Software Developers in the Age of AI

Dissertation

an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

vorgelegt von

Thomas Weber

M.Sc. Software Engineering

München, den 13. Dezember 2024

Erstgutachter:	Prof. Dr. Sven Mayer
Zweitgutachter:	Prof. Dr. Philippe Palanque
Drittgutachter:	Prof. Dr. Fabio Paternò

Tag der mündlichen Prüfung: 10. April 2025

Zusammenfassung

Mit der Einführung und Verbreitung von Künstlicher Intelligenz und Daten-getriebenen Anwendung, also Software, deren Funktionalität über die Verarbeitung großer Datenmengen umsetzt ist, sind viele Bereiche des Lebens grundlegend verändert worden. Dies betrifft viele normale Nutzer und Nutzerinnen von Software, aber insbesondere auch Software-Entwickler. In dieser Arbeit wird diese Veränderung aus der Perspektive der Mensch-Computer-Interaktion betrachtet. Insbesondere geht es darum, wie Technologien wie Machine Learning das Arbeitsverhalten, die Werkzeuge und Methoden von Entwicklern beeinflussen und verändern.

Dies wird aus zwei Perspektiven betrachtet:

Daten-getriebene Anwendung haben in vielen Bereichen neuartige Anwendungen und Lösunganansätze ermöglicht. Dies betrifft auch den Bereich der Software-Entwicklung, in dem neue Software-Entwicklungs-Werkzeuge entstehen oder bestehende Werkzeuge weiterentwickelt werden. Natural Language Processing hat bspw. neue Möglichkeiten eröffnet Code aus informellen, natürlich-sprachlichen Beschreibungen zu generieren. Neben den dadurch ermöglichten neuen Werkzeugen, werden Techniken bspw. des maschinellen Lernens auch in etablierte Werkzeuge integriert um die Arbeit in der Software-Entwicklung zu verbessern, bspw. durch Personalisierung und Adaptivität. Die erste Hälfte dieser Arbeit beschäftigt sich entsprechend damit, wie diese Daten-getriebenen Technologien angewendet werden können um die Arbeit von Software-Entwicklern und -Entwicklerinnen zu verbessern.

Für Software-Entwickler und -Entwicklerinnen hat die Verbreitung dieser Technologien allerdings noch eine zweite Konsequenz: Die Anwendungen, die sie entwickeln, werden zukünftig zunehemnd derartige Daten-getriebene Funktionen beinhalten. Da Daten-getriebene Anwendungen, im Vergleich zu traditioneller Softwar, einem grundlegend anderen Funktionsprinzip folgen, bedeutet dies auch grundlegende Veränderungen in der Arbeitsweise und dem geschriebenen Software-Code. Aus diesem Grund betrachtet die zweite Hälfte dieser Arbeit diese Unterschiede aus einer Mensch-zentrierten Perspektive. Dabei geht es insbes. darum, wie diese Unterschiede sich auf das Verhalten von Entwicklern und Entwicklerinnen auswirkt, wenn sie traditionellen und Daten-getriebenen Code schreiben. Die Ergebnisse dieser Forschung zeigen dabei die grundlegenden Unterschiede zwischen diesen Paradigmen und den damit verbundenen Herausforderungen in der Entwicklung. Daraufhin wird, unter Einbeziehung der Literatur, untersucht, wie verschiedene Entwicklungs-Werkzeuge Entwickler und Entwicklerinnen in ihrer Arbeit unterstützen können. Abschließend wird anhand eine konkreten Implementierung eines Entwicklungs-Werkzeugs evaluiert, inwieweit unterschiedliche Design-Konezpte von Entwicklungs-Werkzeugen den Wünschen und Anforderungen von Entwicklern und Entwicklerinnen in der Daten-getriebenen gerecht werden und die Software-Entwicklungs unterstützen können.

Abstract

The introduction and widespread adoption of Artificial Intelligence and data-driven applications, i.e. software systems that rely on large sets of data to define their functionality, have created a paradigmatic shift in many areas of our lives, not just for end users but particularly for professional software developers. In this dissertation, we take a human-computer-interaction perspective on how technologies like Machine Learning have affected software developers, their work practices, tools, methods, etc. To this end, we consider two perspectives:

First, just as data-driven applications have enabled new types of applications and solutions to challenges in many areas, they have also given software developers new tools and approaches for writing and creating software. For example, natural language processing has yielded systems for quickly generating working code from informal descriptions. Besides new tools, applying Machine Learning techniques to the wide landscape of existing development tools also promises to improve the work of developers, e.g. with personalized and adaptive tools. Thus, the first half of this dissertation will be dedicated to investigating how data-driven applications can be applied to enhance the software development experience.

However, for developers, the widespread adoption of data-driven applications has a second consequence: the kind of software they write changes to incorporate these data-driven functionalities. Since there are paradigmatic differences in how data-driven systems operate, compared to more traditional software systems, this naturally also affects how they are developed and what their underlying code looks like. Therefore, the second part of this dissertation is first concerned with these differences from a human-

centered perspective, particularly how developer behavior differs when working on the development of data-driven and traditional software systems, focusing on code reading. Since this demonstrated that data-driven development has its unique challenges, we then consider how tool support can help developers in these changing circumstances. To this end, we explore the literature on tooling. This revealed both evolving and new tools for this new reality but also highlighted areas where development tools can still be improved and tailored to the requirements of data-driven development. Thus, we finally explore new tool implementations to address the challenges of data-driven development as well as the needs and wishes of developers that we elicited throughout this work.

Insights from these two parts demonstrate how the changes in the underlying technology also drive changes in the human perspective. As the technology continues to evolve at a growing pace, it will become equally important to continuously assess this relationship to address any potential mismatch, ensure the continued quality of software in the face of ever-increasing complexity, and ensure developers can continue to effectively leverage these technologies to shape our lives with technology. This work therefore can only contribute only a momentary assessment of the state of software development and a foundation for future research into how data-driven applications continue to affect software developers.

Acknowledgements

Completing this thesis would not have been possible without the extensive support and supervision of both Heinrich Hußmann and Sven Mayer. Heinrich, as my initial supervisor, first arranged for my position at fortiss to get me started in research and then took it upon himself to find me a position at LMU so I could pursue a doctorate. Sven, of course, took over the role of supervisor after Heinrich's untimely passing. Not only has he provided incredibly in-depth, hands-on guidance for the research of this dissertation in this role, but also far beyond, for virtually any aspect of the academic experience. Gratitude also goes to Philippe Palanque and Fabio Paternò for volunteering to work through this document as external reviewers as well as Albrecht Schmidt and Dirk Beyer as head of the examination committee.

In addition to the research, a doctorate comes with considerable technical and administrative overhead, for which Rainer Fink, Christa Feulner, and Franziska Schwamb have always proven excellent support — and an equally excellent source of dog treats.

Crucially, working on a project like this also means working with many lovely colleagues and collaborators, including but not limited to: my office mates Florian Bemmann and Steeven Villa; my fellow Prüfungsamt survivors: Florian Lang and Bettina Eska; the fortiss compatriots: Gesa Wiegand, Matthias Schmidmaier, and Zhiwei Han; my co-authors (submitted, not necessarily accepted): Amy Melniczuk, Malin Eiband, Rifat Amin, Robin Welsch, Jan Leusmann, Maxi Windl, Carl Oechsner, Xuedong Zhang, Alois Zoitl, Andreas Butz, Yuanting Liu; my teaching companions Nađa Terzimehić, Dennis Dietz, Svenja Schött, Jesse Grootjen, Beat Rossmy, Changkun Ou; the DigiLLab team: Evelyn Kovac, Florian Schulz-Pernice, Sonja Berger, Sabine

Becker, Meral Roeben, Johanna Vejvoda, Anne Lohr, Julia Murböck, Alex Kacina, and Frank Fischer; and many more like Daniel Buschek, Linda Hirsch, Clara Sayffaerth, Henrike Weingärtner, Christina Schneegass, Luke Haliburton, Julian Rasch, Francesco Chiossi, Sophia Sakel, Yannick Weiss, Matthias Hoppe, Fiona Draxler, Sarah Völkel, Teodora Mitrevska, Heiko Drewes.

Much of my research is also built to varying degrees on the work of the 40ish students whom I got to supervise during their Bachelor's, Master's or project theses. Without listing all of them, Maximilian Brandmaier, Rafael Thiel, Christina Winiker, and Janina Ehe stand out, as their work has directly contributed to the publications upon which this dissertation is based.

Last but not least, there are of course friends and family and especially Nati and Mochi, who have provided unconditional support throughout the years that it took to complete this.

Thank you all.

Table of Contents

1 Introduction

In our modern world, just about every domain is enabled, controlled, or in some way influenced by software. Thus, high-quality software is an essential part of making modern society, and many of the things we now take for granted work. The recent widespread proliferation of Artificial Intelligence (AI) can contribute to this. This thesis explores how AI can support the work of software developers in the pursuit of high-quality software, focusing on two aspects: first, how AI can be utilized to improve existing tools and create novel support mechanisms. As software developers are also those who create Artificial Intelligence systems, this thesis will secondly explore how this process differs from the creation of traditional software systems and how we can support the creation of high-quality Artificial Intelligence systems as well.

To make sure that software can be of high quality, it was clear already at the beginning of the field of Computer Science that it would be necessary to establish best practices, and processes, and turn software development into an engineering discipline. In consequence, in the 1968s the seminal NATO Software Engineering Conference in Garmisch-Partenkirchen [21] initiated the field of *Software Engineering*, analogous to mechanical engineering or civil engineering, which has become an essential part of Computer Science.

Since these beginnings, Software Engineering has come a long way and has become a broad, established engineering discipline with an ever-growing population of practitioners [61] and active research to push the field further ahead and improve how software is created.

Recently, the field of Software Engineering has experienced considerable changes due to advances in a second, similarly old research field in Computer Science: Artificial Intelligence (AI). Unlike Software Engineering, the development of AI research over the years has been less consistent. After some early seminal work in the 1950s and 1960s, followed by the “AI winter”, a phase of reduced interest and funding in the 1970s and 1980s, the field has seen a resurgence in the last decades, particularly due to advancements in Machine Learning (ML) and the availability of large volumes of data and computational resources. This has enabled many new applications and technologies that have found their way into various aspects of our lives, from object and speech recognition [236, 267, 344] to natural language processing [76, 193], or recommender systems [136, 337]. More recently, systems such as Large Language Models (LLMs) have enabled new generations of systems, tools, and user interfaces that could increase productivity, simplify usage, and overall improve the work and lives of many. Beyond this, there are many domains where active research and engineering is taking place to create AI-based technologies, for example, autonomous driving [101] or healthcare [277].

Naturally, the emerging capabilities of AI systems also affect Software Engineering. While the promise is that AI will greatly simplify and accelerate software development, the actual effects are not yet clear. For example, emerging technologies such as LLMs have found broad use in software development. However, AI systems continue to change and evolve, driven primarily by technological advancements. Understanding what effect this technology has, in general, and on software developers specifically, could help guide and steer these technological advances more effectively and to greater ultimate benefit. This is particularly important because Software Engineering is a human endeavor — software is ultimately still being developed by humans, making it a crucial part of the effort to ensure its quality. As humans, software developers have developer-specific behaviors and solution strategies to deal with the complex task of building software. Likewise, with new technologies, new behaviors and practices will emerge and affect how developers use these technologies. The domain of human-computer interaction plays an important role in understanding these human-centered aspects of software development. Only by investigating how emerging technologies, like AI, are being used by developers, we can gauge which effects and benefits these technologies can contribute in practice. Additionally, having a sound understanding of the human perspective on software development is an important prerequisite for creating support mechanisms that do not just promise improvements in theory, but are actually useable and useful in practice.

Thus, the goal of this thesis is to further our understanding of the human factors that contribute to software development in a rapidly changing environment. This thesis is therefore positioned at the intersection of these three areas, Software Engineering, Artificial Intelligence, and human-computer interaction. The goal is to explore how the recent rapid advancements in the field of AI affect software development practices from a human-centered perspective, i.e., focusing on the effect on software developers specifically and how to support them in the creation of high-quality software.

1.1 AI and Data-driven Applications

Historically, various technologies and techniques have fallen into the category of Artificial Intelligence. Given this diversity but also the rapid pace at which this field is evolving, precise definitions of the terminology can be challenging, beginning with what "Artificial Intelligence" itself is exactly: according to the definition of McCarthy [185] from as early as 1955, "Artificial Intelligence" describes "the science and engineering of making intelligent machines", thus sidestepping the definition of "intelligence" in this context. Over time, new definitions have emerged, focusing on different aspects and taking different perspectives of what constitutes "Artificial Intelligence" (see De Spiegeleire et al. [63] for a comprehensive typology).

