If the production and interchange between languages becomes very low cost, there is the possibility of using different languages for different purposes. For example, if it were easier to implement an algorithm in Haskell, but a web component in Javascript, and a security audit in Rust/WASM, perhaps you might write the initial code in Haskell and ask an LLM-based tool to translate it to Javascript, integrate it into a Web API, and then ask the LLM to convert it into Rust to be read. This kind of practice is equivalent to the experience of people who speak multiple languages fluently occasionally wanting to express a subtle concept in one language rather than another.

If this, perhaps far fetched, idea becomes plausible it may be that it is the logical conclusion of the research strand discussed in the Match Mismatch hypothesis (Green et al., 1991), that the choices that we have previously seen as fundamental such as OOP vs functional, or static vs dynamic are only tradeoffs for a given purpose (reading, algorithm development, testing), rather than fundamental tradeoffs in themselves. It is perhaps slightly ironic that a tool that comes close to passing the Turing test, demonstrates for the first time the practical utility of the Church-Turing thesis, which suggests that any computation that can be performed by a Turing machine can also be programmed in any programming language.

A further productive avenue of research in LLM-enabled programming languages would be understanding how the data that can be captured about the dynamic execution of a program can be combined with its static description. This is a core task that developers perform during production debugging of systems, and it rests on the ability to manage and manipulate very large volumes of data, but we are not aware of any work currently exploring this direction.

3.5. Supporting live programming and dynamic execution of code

As we have discussed, the introduction of the ability to synthesize significant volumes of code via LLMs may facilitate a change from expressing the program logic to reviewing created programs and their behaviors. This would make it significantly beneficial to be able to execute programs in order to see what effect they have. However in the general case this is risky - what if the program does something you don't want, a say a script has learned `rm -rf *`? You wouldn't want to execute this to "try and see if it's the right program" without the ability to go back.

This has always been the case with live programming environments, and perhaps may be one of the reasons that they're most commonly used in contexts where mistakes might even be welcomed such as online code playgrounds.

4. Anti-Conclusion

LLMs are clearly a significant technical advance and will affect to a greater or lesser extent the way in which programmers go about doing their work. As some members of the PPIG community have been pointing out (Blackwell, 2022), we need to develop a response to this change.

It can sometimes feel that this is rather irrelevant, that the technology contains its own intrinsic logic, and that any opposition to this logic is futile - a position sometimes known as "techno-determinism." This would place the community solely in the position of adapting to the changes as best it could.

However historically this doesn't tend to have been a good description of invention—instead there's a middle ground that we've been proposing in this paper. If we as a community think that the creation of programming languages and their associated ecosystems of technology, *now including LLMs*, are valuable contributions to society, what can we do?

In this discussion, we've seen some of the benefits that LLM might bring the creation of new languages (e.g., easier translation, easier learning, lower switching costs), some of the challenges (e.g., the need for large training sets), and some of the potential opportunities (e.g., better use of live programming technology, a replacement of drudgery with more interesting work, and empowerment of end-user developers).

It's much too early in the life cycle of these technologies to propose a conclusion - instead we'd like to engage in a discussion with programming language and tool designers, researchers, and educators. To do this, we suggest three broad questions for consideration, and encourage those that are interested to get in touch with more:

1. How has the rising adoption of LLMs by programmers changed the way you think about your craft, and motivated or demotivated plans in your institution?
2. If we think that the design, creation, and evaluation of programming languages and tooling is of fundamental benefit to society, how should we react to the shifts associated with the invention of LLMs?

3. What are the benefits and costs of LLMs, and what are some strategies that we might use to leverage and mitigate them?

4. References

Blackwell, A.F. (1999) How to format PPIG submissions. International Journal of PPIG Administrivia, 1(1), 1-3.

Blackwell, A. F. (2022). Coding or AI? Tools for Control, Surprise and Creativity. *PPIG*, 57–66.

Blackwell, A. F., & Cocker, E. (2022). *Live Coding: A User's Manual (Software Studies)*. MIT Press.

Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. R. (2010). Example-centric programming: integrating web search into the development environment. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 513–522.

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1589–1598.

Churchill, E. F. (2019). Scaling UX with design systems. *Interactions*, *26*(5), 22–23.

Church, L., Marasoiu, M., & Blackwell, A. F. (2016, July). Sintr: Experimenting with liveness at scale. *2nd Workshop of Live Programming Systems (LIVE 2016)*. 2nd Workshop of Live Programming Systems, Rome, Italy.

Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). Liveness becomes Entelechy-a scheme for L6. *Second International Conference on Live Coding (ICLC 2016)*. Second International Conference on Live Coding (ICLC 2016), McMaster University, Canada.

Gajos, K., & Weld, D. S. (2004). SUPPLE: automatically generating user interfaces. *Proceedings of the 9th International Conference on Intelligent User Interfaces*, 93–100.

Green, T. R. G., & Fetais, N. (2016). How Notations Are Developed: A Proposed Notational Lifecycle. *Product Lifecycle Management in the Era of Internet of Things*, 659–671.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages & Computing*, *7*(2), 131–174.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). *Comprehensibility of visual and textual programs: A test of superlativism against the "match-mismatch" conjecture*. http://dx.doi.org/

Jones, S. (2023, January 20). *Coding Standards for ChatGPT*. Medium. https://blog.metamirror.io/coding-standards-for-chatgpt-ef25f0d4f6d8

Kato, J., & Shimakage, K. (2020). Rethinking programming “environment”: technical and social environment design toward convivial computing. *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*, 149–157.

Miñón, R., Paternò, F., Arrue, M., & Abascal, J. (2016). Integrating adaptation rules for people with special needs in model-based UI development process. *Universal Access in the Information Society*, *15*(1), 153–168.

Myers, B., Hudson, S. E., & Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, *7*(1), 3–28.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press.

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. In *arXiv [cs.SE]*. arXiv. http://arxiv.org/abs/2302.06590

Tanimoto, S. L. (1990). VIVA: A Visual Language for Image Processing. *J. Vis. Lang. Comput.*, *1*(2), 127–139.