In modern AI systems, Machine Learning has crystallized as an effective technology to facilitate this. In particular, methods that rely on large volumes of data have proven very productive. These methods extract patterns and the desired functionality using statistical methods. First and foremost among are various forms of *neural networks*. Thus, many of the current software systems that are considered AI use these Machine Learning methods.

While ML methods are widely successful, they are by far not the only methods [44]. Furthermore, it may very well be that new methods will emerge in the future that utilize different mechanisms to achieve the same goals. For this reason, this work focuses on the variant “data-driven systems” or “data-driven software”, as they are sometimes referred to in the literature (e.g. [120]). This name emphasizes the key difference between this generation of systems and other types of software: its reliance on data as the key driver. Consider as an illustrative example a software that detects faces in images (a not uncommon use case for ML). If one were to solve this *traditionally*, the developer would have to come up with a systematic algorithm, e.g., scan through the pixels sequentially and store, transform, and compare the pixel values. They would then encode these individual steps in code, thus spelling out the desired functionality. On the other hand, when using *data-driven* software, the desired behavior is not explicitly described step by step. Instead, the developer provides the systems with a large volume of data that contains many examples of the desired input-output pairs. Using ML methods, e.g., neural networks, the software must then infer the relationship between the input and the output from the data. Thus, the desired behavior is encoded in the data and not in the code which the developer writes. Instead, the work of developers typically involves constructing the infrastructure around this process, e.g., how data are read and preprocessed, the architecture of the ML model, and how the results are used. Consequently, this development paradigm differs from the traditional approach with changed or new steps in the development process, like feature engineering and model tuning [120].

While the exact steps of the development process may change with the inclusion of AI, building software is still a human activity. Creating software, data-driven or otherwise, can be complex, though, particularly when the software itself is of a certain complexity or size. The tasks and responsibilities of a developer during this can be quite broad, including various aspects such as project and team management, customer or end-user interaction for requirements engineering, maintaining the underlying infrastructure, for development but also deployment, communication, quality assurance, and much more. Adding AI to the development further complicates this by adding additional steps and processes for, e.g., collecting and managing data [120], or fine-tuning these probabilistic systems towards the desired behavior. Additionally, it also increases the urgency and importance of other aspects, e.g., ethical [129] and legal questions [20, 285, 328]. Thus, it becomes evermore imperative to support software developers as best as possible, so they have the best chance of producing high-quality software.

1.2 Supporting Developers with AI

Software Engineering research has produced and is continuing to contribute support for the full breadth of developer activities. There is work on anything from high-level tasks like understanding end users, requirements, and the underlying issues, over drafting a technically viable system down to the low-level activities of encoding instructions in a programming language. Some of the knowledge from this research on Software Engineering may be transferrable to ML. However, certain aspects of ML, e.g., its probabilistic nature, pose a challenge for applying existing research findings to this new development paradigm. Thus, there are ongoing efforts to add, extend, or adapt this research.

Out of this broad field, this work will primarily focus on one core activity of developers: *the actual writing of the software in code*. Virtually every professional developer already uses a whole array of tools for writing code, from basic tools like simple editors to complex tool suites and Integrated Development Environments (IDEs) that bundle many features into a unified interface. Their popularity highlights the impact that we can have by better understanding how to effectively apply and improve coding tools.

The increasing ubiquity of AI has many effects on the tooling landscape, one of which are new and improved tools, e.g. computational notebooks which are popular for data-driven development [43, 139, 162, 307] or the increased levels of automation, e.g. for bug detection [7, 293] or code generation [141, 208, 229]. Of course, these new support mechanisms may require a change in established work practices, processes, and developer preferences. Using AI to automate steps in the Software Engineering process also raises concerns about best practices in the future. While AI may take over many of the tasks from the developer, this is not without challenges. For example, modern AI systems are inherently probabilistic, raising questions of reliability, and how to ensure quality [306]. A lack of knowledge about the underlying training data also adds uncertainty, e.g., how licensing is affected for code derived from AI [20, 285, 328]. While there may be approaches to these challenges that focus on technology, a human-centered perspective will likely be equally important since, after all, any technological solution can only be successful if it is useful, usable, and ultimately used by humans.

Beyond this, there is also the question of how adopting AI can affect developer behavior. For example, when developers can use AI to quickly generate large volumes of code, they inadvertently can also quickly add complexity, while potentially spending less time on reflecting on the problem and the solution. Thus, it may be increasingly important that developers understand Software Engineering principles, can assess software quality, and are able to steer the AI tools toward the desired level of quality. At the same time, AI support for programming also promises to lower the entry barrier and increase accessibility, beginning to blur the line between trained professionals and end-user developers. When people who may not have received a formal education in Software Engineering are now broadly able to easily create software, it also becomes increasingly important that Software Engineering principles and code quality become similarly accessible as well.

> **Software Development with AI**
> *Thus, the first key topic of this work is concerned with the effects that using AI as a support mechanism has on software developers as an important stakeholder in Software Engineering.*

We contribute to this body of knowledge by investigating different aspects of how tools that incorporate data-driven functionality can support software developers. To this end, we consider how data-driven systems benefit developers, focusing on two questions:

> **Research Question 1:**
> How can existing development tools be extended with data-driven features features?

> **Research Question 2:**
> How do devleopers benefit from newly emerging, AI-powered development tools?

1.3 Supporting Developers who Build AI

Beyond this, software developers are in a somewhat unique position: not only do they benefit from many of the advancements brought by AI, but they also fundamentally shape these changes. AI ultimately is also just software that has been and will be created, adapted, and modified by developers. Thus, they play a central role in facilitating the changes that AI brings to our daily lives and society as a whole. This opportunity does come with a responsibility and further emphasizes the need for a solid foundation in Software Engineering to ensure high-quality software.

However, while AI is at its core software, as mentioned, it does present a paradigmatic shift in how these software systems operate: in traditional, i.e., not AI-powered, software, the core working principles are encoded in algorithms and translated into a programming language. This, however, becomes unfeasible for many of the domains where AI excels due to the sheer complexity of many of the tackled problems. Therefore, in modern AI systems, powered by ML, the instructions encoded by developers in a programming language often only describe the infrastructure and data processing, while the core functionality comes from patterns in large volumes of data. Extracting these patterns happens automatically by means of Machine Learning methods. This not only marks a paradigmatic shift in how the software operates but, in consequence, will also greatly affect how developers think about it and how they behave during the development of these kinds of systems.

At the same time, the ubiquity of these systems and their use in critical domains like healthcare and mobility make it increasingly important that these AI systems operate reliably and as intended. However, orderly and reliable Software Engineering of complex traditional systems was already full of challenges and pitfalls without AI. Adding the complexity of AI highlights more than ever that we need to understand how developers work in this new reality. Understanding the actual practices, challenges, and needs of software developers from a human-centered perspective will help us facilitate good engineering practice and design better support mechanisms and tools. This will be necessary to mitigate the added complexity for the developers, and thus is an important step towards well-engineered, high-quality software.

Software Development for AI
Thus, the second core topic of this work is concerned with a human-centered perspective on this paradigmatic shift. It will explore how developers behave and how to support them when they develop software that uses AI in its functionality.

In an effort to support developers in building software, there are already many tools from academic research, corporate development, or through community efforts (cf. Section 3.2). However, many of the existing tools were originally built to support developers in an era when AI was not as commonplace. Thus, the question is how these tools hold up in this new and changing world and whether they are still suitable for the unique challenges that may arise while creating AI systems, e.g., for dealing with large volumes of data, complexity, etc. To understand of software developers and their tools are affected, we therefore work towards answering the following question:

Research Question 3:
What are the differences in the development of data-driven applications compared to the development of traditional software that affects the behavior and needs of developers?

Understanding the effect of AI on the behavior of developers can help to gauge in which areas existing knowledge on Software Engineering still applies, and where the behavior is so different that new or evolved tools are necessary. With a better understanding of the developers' behavior, we then investigate how specific tool implementations can support the development of data-driven applications, including an exploration of how different programming paradigms can help them:

Research Question 4:
How can different tooling paradigms help to address the challenges that software developers face during the creation of data-driven systems?

By addressing these questions, this thesis provides a better understanding of the effects that the inclusion of AI in Software Engineering has on developers and their work. Our findings show that AI has great potential to support developers by giving them new and evolved tools that offer concrete benefits like personalization or higher productivity. At the same time, it also demonstrates how the creation of AI systems is fundamentally different in some aspects. To cope with these differences, existing support mechanisms may be inadequate and developers will require new or adapted solutions to help them. The growing complexity that comes with adding AI makes it essential that software developers receive adequate support whenever possible, in a way that meets their needs, tackles the challenges of AI, and steers them towards the goal of higher-quality software. A recurring thread in our work is the specific challenges of supporting software developers, e.g., to achieve acceptance and adoption. This emphasizes the importance of a human-centric perspective in this space. Thus, the success of AI applications and, with it, the experience of end users and developers alike will rely on insights from the intersection of research on AI technology, human-computer interaction, and Software Engineering practices.

Summary and Overview of the Thesis

Table 1.1: Contributions of the individual chapters to the research questions

	RQ	Publications	Contributions
Chapter 2	RQ 1	[C1, C6]	Implemented tool, collected data set, data from interview
	RQ 2	[C2]	Data from user study, incl. questionnaires, interaction data, static analysis of generated code
Chapter 3	RQ 3	[C7, C8]	Data set of code repositories, static code analysis, eye tracking, and questionnaire data and analysis
	RQ 4	[C3–C5]	Literature review, tool design concepts, tool implementation, data from iterative user studies incl. interaction data, questionnaires, and interviews

This thesis is divided into two main parts: Chapter 2 covers the first set of questions about how we can improve the development process and the activities of developers *with* data-driven systems, i.e., by applying them to improve the work of software developers. This includes efforts to enhance and improve existing tools with data-driven technology to facilitate personalization and adaptation of the interface, as well as an exploration of what effect this may have (Section 2.1).

Furthermore, with the evolving capabilities of AI systems, there are new and emerging tools as well, e.g., Large Language Models (LLMs). Thus, this chapter will also cover the effect that this technology has and how AI coding assistants can affect the productivity and behavior of software developers (Section 2.2).

The second part deals with the fact that, compared to many other domains, software developers are not only users of AI but are also in charge of building it. Given their nature, developing data-driven systems presents a paradigmatic shift, compared to developing traditional software. Chapter 3 will therefore explore the process of software development *for* data-driven applications and what this means for software developers. It first investigates what differences exist between traditional and data-driven software, e.g., in terms of code structure, and how this affects the behavior of developers, e.g., code reading (Section 3.1).

Based on these insights, this second part then explores how tools that help developers build data-driven systems have evolved and may continue to adapt to these changing circumstances in the future. Based on a literature review (Section 3.2) and newly built tools that utilize multiple programming paradigms (Section 3.3), this chapter provides some insights into what may be necessary to better support developers in a way that is useful and finds adoption in practice.

Following the findings from this research, I will reflect in Chapter 4 on the state of software development and how the three research fields of Software Engineering, Artificial Intelligence, and human-computer interaction can come together to facilitate better development practices and ultimately better software. Finally, Chapter 5 will conclude with potential future directions for these fields in a world where data-driven applications become increasingly ubiquitous and influential.

Research Context

The research summarized in this thesis was conducted during my time at the Media Informatics Group and LMU Munich, starting in 2019. The work was supervised by Heinrich Hußmann and Sven Mayer. Parts of it are based on a series of student theses. See the list of Contributing Publications for a detailed breakdown of the contributions. Additionally, Section 2.1 was informed by research conducted at fortiss in 2018.

In addition to the research in this thesis and extended work on software development tools [315], I also contributed to research projects on interactive and explainable AI [312] for the automotive context [322] and for image restoration [105, 313], as well as a series of projects headed by Amy Melniczuk to enhance education with tangible user interfaces [163–168, 170].

2

Software Development with AI

As AI technologies are becoming more widely available, many domains attempt to integrate it into their existing workflows, processes, and tools to reap the promised benefits. Software development is no exception to that. Common software development activities like bug detection [7, 293], or code [141, 208, 229], test [58, 266] and documentation generation [115] are some areas where AI techniques may exceed human capabilities and alternative existing technologies. These examples focus primarily on the automation of common development activities, thus taking workload off of developers. Yet, AI also has the potential to improve the development experience when a human still needs to be in the loop. This typically comes in the form of augmenting existing tools with AI features. In this chapter, we will explore RQ1, i.e. how we can augment existing development tools using AI, specifically by adding adaptivity and personalization to IDEs as common development tools. Secondly, we will address RQ2 and investigate how novel, AI-powered tools for automating one of the most common development activity — writing code — affect developer productivity.

2.1 Software Development with an Adaptive User Interface

This section is based on the following publications:

Thomas Weber, Rafael Vinicius Mourao Thiel, and Sven Mayer. “Supporting Software Developers Through a Gaze-Based Adaptive IDE.” in: *Proceedings of Mensch Und Computer 2023*. MuC ’23. Rapperswil, Switzerland: Association for Computing Machinery, 2023, pp. 267–276. ISBN: 9798400707711. DOI: 10.1145/3603555.3603571
Matthias Schmidmaier, Zhiwei Han, Thomas Weber, Yuanting Liu, and Heinrich Hußmann. “Real-Time Personalization in Adaptive IDEs.” In: *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization*. UMAP’19 Adjunct. Larnaca, Cyprus: Association for Computing Machinery, 2019, pp. 81–86. ISBN: 9781450367110. DOI: 10.1145/3314183.3324975

One type of feature that benefits greatly from data-driven software is personalization. Tailoring a system to specific user needs can allow the user to work more efficiently and effectively. While this has been possible in the past, it would require a considerable configuration effort by the user. Automated changes on the other hand did not have the flexibility that would be necessary for personalization to individual user needs. The introduction of AI to this process promises to enable highly flexible, very specific adaptation. The data-driven nature of modern AI should allow it to detect highly user-specific patterns from usage data which can then lead to automatic, user-, context-, and situation-specific adaptations. The goal of these adaptions typically is to enhance productivity and minimize the complexity of an interface.

One type of software development tool that, due to its abundance of functionality, can be complex are IDEs. Due to the number of features, they risk to overwhelm users [154]. Yet, rarely anyone requires all the features, particularly for novices, as the sheer volume of panels, buttons, and parameters can be overwhelming [258]. Selecting the correct sub-system and working context for a task adds extra steps and mental load [154]. Command-search and context-based hints are some of the methods that modern IDEs use to address this issue. One step further are adaptive IDEs, which apply user modeling to adapt and personalize the system to a developer’s individual preferences and demands. Adaptive UIs can support the user by shifting some workload and responsibility to the system; to be successful, the system must determine the user's intent in a given context [175]. Several publications investigated concepts for user adaptation in software development. Robillard et al. [252] present an overview on recommendation systems for software engineering (RSSE). They describe systems that provide recommendations based on user characteristics, conducted tasks, task characteristics, or past user actions. Murphy-Hill et al. [198] combine collaborative filtering and a most widely-used algorithm to make developer command recommendations. With a similar goal, Damevski et al. [60] present a topic modeling approach for predicting future developer behavior in an IDE. Bulmer et al. [34] also predict the developer's next interaction with several statistical learning models.

In the following, we present a concrete implementation of an adaptive IDE that leverages usage and gaze data to support developers by adapting the layout of their Visual Studio IDE. Furthermore, we contribute a first assessment of the feasibility and perception by developers of such a system. This addresses RQ1 with a focus specifically on adaptive interfaces as an example of a data-driven functionality in software development tools.

To evaluate the potential benefits of adaptive interfaces, we developed a Visual Studio extension that modified the interface to be adaptive. We conducted an early evaluation study (N=6) with software developers, which highlights that meaningful adaptations can be achieved even from a small data set. Moreover, we gain insights into the perceived benefits and drawbacks of layout adaptation and explore the desirability of adaptive UIs for software development. Our results show that UI adaptations can support developers during coding if, throughout the automated process, they still ensure a degree of user control. The key here is that the layout adaptation is not unexpected and distracting for the developer, which can be achieved by well-defined, infrequent, and less abrupt adaptations.

In contrast to previous work we do not only explore how to identify and predict individual user behavior, but also how to apply recommendations and interface adaptations. Our system adapted the IDE up to 2.3 times per minute, but developers reported that they perceived these changes only minimally. Additionally, developers are generally open to layout adaptations supporting them. These are promising first results as our gaze-based adaptation could support them while not distracting them from their primary task. On the other hand, we also found that developers are skeptical when the adaptation would restrict functionality in a future version of this system. In summary, developers favor automatic adaptation as long as they have the opportunity to override it if needed; thus, they never restrict them in their development.

2.1.1 Related Work

While adaptive UIs are an idea that has been around for a while, both, the application for software development tools and the use of gaze as an information source, has been explored only to a limited degree. The following section will outline some general background information and highlight some of the existing literature on adaptive UIs in this specific context.

2.1.1.1 Adapting UI

The UI layer is a crucial component in software applications. Ultimately, it connects users to a software's functionality. So even a well-made software application might fall short due to a suboptimal UI layer. Whereas some older user interface development techniques, such as *universal design* [180], *inclusive design* [144], and *design for all* [286], advocate a *one-design-fits-all* approach, a user interface is dependent on its *context of use*, which can be "decomposed into three facets: the user, the computing platform [...] and the complete environment" [39]. Hence, it is hardly possible for one user interface to accommodate all the use cases in a given context of use, leading to a potentially diminished user experience [6] and less effective operation of the software application.

Building only a single, fixed UI forces the designer to predict all possible variants of usage — a task hardly possible. Alternatively, the UI can be configurable to user preferences, but this requires effort from the user and is inflexible in changing circumstances. To mitigate this, one can instead design the UI to be *adaptive*. Such user interfaces with adaptive properties are occasionally referred to as multi-context or multi-target UIs [40], but this work will use the term *adaptive UI*.

An adaptive UI extends a regular user interface by adding adaptive elements and layouts [25], thus tailoring the presentation of functionality to a user's task at hand, personal preferences, or different contexts of use. This can potentially “reduce visual search time, cognitive load, and motor movement” [85]. In essence, an adaptive user interface attempts to predict what a user might want to know or do and assist them with it, so they have to spend less time and effort. This prediction can be defined by users in the form of rule-based systems [94, 157], fully automated using, e.g., Machine Learning [77, 160], or a hybrid human-in-the-loop system where user feedback steers the automated prediction [188, C1]. All these variants roughly follow the blueprint for a “dynamic [and] seamlessly personalized” [175] adaptive user interface as outlined by Liu et al. [175] where an adaptive UI executes the following functions: (1) Observe the interaction between the user and software application, (2) Identify distinguishable episodes, (3) Recognize user behavior patterns, (4) Help users according to recognized user plans, and (5) Build user profiles to enable personalized interactions.

To perform these tasks, an adaptive UI should “extract as much useful information [...] as possible” [175] and categorize that data into “episodes”. Data to make this categorization can come from various sources, e.g., internal metrics from the software, user interaction with mouse and keyboard, or, as we further explore, physical properties of the user, like gaze. A sufficiently large and varied database allows the system to find hidden patterns in the data, which inform the adaptive functionality.

Whenever the system recognizes a pattern, thus being able to predict what the user is going execute, it can adaptively offer support or simplify the actions. Lastly, the system can create user profiles to store relevant user-defined preferences and user-specific patterns to provide long-term and evolving adaptions and personalization.

To further describe adaptive systems, Salehie et al. [260] presented a hierarchy of adaptability in general software systems. This hierarchy also applies to adaptive UIs (cf. Akiki et al. [6]). The hierarchy is as follows:

Context-awareness “indicates that a system is aware of its context, which is its operating environment” [260] Only if the UI is aware of its context, it is able to trigger adequate adaptations [6].

Self-configuring “is the capability of re-configuring automatically and dynamically in response to changes” [260] The context of use is not static; it is constantly evolving (for example, a user’s computer skills improve). Therefore, the adaptation rules have to be kept up to date. This can be done with a mechanism that monitors changes in context; another alternative is the incorporation of user feedback.

Self-optimizing "is the capability of managing performance and resource allocation in order to satisfy the requirements of different users" [260]. In the context of adaptive user interfaces, this can mean that a system can "self optimize by adapting some of its properties" [6].

Where an adaptive UI can be positioned within this hierarchy depends on several factors, e.g., the volume and variety of data. The more data a system has available, particularly real-time data, the more it can typically infer about the context and user and thus offer more complex adaptations. To this end, we explore gaze as an additional data source beyond interaction patterns in the UI itself.

Eye tracking as a data source for adaptive UIs has been the topic of some research projects, e.g., by Pfeuffer et al. [232] who utilized gaze data for an adaptive UI in an augmented reality (AR) setting to show or hide virtual information depending on whether the user looks at the real-world environment or the AR elements. Göbel et al. [96] also uses eye tracking to adapt a UI for displaying maps to hide irrelevant information and highlight relevant UI elements. An alternative to this is to rearrange UI elements to minimize load on the user, as done, for example, by Gebhardt et al. [93].

Zhang et al. [335] similarly used gaze as a means to infer users' intent for a text formatting task. Using gaze data alone, they were only able to achieve an accuracy of roughly 40%. While they did also use actions for intent classification to greater success, they did not yet combine multiple data sources and propose this as an interesting approach. Furthermore, their analysis, so far, focused on accuracy metrics, and did not assess adaptations based on intent prediction or the subjective perception of these.

As these examples show, using gaze data has proven an interesting and feasible source of data for adaptive UIs in select specific contexts and also to infer a user's mental state [55]. However, it appears gaze should be leveraged in combination with additional data sources [335]. With this work described in this chapter, we contribute to this by investigating an adaptive UIs using gaze and interaction data in the context of software development tools where existing tools offer a high degree of complexity, putting a lot of mental load onto the user [154, 258]. These UIs are often not very visual and rely heavily on the text, while many expert users often rely on shortcuts. A lot of the interaction thus happens during reading and thinking of the developer in front of the UI, which makes common data sources for adaptations like user input or prior-knowledge [295] less helpful on their own, thus making gaze an interesting additional data source.

2.1.1.2 Adaptive UI Modifiactions

The previous section described on which basis an adaptive user interface should make its adaptations. Now, how can the adaptive UI then translate those triggers into the visual layer and offer assistance? We distinguish between different degrees of adaptation, ranging from adaptations of the visual presentation to adaptations of the underlying system functionality.

Changes to the presentation can start with very subtle adaptations. Providing contextual hints and tips is one fairly common example. Instead of showing additional information, an adaptive system can also do the opposite and hide information, which presents a slightly more serious intervention in the interaction. Ideally, this should only affect less relevant information and be easily reversible. Since this must not be a binary decision, the information is there or not, but can also mean that information is partially hidden or made less visible or intrusive, sometimes described as “dimming” the information [35]. The goal here is typically to “reduce cognitive overload and support task focus flow” [C1].

Going beyond simple changes to the visual appearance, adaptations can also trigger or change system behavior. This is typically done in an effort to “dynamically enhance the user’s workflow” [C1], provide shortcuts, and increase efficiency. A system can determine what sequence of actions is typical for a user and then provide specific shortcuts that group these actions into a single command. Taking this one step further, when the system is able to predict what a user typically would do next, it could go ahead and execute those commands automatically (cf. Schmidmaier et al. [C1]). To predict the next upcoming command, the system will require an abundance of contextual information, though, particularly in complex systems where adaptations make sense, the search space for potential commands tends to be quite large.

Such dramatic changes only make sense in a few circumstances and require the user to be very familiar with them so as not to be confusing, as they otherwise run the risk of having adverse effects on usability, which could negate the benefit of the adaptations.

2.1.1.3 User Acceptance

Adaptations, particularly the more aggressive interventions into the system's functionality like those mentioned above, can certainly reduce the time spent on tasks, but they can also increase the risk that it becomes unclear to the user why and when actions are executed and reduce the predictability of the system. "Disorientation and the feeling of losing control" [227] make it necessary that an adaptive UI offers additional mitigating support mechanisms to maintain a certain degree of understanding and control like post-hoc explanations, confirmation requests, feedback and undo functionality. Whenever adaptations occur, the user has to be able to understand the automatic changes to the UI — it is important to support a "feeling of continuity between the UI before and after adaption" [70] and to show the user what caused the adaptation, leading to better predictability of the system behavior [134]. The implementation of a feedback loop can increase transparency and user acceptance. Furthermore, user feedback should not only be used for future adaptations but, if applicable, undo current adaptations as well to give back control to the user. Since incorrect adaptations can be considered an additional burden to the user [319], the benefits of correct adaptation have to outweigh the costs of those usability hiccups. Using user-centered methods during design and continuous user feedback should help reduce the average usability cost of an adaptive UI, making them more viable and beneficial.

2.1.1.4 Adaptive UIs for Software Development

Software devleopers, of course, are a different target group than general end users. Given their expertise with technology, they may be open to controlling an adaptive

UI or may even expect this level of user control. There is, in fact, research into how software developers can facilitate and control UI adaptations, like the Adapt-UI system by Yigitbas et al. [331, 332] which uses the model-driven development to simplify the creation of runtime adaptations of UIs.

So far, most such systems that predict the developers behavior are limited in how the prediction is utilized [34, 60, 92, 198], though, typically in the form of action recommendations and not in automatic adaptations. Robillard et al. [251] present an overview of such recommendation systems for software engineering (RSSE). They describe systems that provide recommendations based on user characteristics, conducted task, task characteristics or past user actions. The presented applications support developers by recommending reuse of specific code fragments, expert consultants, code examples, information navigation or what parts of code to change next. A similar overview on software development recommendation systems is given by Happel et al. [108]. Murphy-Hill et al. [198] combine collaborative filtering and multiple discovery algorithms to make developer command recommendations. Based on a dataset of around 4000 developers, they were able to recommend commands with an accuracy of about 30%. In addition, they conducted a live study, manually presenting recommendations to novice and expert users to evaluate usefulness and acceptance. In a more recent work, Damevski et al. [60] present a topic modeling approach (Temporal Latent Dirichlet Allocation) for predicting future developer behavior in an IDE. However, they do not implement or evaluate any forms of recommendation or system adaptation. Similarly, Bulmer et al. [34] also predict developers' next interaction with up to 64% accuracy using a neural network trained on a dataset with interaction data from 3000 developers. Regarding personalized recommendation in IDEs, Gasparic et al. [92] introduced a command recommender system based on the developer's current work context and his previous knowledge of commands. With this approach, the user also gets personalized recommendations for relevant commands that he is assumingly not aware of.

These systems build on the interaction data, which is available directly from the development environment but do not yet draw on additional external sources of information about the developer, like gaze. Still, in the context of software developers, eye-tracking has proven useful, though, yielding information about the behavior of software developers, e.g., code reading behavior [37, 228, C7], code comprehension [23, 254], or debugging [22, 118], so it is a promising additional source of information for UI adaptations. We will therefore explore how gaze data can contribute to an adaptive IDE.

2.1.2 A Gaze-Adaptive IDE

While adaptive UIs can come in many forms, the goal is always to ease user access to functionality or information. In the context of an IDE, a UI that contains an overabundance of information and functions, we see two particular aspects of the UI that can be simplified by automatic adaptation: (1) timely access to relevant information and (2) choosing the correct functionality from the large range of IDE features. For this, we utilize the users' gaze and interaction data to adapt the UI's layout to make relevant information and features more visible. In detail, we developed a Visual Studio plugin, which adapts the layout of the IDE to support the developer during various tasks by changing the size of window elements.

2.1.2.1 System Design

We base the design of our prototype — as most adaptive systems — on two primary decisions: (1) what does change in the UI? and (2) what triggers these changes?

When deciding what parts of the UI should be changed, we considered that radical UI changes can also be distracting, particularly when UI elements change position [85, 194]. This is particularly unfortunate when the developer is in the middle of a lengthy task and is in a state of flow [205], where distractions may be worse than what is gained by the adaptations. Therefore, we decided to focus on information dimming and highlighting. Rather than radical changes in the UI, this means gradually removing less relevant or redundant information and making important information more salient.

Most IDEs continuously present much information unrelated to the user's current task and only becomes important in certain situations. Being ever present, the user may easily ignore or overlook this information when it suddenly becomes relevant. Thus, we focus on ensuring that this information is noticed adequately by changing the layout and dimensions of the IDE's panels to increase their visual presence. This way, information will grow gradually if the underlying system deems it increasingly relevant to a variable size, and no UI element vanishes or becomes inaccessible. By tuning our system, we also ensured that the main editor of the IDE, where mostly all user input and little output happens, remains sufficiently large, as we do not want to deflect from the IDE's main purpose, to write code. Layout changes are also fairly easy to implement, making it a good choice for early design explorations, and it is easy to determine whether an adaptation was actually performed.

We generally decided to limit the changes and only apply them in small increments to minimize the risk of the aforementioned unwanted effects. Other, more aggressive changes, like pop-ups, changes in color, or even animations to get the users' attention, are possible. Such more drastic changes are likely more subject to personal preferences, though, and would skew the perception of our users. When subdued changes already elicit positive feedback, then future studies can determine how much adaptation is acceptable. Likewise, combinations of different adaptations are a reasonable approach for real-world applications. However, this makes any systematic evaluation challenging, so we focused on layout changes for now.

Concerning the trigger, we wanted them to be triggered by the context of use and guided by gaze data. With contextual information, we had two options: (1) manually define rules that map a given context to an adaptation, or (2) automate this process and let the system infer the relationship using machine learning. Rule-based systems have the advantage that they are typically easier to comprehend and control. However, they require expertise and experience to define good rules, and it is unclear how exactly we can utilize gaze data for manual rules. Automation with a continuously learning system also offers added flexibility, where rules can be personalized on-the-fly and through continued usage. Thus, we chose the second choice, using Machine Learning for layout adaptation. Although, it remained a design consideration that adaptations should not occur randomly since this would likely distract the user and, thus, limit efficiency for their tasks. Instead, an optimal adaptation system triggers the adaptations, ideally so that the user can understand and expect the adaptations after a learning phase.

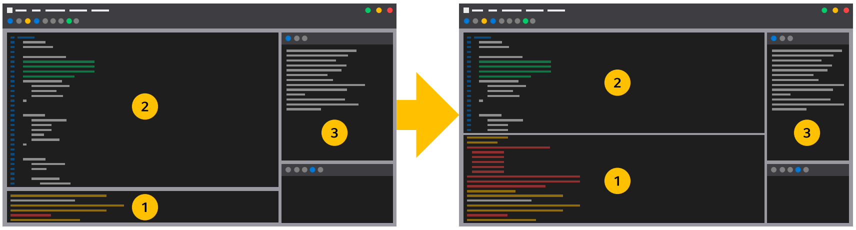


Figure 2.1: Our gaze-based adaptive IDE extension changes the layout of the IDE depending on the developers' intent. For example, when the prediction models determine an increased relevance for the terminal output in the IDE (1), the code editor (2) and other UI elements (3) shrink while the output panel (1) grows in size. This can highlight or reveal otherwise missed information.

2.1.2.2 Data Collection Study

For our adaptive IDE, we focused on a time series of gaze and interaction data, which, given a high enough resolution, give a detailed context of the user's focus of attention and the usage of the IDE, respectively. Since no suitable data set to support such development was publicly available, we first collected a set of data enabling us to train a machine learning model for real-time layout adaptation.

Procedure After welcoming the participating software developers and answering any open questions, we asked them to sign an informed-consent form and fill in a demographics form. Then we invited the software developers to perform a series of well-defined tasks using an instrumented IDE. The tasks engaged the users with the IDE to ensure that our training data was meaningful and not just undirected, idle action. Therefore, we asked them to find and fix bugs in a set of seven sorting algorithms we provided, as this included both reading and writing code across multiple files. We did not give a number of errors in the code, but merely asked the participant to correct the code until it performs as specified, i.e., correctly sorting the input. The task description was always available on a secondary screen. Each participant had 30 minutes to do this. This time limit was chosen based on feedback from previous studies, in an effort to prevent exhaustion, which would negatively affect task performance, attention and thus potentially diminish the value of the gaze data.

Apparatus Participants worked in a well-lit working space with a mouse, a keyboard, and a 27" screen with a 1920 $\times$ 1080 pixels resolution. We used a Tobii Eye Tracker 4C[1] to record the gaze. We calibrated the eye tracker for each participant. With the *iTrace* software [269], we could then match gaze to UI elements in the IDE window. Due to compatibility with the iTrace software, we exclusively worked with Microsoft Visual Studio 2017. To get a richer data set, we also utilized the Visual Studio API to extract the window's name, i.e., UI segment, on which iTrace detected the gaze. Beyond this, we used the Visual Studio API to capture the user's interactions with the IDE in the form of UI events.

Participants Six participants contributed to the data set, yielding 180 minutes of gaze and interaction data with about one million data points. The participants were between 20 and 30 years old and had an academic computer science background and at least a Bachelor's degree, with up to five years of professional software development experience. All of them completed the bug fixing task to completion in the given timeframe.

Recorded Data We preprocessed the raw gaze data from our participants, mainly a standardization of the timestamps. Next, we extracted the gaze fixations; leaving all other gaze points in the data set could lead to information being hidden or highlighted, even though the triggering gaze was only a glance and not intentional attention. For this, we used the algorithm of the PyGaze [59] library to filter these gaze points and transform the raw gaze coordinates into fixation points. With continuous gaze turned into discrete time events, we could then pair each fixation with the corresponding UI events from the IDE. The combination of fixated UI event and actual interactions in the IDE should further reduce the likelihood of unexpected changes. The Visual Studio API constrained the kind of UI events we could collect. It provided us with four event

[1]https://www.tobii.com

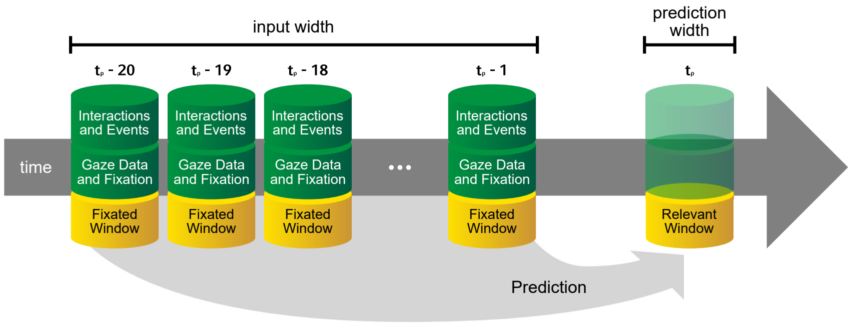


Figure 2.2: Our model uses the last twenty entries (input width) of interaction and gaze data, which are labeled with the fixated UI window, to predict a single step into the future (prediction width) which UI element will likely become relevant[1].

types — Window Event, Command Event, Document Event, and Solution Event — each representing an interaction with the corresponding part of the IDE. Every event itself also holds additional properties and meta-information, for example, the triggering interaction and what element the target was. The Visual Studio API, iTrace [269], and FeedBaG++ [10] allowed us further to enrich each fixation with information from the UI elements. In total, each sample has 168 features values. So, we matched each fixation to one of the following elements:

- The **code editor window** that allows the user to write and change code inside an open file.
- The **output window** where a running program prints out logs and results.
- A window that allows the user to change **project-specific settings**.
- The **file explorer** from which a user can open or move files.
- A Visual Studio window that displays varying content depending on the IDE tools used by the user.

The combination of fixations from the gaze data, the meta-information for its corresponding UI element, and the UI events left us with a data set of about half a million sequential data.

2.1.2.3 Model

We trained a long short-term memory (LSTM) model with this data set to predict a suitable UI adaption using TensorFlow and Keras. We used a window size of 20 samples to predict the next important UI element, see Figure 2.2. While window size can be selected arbitrarily, we used a window size of 20, as it yielded good results and seemed to capture the current task. We trained the LSTM model to output a numerical importance score for each of the five possible panels.

After hyperparameter tuning, we found that three LSTM layers with 32, 64, and 32 neurons, followed by a dense layer as the output layer, performed the best. Moreover, we used a categorical cross-entropy loss function with a softmax activation function on the output layer. Any other parameters were left as their default value. As optimizer, we used the Adam optimizer with a batch size of 32, and we trained the model for 50 epochs. Finally, we used a time-based 70:30 train-validation split.

The final model achieved $\sim 82\%$ accuracy on the validation set. This model may leave room for improvement, but it does provide good results, given our validation set. The number of respective units and the amount of stacked LSTM layers may be altered in the future to optimize the model further.

2.1.2.4 Prototype

To incorporate this model into an IDE, we implemented a Visual Studio extension using the Visual Studio Extension API to access the IDE's internal event logging and the ability to change its behavior, in our case, the size and layout of the panels within the IDE window. A Python backend hooked up via a REST API is responsible for incorporating the Machine Learning model, which decides when and how to change the UI. The expansion forwards any registered user input at the UI level to the backend. The backend uses 20 user inputs to generate a prediction and assess which UI element will be relevant next. The backend returns the prediction to the IDE, which adapts its user interface accordingly.

The whole architecture of our adaptive IDE Visual Studio Extension is illustrated in Figure 2.3. The Visual Studio Extension captures the *Event Stream* similarly to

[1] cf. https://www.tensorflow.org/tutorials/structured_data/time_series

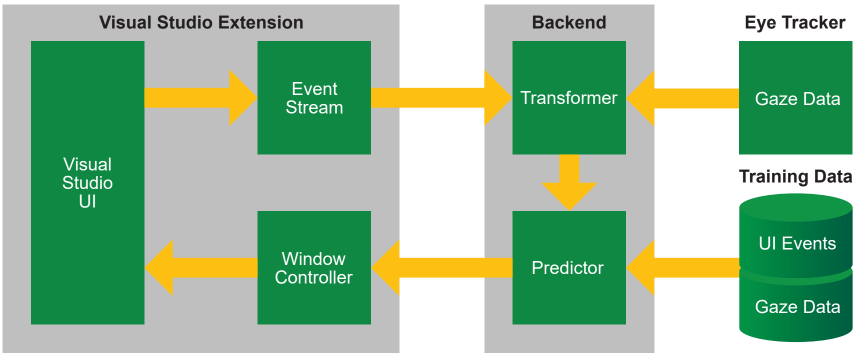


Figure 2.3: The prototype captures events inside the IDE and sends them to a backend via REST API. There the data is preprocessed, and a Machine Learning system predicts what UI elements are likely to be relevant. After sending this prediction back to the IDE, the IDE extensions adapt the UI accordingly, scaling the elements to match their predicted relevance.

the FeedBaG++ tool [10]. The backend *Transformer* preprocesses the data for the Predictor. We pre-trained the *Predictor* with the *Training Data* from the data collection study. Whenever the *Window Controller* receives a prediction, it adapts the UI windows in size; in detail, it extends the default UI elements such that they can be stretched and shrunk at will (as showcased in Figure 2.1). Users can still manually change their size by drag-and-drop, and the *Window Controller* adapts their size relative to this manual preset. So, when the *Predictor* predicts one window of greater relevance, the *Window Controller* stretches this window and shrinks the other windows accordingly. If the prediction is ambiguous, the *Window Controller* promotes the code editor as a fallback solution. The changes made to the window sizes happen instantaneously without animation, so abrupt size changes are possible.

At this stage, even with the limited input data set, the prototype system can capture the user's context of use in real time and forward this information to a trained LSTM model that attempts to predict a window the user might consider relevant for the given context of use, and adapt the IDE's user interface to the predictions accordingly. The whole process takes no more than a few seconds and does not slow down the IDE's performance. While the technical feasibility is a good starting point, feedback from users is necessary to determine whether this technical solution also provides meaningful benefits in practice.

2.1.2.5 System Evaluation

We conducted a qualitative evaluation to assess whether our proof-of-concept already provides enough support that developers notice the adaptations and see their benefits. Since programming in an IDE is a highly creative activity where many ways can lead to a solution, comparability between software developers and programming tasks is often barely possible.

As personal experience with Visual Studio and its many features result in personal preferences, quantitative performance measures are hard to assess. Gajos et al. [88] highlighted this as they attempted to evaluate adaptive UIs quantitatively but found it challenging, while subjective and qualitative measures were more expressive. In contrast, qualitative feedback can still give a good impression of how people view the changes in our adaptive IDE generally, whether they see value in them and how the current prototype can be improved in the future into a production-ready tool. Therefore, we only collected qualitative feedback about the adaptations and their benefits. For this, we invited six new participants to use the adaptive IDE for a small set of tasks, after which we interviewed them about their experience. Additionally, we recorded the internal behavior of the adaptive component to be able to reproduce when and how often UI elements changed during the evaluation.

Procedure After welcoming participants in person, we explained the purpose of the study and answered any open questions. Then we asked them to give written informed consent. Afterward, we asked them to sit down in front of a 27-inch monitor with a 1920 $\times$ 1080 pixel resolution, a keyboard, a mouse, and a Tobii Eye Tracker 4C (as in the data collection study). We presented the task as a Visual Studio project.

We asked participants to work on tasks in a controlled environment using the adaptive IDE with the eye tracker for 30 minutes. During the interaction, the adaptive IDE recorded and analyzed their behavior and adapted the UI accordingly. To avoid confusion, we informed the users of this behavior — after all, when someone would use an adaptive IDE in the wild, they too would know of this feature. The objective of the tasks was to keep the participants busy and interacting with the IDE, thus triggering a constant stream of events. In detail, we asked them to implement an algorithm for searching text and benchmark their code against a provided linear search algorithm. The success of the task itself was not the goal here. Instead, we designed the task to involve various interactions within the IDE, including writing and reading code, executing it, assessing the output, and making adjustments to ensure that different UI adaptations would occur.

After performing this task for 30 minutes, we asked the participants a series of questions in the form of a semi-structured interview. During the interview, participants had the opportunity to provide us with feedback and were encouraged to share their ideas with us for potential improvements.

Interview Guide We wanted to know whether participants noticed any adaptations and how they perceived them. After getting a first impression, we moved on to more precise questions regarding the adaptations that occurred during the task, e.g., did they maintain their state for too long or too short? With a firm grasp of the capabilities of our prototype, we then asked to give specific feedback for our prototype, specifically what changes in the UI were helpful and which were detrimental, which further aspects of the UI they would like to see adapted, and what their expectations are for long-term use. Moving from our prototype to adaptive UIs in general, we also discussed different types of adaptations and methods of personalization in IDEs and how the participants perceived them.

Participants We invited six participants. Like the group for data collection, they all fell in the 20 to 30-year age group, and had at least a bachelor's degree in computer science and prior experience with professional software development. According to their feedback, all felt comfortable with the given code and had no problems with the task. All of them also had previously used an IDE, although not necessarily this version of Visual Studio, and were thus familiar with all the general UI elements which were present in our study.

2.1.2.6 Results

We used affinity diagramming [109] to label and cluster the statements of our participants into common themes and by their value judgment. For example, whether they perceived the IDE adaptation as positive or negative.

All participants noticed adaptations, although some only barely. Some appeared to remember some changes to the UI happening during their coding, and only in hindsight could they attribute those impressions to the adaptive IDE. However, even if they did not notice all of them, the extension logs revealed that each participant had between 5 and 70 adaptations in the 30 minutes. The participants were surprised when we confronted them with the absolute number of adaptations during the 30 minutes task, as they assumed a lower number. However, it is important to note that not all adaptions during the usage were extreme changes to the UI. Sometimes changes were subtle, short-lived, and incremental, which participants can easily miss due to change blindness [282].

The overall impression was fairly neutral when asked about their attitude toward the adaptations. The adaptations did not negatively impact the coding task or distract participants from their workflow. Moreover, participants agreed that adaptations were suitable for a given context. Finally, participants identified that the adaptations were suitable for the context in which they appeared. They could see the value of an adaptive, supporting UI regarding ease of use and efficiency.

Participants stated that to rate the adaptation positively, they needed to perceive the impact of the adaptation more clearly. Thus, the participants concluded that they observed too few adaptations and that a longer period of working with an adaptive IDE would be necessary to gauge their effect better. In contrast, we argue that perceivable adaptations are not desirable, as they most likely will distract from the development task itself.

The participants agreed that the worst-case scenario for adaptations in real-world use would be that they could hide relevant information, since the system erroneously considers other information to be more important. Fortunately, during our evaluation, this only occurred in a single instance. Here, the output window overlapped the code editor, hindering them from proceeding with coding. However, the participant fixed the issue quickly by adjusting the window layout. To account for such issues, another participant noted that a useful constraint could be to have the code visible at all times.

We also asked whether our participants would prefer smooth, animated transitions or more abrupt changes. Here, everyone except one participant agreed that they preferred the transition-less adaptations. They argued that animations could attract too much attention. Beyond our initial IDE adaptation, participants positively perceived UI adaptations but remained skeptical towards adaptations that influence the system's functionality. Many categorically stated that they did not want this adaptation in their IDEs but remained open to less intrusive adaptations like the layout adaptation. One participant was only willing to try adapting the functionality when extensive feedback and intervention mechanisms were in place to counteract the adaptations when necessary. The wish for more control and transparency was a common theme overall. One of the participants mentioned preferences with user-based priority weighting for supported windows. Several participants mentioned that predefined rules could also lead to better adaptations.

In summary, all participants showed interest in an adaptive IDE that has more to offer than what this proof of concept was capable of delivering. Everybody said they would try an adaptive IDE that highlights and hides information to see how it affects their work in the long run.

2.1.3 Discussion

In this chapter, we discuss our findings with a focus on two key aspects: (1) our experience with eye tracking as a data basis for our implementation and the resulting learnings from implementing an adaptive IDE, and (2) the feedback from the users during the evaluation.

2.1.3.1 Gaze as Data-Basis for an Adaptive UI

We showcased the viability of using gaze data for an adaptive UI. Our prototype and the subsequent evaluation show that a functioning, adaptive UI is possible and even with the limited data set provides adequate adaptions. Our LSTM model could define and recognize usage patterns from the fixations. Our evaluation also showed that these patterns, e.g., highlighting of the output window, differed between the users, allowing for a personalized experience. The users, in turn, considered the adaptations suitable and unobtrusive for their context. Nevertheless, our implementation provided only a limited degree of adaptation, highlighting some panels and UI elements for users. However, this is a natural consequence of the relatively small data set we were able to use as input data. An industrial solution with wide adoption could utilize a more extensive initial set and incrementally increase its performance through online learning and real-world usage. On the one hand, it would allow training more complex models, yielding better accuracy. It would also need to support various use cases. In our data-collection and evaluation, we focused only on searching and sorting algorithms for which participants had to write or update code. While the adaptions we trained from one task and showed in another one were perceived as adequate, these task can already offer a wide variety. Of course, software development encompasses even more tasks and use cases and how people use their IDE can depend on factors like the programming language or the phase in the development life cycle. Furthermore, the flexibility of IDEs allows for broad personalization beyond the fairly fixed state in our study. While most people will likely not completely alter the layout of their IDE for every task, it still highlights the challenge of tools as complex as IDEs and the broad range of starting conditions an adaptive UI will be able to cope with. So, for practical use and wider adoption, the underlying model needs to be expanded, which also requires additional data. The fact that even our small data set was able to generate meaningful adaptions, though, is promising, particularly for a personalized version of such an adaptive IDE, i.e., a use case where no large amounts of data will be available. Once a developer has configured their IDE, it would seem that training and personalization can happen quickly. particularly when using a combined online and offline learning approach [C1]. The fact that a small data set was sufficient to show meaningful results is also relevant for future studies, where one might investigate other data sources. Not having to collect a large volume of data means that we can, in the future, quickly prototype variations of an adaptive IDE and evaluate them to get a better understanding which types of data are beneficial for timely and meaningful adaptations.

Window Size As we learned during our work, individual gaze points or UI events can only be one part of such a data set, though, and are barely suitable in isolation. For example, when developers are searching for a solution or just thinking, their gaze may wander across the screen. Thus, interactions and gaze points may fluctuate over a short period. These fluctuations offer limited value compared to the larger context and longer periods of previous interactions. Thus, as was apparent during our work, it becomes a challenge to choose the right degree of granularity for both input data and adaptations to ensure that the changes in the UI are timely and context-sensitive but not overly sensitive and thus over-steering. In our prototype, we chose a window of 20 samples to aggregate a time series, which worked well for our use case. The exact effect of this aggregation window, i.e., whether larger or smaller frames would improve the adaptation quality, remains an open question, which warrants further investigation. The same goes for the variety of data, particularly how more data source like gaze will increase the quality or whether we will reach a point of diminishing returns.

Improving the Performance Further Beyond this, our prototype also highlighted some issues with gaze as one such data source and the resulting adaptations. The most notable of these is a consequence of how we look at the world: while we may be cognitively focused on one thing, our eyes and gaze may wander, sometimes erratically. Furthermore, sudden jumps of the eyes are much more likely than for other input modalities, e.g., the cursor. Not only does this complicate predictions, but since patterns may be interrupted by completely irrelevant jumps, it also affects the resulting adaptations. If the input jumps frequently and these jumps affect the adaptations in the short term, the system may recommend barely noticeable adaptations because they are immediately overwritten and reset. In the worst case, this could lead to a jittery, flickering UI. Therefore, using a windowing approach is necessary. Moreover, our LSTM network architecture helps to reduce jittering.

Gaze and events are valuable sources on their own, and showcasing their feasibility for prediction is the core of this research. However, other data sources, such as the mouse, will be of value in improving the prediction, cf. [48, 182, 335]. On the other hand, expert users rely less on the mouse and instead use keyboard shortcuts. In addition, mouse movement tends to be very individual, c.f. [341]. Thus, in line with prior work Zhang, we argue that mouse movements will support the prediction, but a successful prediction will only be possible with gaze data. However, mouse data, in addition to gaze data, will further stabilize the prediction.

Comparison to Rule-Bases Adaptation Rule-based adaptations could work around the challenges of fully autonomous adaptation. However, the granularity of gaze data makes it unfeasible to prescribe rules on the low-level data points manually. Automated classification methods and aggregation in conjunction with high-level rules that utilize the classified context could provide a hybrid solution. The human involvement then checks whether the aggregation works to determine the correct context. In such a scenario, users could then use this context to specify in which situations they want what changes and the system would primarily determine whether the conditions for an adaptation are met. We support such a hybrid approach, with feedback during our evaluation regarding rule-based, which was fairly positive. It would give additional control to the user, who can enable, disable, and change individual rules to suit their personal preferences. Increased user control could also facilitate a greater understanding of the adaptations, which prevents confusion due to sudden UI changes.

2.1.3.2 User Feedback

The feedback from our evaluation was primarily positive. Since all participants could perform the tasks we asked them to do, our prototype and its adaptions did not prove to be a hindrance. Our participants did not perceive the adaptive UI as distracting or interrupting.

Based on the feedback from the evaluation study, we conclude that implicit data sources like gaze can provide additional usage context and help with predictions. Nevertheless, their implicit nature makes them hard to control and understand for users. Therefore, it is essential to balance user control and automation. Thus, we argue that a hybrid system where automation determines the context and user feedback defines the behavior might foster enough trust that fully autonomous adaptation gains long-term adoption. This aspect of developers wanting to maintain control of tools will likely also apply to further AI-powered automation and thus provides insights towards RQ1 beyond just adaptive interfaces.

An objective performance metric certainly would be desirable, but this is challenging with a creative activity like programming and an artificial, short task in the lab. Long-term field observations may yield more insights into how adaptions in the IDE affect working behavior. For the subjective assessment of their performance, however, they did consider themselves to perform at least as good with the adaptions as without, so they proved not to be detrimental and potentially beneficial.

Longitudinal evaluations would offer further insights into which adaptions are most valuable. If adaptions have sufficiently impactful that they change the work practice of developers, we would also investigate how these behavioral changes interact with the adaptations, i.e., whether the initially trained adaptions remain adequate or whether the system needs to continuously improved, evolving adaptions and developer behavior alongside each other. Further, we chose only to investigate information dimming (adapting the UI layout), but alternative adaptations are possible with our setup. Such studies inform the decision of which aspects of the adaptions can be fully automated and which ones users wish to control and manually personalize. Here, we see two options: (a) the rule-based adaptations, where users explicitly define the system's behavior in a given context, and (b) continuous human-in-the-loop feedback, where users provide immediate feedback when an adaption occurs.

However, we must keep in mind that the target group for an adaptive IDE, i.e., software developers, also does not represent the general public. It is very plausible to assume that those that develop software have a certain inclination to control it. They also have a higher degree of expertise in defining software behavior, so writing rules that map from usage context to system changes may be straightforward for them but not the average user. So, how these findings generalize beyond the context of our work will need to be tested, e.g. in similarly complex software where eye tracking already has shown potential like multimedia editing tools [233].

2.2 Software Development with Large Language Models

This section is based on the following publication:

Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. "Significant Productivity Gains through Programming with Large Language Models." In: *Proc. ACM Hum.-Comput. Interact.* 8.EICS (June 2024). DOI: 10.1145/3661145

While AI excels in detecting patterns in data, it is widely and actively used for far more than just that. Particularly generative AI (GenAI) systems not only detect patterns in data, but are also able to generate new content based on them. Recent advances in this field, particularly with Large Language Models (LLMs), have greatly boosted what is possible in terms of code generation. With this technology, there has been an abudance of novel tools. We will investigate RQ2 by evaluating the effect that some of these tools have on developer productivity.

GenAI and LLMs have been applied in various areas, including applications of GenAI for software development, e.g., for generating test cases, documentation [296, 297], but most notably to generate code from natural language descriptions. While this can be done general LLMs like the family of GPT models [31], there are also dedicated models specifically tailored for this use case [31, 46, 50, 53, 84, 169]. Thus, code generation with LLMs represents an prominent and popular example how AI-powered development tools can support developers.

Given the ability of these tools to generate a complex piece of software from an adequate natural language prompt almost instantly, they clearly have the potentially to greatly boost development productivity. Developers can access these models through a wide variety of modalities, from a conversational format like ChatGPT [239, 257], integrated into their familiar coding environment like GitHub's Copilot [19, 137, 196, 343] or even with new tools, built around these models, like the Cursor Code Editor[1]. These different tools use a variety of different interaction patterns for accessing LLMs, ranging from an auto-complete approach, as is already common in development tools, to a conversational style, as found in modern chatbot systems. GitHub Copilot recently also extended its functionality to offer both variants for the same underlying models [340].

[1]https://www.cursor.so

While the underlying model may be the same, the different forms of interactions play an important role from the human perspective and can have wide-ranging effects, from how efficiently these tools can be used to what code is generated in actuality. For example, the type of interaction affects how prompts may be written, which impacts which information is provided to the LLM and, thus, what the responses look like. Thus, the presentation and interaction are essential components to ensure that developers can use the LLMs productively in their work [19, 137, 343]. However, in a field that is evolving as this, the development of new tools is often driven through a technology-focused perspective, while the human perspective lacks systematic analysis and is instead frequently supported more by anecdotal evidence.

To advance our understanding of how human-centered aspects like interaction design affects productivity in this domain, we compare two AI-supported approaches to traditional coding supported by searching the internet using a web browser (*Baseline*). Thus we refine RQ2 as follows:

> **RQ 2.1:**
> How do different interaction patterns of LLM-powered code generation contribute to developer productivity?

In detail, we compare an AI-supported auto-complete interface using GitHub Copilot (*Auto-complete*) and a conversational system using GPT-3 (*Conversational*) against a baseline with only conventional support mechanisms. In a within-subjects study ($N = 24$), we determined how the support mechanisms affect developer productivity and satisfaction, and how much they rely on these support mechanisms in different scenarios. To assess their impact, we asked participants to complete three Python programming tasks, with AI support and without.

Our within-subject study with 24 participants demonstrates the benefit of using AI assistants and how they can benefit the different aspects of productivity. With participants in our study showing distinctive usage patterns, it furthermore showcases how AI is integrated and which form of interaction it allows, which can greatly affect how it is used. This demonstrates that the success of AI-support mechanisms does not just rely on the underlying technical implementation but is greatly affected by interface and interaction design. Thus, our contribution is twofold: (1) we offer insights into how

AI coding assistants in their current state of development affect software developers and their productivity, and (2) we do this using a more holistic perspective on this area, taking into account a diverse set of factors that contribute to overall productivity. We hope that this broader perspective will be adopted in future investigations in this highly dynamic research field, thus allowing us to gain a better, more rounded understanding of how the continuing development of AI assistants has affected and will affect humans.

2.2.1 Related Work

As the previous section and many other examples demonstrate, software development tools have evolved far beyond just being input-output programs. One quite ubiquitous feature in many modern editors is simple code completion [199], which suggests what code the developer might type next. Over the years, researchers and industry have explored different mechanisms how these suggestions and code completions can be generated, from semantic analysis of the source code [16, 125, 146, 249, 250] to machine learning approaches [238, 291] and different combinations [32, 243]. While these systems can play an important role in productive development [146], there is still limited research regarding their impact on developer behavior [138].

Recent advancements in the area of Large Language Models (LLMs) have furthermore enabled new means not just to support developers with suggestions, but to generate larger pieces of software from simple, natural language prompts. This will likely impact and change how developers use code completion. Thus, our work explores how these modern AI tools affect developer behavior, specifically productivity. To this end, we will first briefly outline some of the work on AI code generation and how it produced modern coding assistants. We will then summarize different aspects of developer productivity and highlight literature that explores how these AI support mechanisms affect them.

2.2.1.1 AI-Assisted Programming

Automated code completion is a technology that has been used in one way or another for many years. While single statement auto-complete with syntactically valid expressions is likely the most common way, e.g., Microsoft's IntelliSense, there are other forms as well, which can generate larger code snippets. Frequently, these relied on a clear specification in a format like state machines [75].

Recent developments in Machine Learning and Natural Language Processing have led to a type of code generation tool that allows more flexibility and the use of natural language to describe the software's behavior. These systems rely on the Transformer architecture [302] and resulting Large Language Models like Generative Pre-Trained (GPT) Transformers [31] or Bidirectional Encoder Representations from Transformers (BERT) systems [71]. These systems are pre-trained with a large volume of data, which may include natural language text, code, etc., which shapes for which tasks these systems can be used [263]. The pre-trained system can then be fine-tuned for specific tasks [127, 240, 263]. By prompting a pre-trained system with specific requests, a user can then use such a system to generate desirable outputs, e.g., in our use case, pieces of source code. These systems are already capable of producing code, often at least on par with what human developers can create [57].

In the last years, a number of these systems have become widely available, e.g., GPT-3 [31], GPT-4, etc. and their popular chat-based interface ChatGPT[1], Codex [46], a GPT-3 based system specializing in source code, which GitHub Copilot also uses, and many more [50, 53, 84, 169]. These examples also demonstrate that different interaction paradigms are viable with these systems: ChatGPT uses an interface where users prompt input to the system, and its output is presented as a chat. Using the Codex system, GitHub Copilot can directly integrate into coding editors like Visual Studio Code or IDEs [330]. While these types of presentations have a negligible impact on the system's behavior and output, the user interface may affect how accessible it is to the user and how it is used. This can have downstream effects, which still affect the software that is created with them.

We have found several evaluation efforts to assess the quality of the generated software (e.g. [5, 46, 174, 333]). Many of those, however, focus on purely technical quality, typically how much of the code a model generates satisfies the requirements. While software development remains an exercise that involves humans, as developers, end users, etc., these metrics alone will not suffice. Instead, we must also consider human

[1] https://openai.com/chatgpt

factors, e.g., how productively a software developer can work with these tools [121]. This is echoed by Bird et al. [27]. They highlight how rapidly the underlying models can change but emphasize that the activities of humans also change and require continued evaluation as this field evolves. Their findings are based on an earlier version of GitHub Copilot. Using different methods, they found improvements in perceived productivity and efficiency with early adopters and first-time users.

2.2.1.2 Quantifying Productivity

Quantifying how productive programmers are during software development is useful not just for research but also, for example, for project management or to collaborate efficiently. Naturally, there is a wide body of research on different metrics and aspects that help to gauge developer productivity.

A fairly simple but common way is to use the *Lines of Code* (LoC), which a developer creates, deletes, or modifies, as a proxy for productivity [231]. As Petersen [231] point out, though, this metric requires some context to be useful since LoC can widely vary due to other factors, e.g., programming language and programming style. Therefore, it is essential to very clearly define what counts as an LoC and the inferred productivity. Petersen [231] and Hernández-López et al. [119] describe a second approach using *Function Points*. Function Points quantify the functionality a piece of software delivers and consider the complexity of inputs, outputs, internal files, and external interfaces. While they consider more context than LoC, it becomes more challenging to accurately and consistently assign Function Points [231].

Recently, Oliveira et al. [213] reviewed the use of additional metrics. These included *Lines of Code by Time*, which puts the absolute number of LoC into context, the *Halstead Effort by Time*, which assesses the complexity of the code the developer works with, and the *Code Owned by Time*, which signifies how much a developer contributed to a project's code base. In larger software projects that use version control, the commit history becomes an additional source of information. Oliveira et al. [213] also analyzed commit based metrics, like *Commits by Time*, the *Committed Lines of Code by Time*, and the *Committed Characters by Time*. Based on their analysis and feedback from team leaders in software development projects, *Lines of Code by Time* and *Code Ownership by Time* correlate strongly to the subjective perception of the project team leaders.

The importance of time-relative metrics matches with the findings of a literature review by Hernández-López et al. [119] who found that *Lines of Code by Time* or *Functions Points by Time*, or *Tasks Completed by Time* in long term projects, are commonly used metrics. In their review, they also point out that some contextual aspects are still rarely considered. They argue that metrics like “customer satisfaction, worker responsibility, task importance, perceived productivity, and absenteeism should be included in productivity measurement” [119].

Beyond these code-centric metrics, Storey et al. [287] emphasizes the importance of developer satisfaction, and Meyer et al. [187] emphasize the focus on developers' subjective perception. Thus, they conducted a survey with developers on which code metrics they found to be most useful. The feedback indicated that some of the time-based metrics like *Number of Work Items (Tasks, Bugs) Closed*, *Time Spend on Each Work Item Time Spend Writing Code* are considered important but also a number of metrics focusing on collaboration like *Time Spend Reviewing Code Number of Code Reviews Contributed To*, and *Time Spent in Meetings*.

Forsgren et al. [87] combines several perspectives and advocates for a holistic approach to measuring productivity. They propose a framework for productivity measurement using five categories: *Satisfaction*, *Performance*, *Activity*, *Communication*, and *Efficiency*. This *“SPACE”* framework does not dictate specific scales or metrics for each of these categories, though. The goal of the *Satisfaction* dimension is to include the users' subjective perception, for example, developers. *Performance* ensures that code quality is included, since just because developers write a lot, they “may not be producing high-quality code” [87]. Static analysis of the code base is one way to quantify this. *Activity* is the “count of actions or outputs completed in the course of performing work” [87]. Depending on the context and the requirements, this can be recorded at different levels of granularity, from individual actions in an interface to commits and also to larger, project-level tasks. Most software projects happen in collaboration, so *Communication* includes how effectively information different contributors interact with each other. Lastly, *Efficiency* puts the development efforts into time context and considers how distractions, interruptions, etc., affect productivity.

2.2.1.3 Productivity of AI-Assisted Programming

In the last few years, particularly since LLMs have become easily accessible, the use of AI as a support mechanism in software development projects has increased. Consequently, there are also a number of studies that investigate how AI assistants impact developer productivity.

In the domain of auto-complete systems, Peng et al. [229] conducted a between-groups experiment where participants had to create an HTTP server using JavaScript. One of the groups used GitHub Copilot, while the other group had no access to AI support. Using Copilot led to a significant reduction in task completion time by more than half. Participants noted the increase in speed but estimated it to be only about 35%. Furthermore, while the control group took a bit longer, the task success rate was not significantly different. In contrast, Vaithilingam et al. [298] found that participants failed to complete tasks more often using Copilot compared to just using IntelliSense, and it did not have a significant impact on task completion time. One reason for this appears to be an overreliance on the AI so that erroneous solutions were accepted without a thorough review, which led to "time-consuming debugging" [298]. On a larger scale, an analysis of the usage data of 2047 GitHub Copilot users [343] showed that the subjective perception of productivity is primarily influenced by the number of accepted suggestions, not necessarily how valuable or lasting they are. The pure acceptance rate, however, is influenced by other factors, such as a user's work context, with suggestions during non-working hours being less likely to be accepted. It appears that in a professional context, users are focusing more on speed, which can lead to the aforementioned quality issues.

In contrast to the above-discussed auto-complete systems, such as GitHub Copilot, we see an additional trend of support, which is rooted in conversational systems. For example, Qureshi [239] investigated the use of ChatGPT as a support mechanism in a study with 24 students. They asked them to complete group exercises for an undergraduate computer science course on “Data Structures and Algorithms” [239]. Half of the students were given access to ChatGPT, while the other half had to solve them without. While the students with AI assistance were able to achieve significantly better scores, there was a shift in tasks, with more time being spent on debugging.

With GitHub now offering both types of systems, GitHub Copilot and GitHub Copilot Chat, they are in a position to provide additional insights from their customer data and evaluations. According to that, they claim a significant increase in speed of about 55% using Copilot in the auto-completion form [141]. Rodriguez [256] provided no information on task completion time but reported that AI assistance led to faster code reviews using the chat variant. Considering broader aspects of productivity, between the two reports, they saw an increase of satisfaction, from $\sim$60–70% using auto-completion [141] to 88% using the chat [256]

These studies demonstrate that the various activities, aspects, and the perception of productivity are all differently affected by the use of AI assistants. Table 2.1 provides an overview of these and further studies, which study various aspects of productivity in AI-assisted programming. From our findings in the literature, we concluded that most research so far has focused on individual AI assistants. Ross et al. [257] emphasize that the different forms of AI assistants will each have different trade-offs for productivity, though. Furthermore, prior work [19, 137, 343] also suggests that comparing the different forms of AI assistants could lead to valuable insights into how productivity is affected.

The state of the literature thus paints a heterogeneous picture of different AI assistants affecting productivity, sometimes more and sometimes less. Given the fast-paced nature of the field, this is unsurprising, since technical advancements and the enthusiasm they can generate will continue to affect how these systems are perceived and how effectively users utilize them to increase productivity. Different studies from prior work also had to rely on older versions of coding assistants and underlying models, as they were available at the time. Thus, our work contributes to an updated understanding of these tools. Furthermore, the heterogeneity also stems from the fact that different studies focus on different types of assistance and aspects of productivity. At this point, we contribute insights into the current state of AI-assisted software development from a broader perspective using the SPACE framework. A continued understanding of the human factors will be essential to steer the field in the right direction. Still, empirical findings will always present only a snapshot of the current time, and any technical improvements of these systems in the future will require continued efforts to make them useful and valuable to their users.

2.2.2 Methods

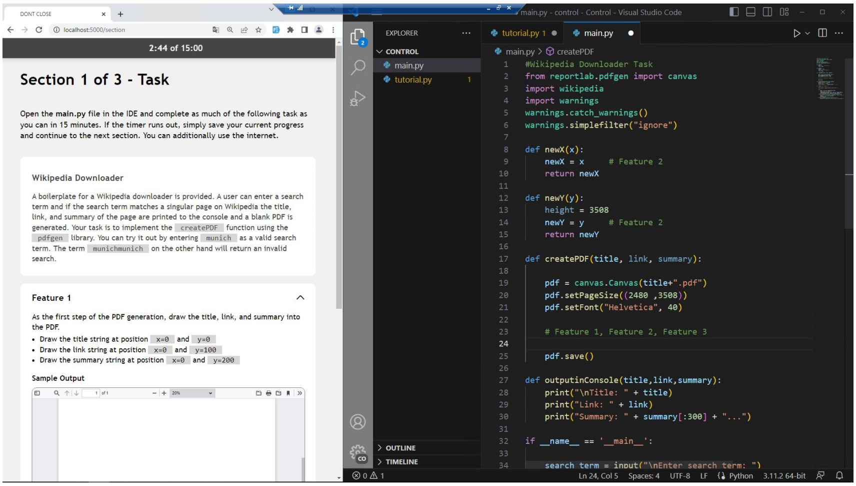
To investigate the effect of different forms of AI assistants, we conducted a within-subject user study to compare different variants of LLM-supported programming. In total, we compare three levels of SUPPORTMECHANISMS: a *Baseline* without AI support, *Auto-complete* using Github Copilot[1], and *Conversational* support using chatbot-like interactions with GPT-3[2]. We used VSC extensions to enable direct support, see Figure 2.4. In each condition, we allowed participants to use the browser to search for additional information online, as this is common practice and makes for a more realistic scenario.

2.2.2.1 Participants

To determine the proper benefits of AI assistants for programming, we recruited participants with programming expertise. During recruiting, we aimed for a diverse set of participants to keep the ecological validity high. For this reason, we recruited through a variety of channels, including social media, mailing lists of three institutions working in the area of computer science and software engineering, and by directly contacting industry professionals from these fields. The call for participants contained information about the purpose of the study and that only basic knowledge is sufficient to participate in the study, further ensuring that we recruit a diverse sample and not only experts.

We only required participants who have basic knowledge of the Python programming language to participate in the study. General knowledge of the syntax was sufficient, though. We selected the Python language not only because it is one of the most popular, general-purpose programming language[3]. Additionally, it is frequently used in introductory programming lectures or for occasional scripting. Due to this, many

[1] https://marketplace.visualstudio.com/items?itemName=GitHub.copilot
[2] via https://marketplace.visualstudio.com/items?itemName=genieai.chatgpt-vscode
[3] https://www.tiobe.com/tiobe-index/



Section 1 of 3 - Task

Open the **main.py** file in the IDE and complete as much of the following task as you can in 15 minutes. If the timer runs out, simply save your current progress and continue to the next section. You can additionally use the internet.

Wikipedia Downloader

A boilerplate for a Wikipedia downloader is provided. A user can enter a search term and if the search term matches a singular page on Wikipedia the title, link, and summary of the page are printed to the console and a blank PDF is generated. Your task is to implement the `createPDF` function using the `pdfgen` library. You can try it out by entering `munich` as a valid search term. The term `munichmunich` on the other hand will return an invalid search.

Feature 1

As the first step of the PDF generation, draw the title, link, and summary into the PDF.

- Draw the title string at position `x=0` and `y=0`
- Draw the link string at position `x=0` and `y=100`
- Draw the summary string at position `x=0` and `y=200`

Sample Output

```python
#Wikipedia Downloader Task
from reportlab.pdfgen import canvas
import wikipedia
import warnings
warnings.catch_warnings()
warnings.simplefilter("ignore")

def newX(x):
    newX = x      # Feature 2
    return newX

def newY(y):
    height = 3508
    newY = y      # Feature 2
    return newY

def createPDF(title, link, summary):

    pdf = canvas.Canvas(title+".pdf")
    pdf.setPageSize((2480 ,3508))
    pdf.setFont("Helvetica", 40)

    # Feature 1, Feature 2, Feature 3

    pdf.save()

def outputinConsole(title,link,summary):
    print("\nTitle: " + title)
    print("Link: " + link)
    print("Summary: " + summary[:300] + "...")

if __name__ == '__main__':

    search term = input("\nEnter search term: ")
```

Figure 2.4: Instructions were displayed in the web browser (left) and needed to be completed in Visual Studio Code (right). Visual Studio Code was automatically configured for each study condition to include the correct level of support mechanisms.

programmers of any skill level are generally familiar with the language, even if they do not use it daily or in large software projects. While the invitation for the study mentioned the use of AI for programming, knowledge or experience with any of the AI assistants used in the study was explicitly not required.

From this, we got 29 people who participated in our study. We had to exclude five participants as they either did not complete any of the programming tasks or violated the constraints of the study. For these cases, we note that they used ChatGPT in the browser in our baseline condition, hinting that they believe ChatGPT can enhance their skills. Of the remaining 24 participants, four identified as female, while the remainder identified as male, with an average age of 26.8 years (SD: 3.6). Regarding their education, five participants had a completed Master's degree, 16 completed a Bachelor's program, and three were currently pursuing their first university degree. Nine participants answered that they were currently involved with software development in a

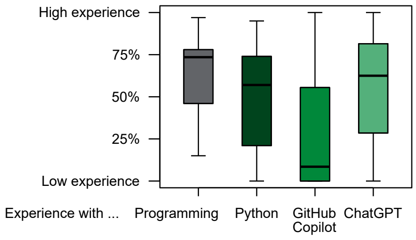


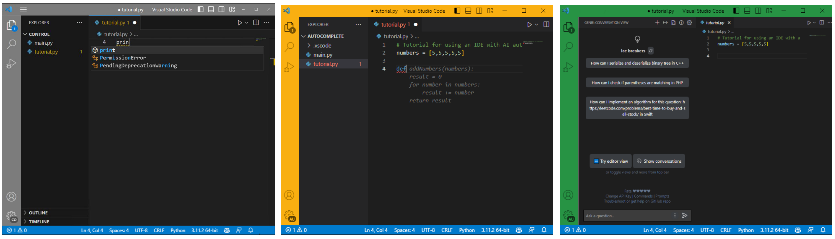
Figure 2.5: When rating their experience on a 100-point scale, participants showed a wide range of experience in general programming and using AI assistants.

professional capacity. The other participants had coding experience through their study program and personal projects. Thus, we gathered a cross-section of potential users for programming assistants.

Consequently, according to their self-assessment in the survey, participants also covered a wide spectrum of prior knowledge, as displayed in Figure 2.5. This is especially the case for the AI assistant tools, where the group covered the whole spectrum for LLMs as chatbots, like ChatGPT, and in GitHub Copilot.

2.2.2.2 Apparatus

We conducted this study online. By pre-installing and running the software on an online virtual machine (VM), we had a common environment with consistent starting conditions for all participants. The VM was provisioned with ten vCores and 18 GB of RAM to ensure smooth interactions. Participants accessed this setup from their own computers, which opened the study to more than just people in the vicinity. We asked them to use a 1080p monitor and, before the study, verified that the interaction via remote desktop worked. We ran the Windows operating system on the machine with Visual Studio Code (VSC) and the Chrome web browser pre-installed. Upon connecting via a remote desktop connection, we displayed instructions using the web browser. While participants followed these instructions, a background script automatically opened a Visual Studio Code instance whenever participants had to use it side-by-side with the instructions, see Figure 2.4. These VSC instances were configured in such a way


EXPLORER
CONTROL
main.py
tutorial.py
OUTLINE
TIMELINE
AUTOCOMPLETE
.vscode
GENIE: CONVERSATION VIEW
Ice breakers
How can I serialize and deserialize binary tree in C++
How can I check if parentheses are matching in PHP
How can I implement an algorithm for this question: https://leetcode.com/problems/best-time-to-buy-and-sell-stock/ in Swift
Try editor view
Show conversations
or toggle views and more from top bar
Rate
Change API Key | Commands | Prompts
Troubleshoot or get help on GitHub repo
Ask a question...
Ln 4, Col 4
Spaces: 4
UTF-8
CRLF
Python
3.11.2 64-bit

```
prin
print
PermissionError
PendingDeprecationWarning
```

```
# Tutorial for using an IDE with AI aut
numbers = [5,5,5,5,5]

def addNumbers(numbers):
    result = 0
    for number in numbers:
        result += number
    return result
```

(a) Baseline without AI assistant **(b) Auto-complete interface** **(c) Conversational interface**

Figure 2.6: The interfaces for the three different study conditions. The correct extensions, files, and context for Visual Studio Code were automatically loaded for each.

that only the necessary files for each task were accessible, and only those extensions were available, which facilitated the type of AI assistance for each study condition (see Figure 2.6).

In the *Baseline (browser only)* without AI assistant, we presented users with a pure VSC without extensions (see Figure 2.6a) and the browser (Google Chrome) to the side. Thus, the baseline replicates a standard setup where users use the World Wide Web to understand the context and potential error messages. This allowed, for instance, looking up information on Stack Overflow, a typical development behavior [2, 324].

The *Auto-complete (Copilot)* condition uses the GitHub Copilot extension for the auto-completion suggestions of code snippets as the user types in place, i.e., directly behind the cursor (see Figure 2.6b). Users can then choose to accept the suggestion, which is inserted into the editor. Alternatively, users simply continue typing their code when they reject the suggestion.

The *Conversational (chatbot))* condition uses a chat panel integrated into VSC (see Figure 2.6c). However, it has some additional contextual interactions, e.g., users can select code and directly send it to the chat for further inquiry. Likewise, code generated in the chat interface can be inserted with the click of a button. Working with GPT3 models under the hood, both systems are subject to the limitations of these models, particularly regarding how much contextual token can be considered for newly generated output.

The surveys between the test conditions were displayed in the browser between task descriptions. Aside from the answers to the survey questions, additional interaction data was recorded automatically, and we screen-recorded the whole study. Additionally, to catch any errors and facilitate observations, we supervised each study participant via screen sharing but kept interaction with the participants to a minimum.

2.2.2.3 Procedure

Upon connecting to the remote machine, participants saw an introductory video that explained the study procedure and its goal. This was followed by a short survey, which included the consent form, demographic information questions, and prior knowledge about programming questions. Additionally, we used this step to explain the study setup generally. After this, participants worked through our three *support mechanisms* conditions (*Baseline*, *Auto-complete*, and *Conversational*), each with a different programming task. Each task was preceded by a video explanation of the tasks and the tools used in the respective conditions. The sequence and combination of task and *support mechanism* were systematically varied using the Latin Square method to reduce effects of order or by specific tasks.

For each programming task, participants first watched a short video tutorial and could code along with this tutorial. These videos briefly explained the tool's functionality, which participants would use, and allowed them to get familiar with it. For this phase, we did not set a time limit so that participants could complete it at their own pace. After this, we showed the actual task, for which participants then had 15 minutes to complete it. After these 15 minutes, participants filled out a survey for this specific study condition.

Upon completing all three conditions, we asked participants to complete a final survey about their overall impression of the different tools used in the study. For their participation, they were compensated with an equivalent of 10 US$ per hour.

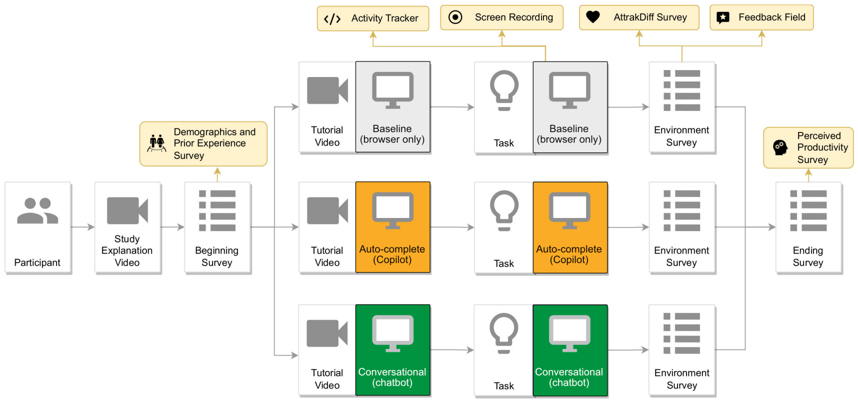


Figure 2.7: Following an introductory survey, each participant performed three tasks and provided feedback. These tasks were permuted by the Latin Square method. Afterward, the study was completed with a final survey about overall impressions.

2.2.2.4 Tasks

To assess these three levels of support, we selected three tasks considering literature in this domain [19, 57, 196, 283, 298, 327] and popular software projects[1]):

Task A **Text Analysis:** For the text analysis task, participants had to perform text analysis of a scraped website, extracting the number of words, unique words, and average word length. We then asked them to produce a list of stopwords and filter the website's content, after which they should calculate a word count to determine the most used words in the text.

Task B **CSV Transformation:** The *CSV transformation* task requires participants to read and write a CSV file. The goal was to process and transform car production data from Wikipedia in multiple steps, and calculate new information from this data.

Task C **PDF Editing:** In the final task, participants were asked to generate a PDF from a template and add text to it, which required some calculations for proper text formatting and placement.

[1] cf. https://www.w3resource.com/, https://github.com/Python-World/python-mini-projects

We iterated the type of task, their subtasks, and their wording during a pilot study phase with three users of different skill levels until they considered them to be similarly complex and to require roughly the same amount of time to complete. The pre-test also showed that it would be possible to complete the tasks entirely by providing the requirements to an AI assistant. For this reason, we split the instructions into smaller sub-tasks of roughly equal size, some of which would build upon earlier ones. Since this enforced some sequence, it also ensured that participants would not pick only requirements that they considered easy to implement, thus skewing the number of features they could implement in the time frame. Furthermore, we provided an initial code snippet, which participants were required to use and extend. While it would still be possible to complete these tasks with AI assistance with few prompts, this setup encouraged participants to engage more with the code and the instructions.

The relatively guided nature of the tutorial, of course, meant that the degree of freedom and creativity to solve the tasks was reduced. Typically, a key part of programming is to decide how to decompose a larger problem into smaller sub-tasks. While the study design reduced this aspect, using equivalent tutorials in all study conditions meant that it was equally reduced. Thus, we consider the comparison between the study conditions to be valid.

While we constructed these tasks to be similar, they are not identical, and different solutions to them may require different amounts of code and time. Thus, we permuted the combination of tool and task using the Latin Square method for a balanced combination and order of SUPPORTMECHANISMS and tasks across all participants.

2.2.2.5 Recorded Data

To assess developer productivity, we use the five dimensions of the *SPACE* framework [87]. For each of its dimensions, we collected both data from the interactions directly and subjective feedback through the surveys.

Satisfaction Participants completed the short AttrakDiff [86, 113, 114] after each test condition to quantify satisfaction. This survey contains ten questions resulting in measures for*pragmatic quality*, *hedonic quality*, and *attractiveness*, which are closely related to user satisfaction [113]. In addition, participants provided additional feedback about their perceived satisfaction across all three environments at the end using a survey with agree-disagree Likert scales as suggested by Forsgren et al. [87].

Performance Hernández-López et al. [119] call for the inclusion of quality metrics into a holistic assessment of productivity, particularly including metrics for output quality. For this reason, we include the correctness of the code that participants created during the task, which corresponds to the *number of work items (bugs, tasks) closed* [187] and the *absence of bugs* [87]. Specifically, we test whether each implemented feature performs as expected using pre-defined unit tests. From this, we calculate the percentage of correctly implemented features out of all implemented features. Furthermore, we include the *Maintainability Index* [215, 216] as a metric, which combines multiple static code metrics, like Cyclomatic Complexity and Halstead Volume [215, 320]. Similar to the previous dimension, participants also provided an additional self-assessment of how they rate the quality of the code they produced after completing all tasks.

Activity Developer activity typically is a measure of working hours, produced code, or number of commits. Given the constrained nature of a user study, these metrics are not feasible. Instead, we looked at actions at a finer granularity, namely the number of individual inputs, e.g., typed characters. While breaking down complex interaction patterns into individual interactions is a well-established pattern [41], prior work (e.g., Forsgren et al. [87] typically considers a higher number of interactions as indicative of high activity. Since any number of arbitrary interactions can occur independent of the study tasks, we only consider those interactions that contribute to the resulting code, i.e., typing, copying and pasting, and restructuring. Ultimately, this is similar to the common Lines of Code metric [87, 231], albeit at an even finer granularity.

In our study, the total time was constrained, though, with some participants taking the full 15 minutes or slightly more while others did not need the full time. Thus, we normalized the number of characters by time.

Since code can also be automatically generated in our study, we make sure to differentiate from where the input is coming from, i.e., whether it was a genuine interaction by the user or generated by a support mechanism. We therefore consider the percentage of code that was written by the human, as opposed to generated by the AI tool, an additional metric for the *Activity* category.

Communication Software development typically happens in teams, so effective communication with collaborators is essential. While participants completed the study alone, they did, in parts, collaborate with their AI support mechanisms. Thus, we focus on how developers retrieve information from their support mechanisms, i.e., the AI-powered chatbot, the auto-completion of GitHub Copilot, but also how they utilize the browser to access information. First, we record how frequently participants *accept code recommendations*, i.e., how frequently participants decided to use a code snippet from an outside source, e.g., the browser or any of the AI assistants. Since the browser was also available in the study conditions with AI assistant, we differentiated the source and normalized the number of code snippets over the time required to complete the tasks. An accepted code snippet, in this case, is any piece of code that ends up as part of the code base, either by copying and pasting, typing it out manually, or using auto-completion. Not all code snippets are equal, so we also record the overall length in characters of any such accepted code snippet. For this, as before, we only consider executable code, though, i.e., we exclude comments, as these have different degrees of verbosity in different environments. We count the length of the code snippet at the time of acceptance. From these two values, we can then also calculate the average size of accepted code snippets. The subjective assessment after the three tasks furthermore asks for subjective perception of how easy it was to retrieve code snippets.

Efficiency Finally, efficiency describes the amount of work a developer can perform in a given time. Since the time for our study was fixed, we recorded how many of the features and requirements each participant was able to implement in this timeframe. With multiple smaller tasks, this also helps alleviate the limitation described by Vaithilingam et al. [298], that participants may not be able to complete their task fully. While we set the subtasks in a way that, under normal circumstances, participants should not run out of them in the allotted 15 minutes, we normalized the results by total task completion time, thus preventing skewed results from very quick participants. Efficiency was also part of the self-assessment at the end of the survey.

Additional Data Besides the feedback for each category after each task and once more at the end, our survey also recorded additional, more general feedback. This includes positive and negative comments for each study condition and a subjective assessment of how well participants felt supported by the different tools and how they felt these tools affected their work.

2.2.3 Results

In the following, we report the recorded information from the participants. We first cover the categories of the SPACE framework for productivity, followed by additional feedback, including the qualitative responses by the participants. The underlying anonymized data is available as part of the supplementary material.

For each task, participants created code ranging from 45 to 135 LOC. To complete the CSV task, participants required significantly more code (mean: 95 LOC) than the PDF (mean: 67 LOC, $p < 0.001$, pairwise Wilcoxon test, Bonferroni adjusted) and Text task (mean: 72 LOC, $p = 0.002$). While we accounted for these differences between the tasks by permuting task and tool combinations and their order, we additionally checked the resulting data for effects. In it, we could not determine any notable differences where the task had a significant effect on the performance metrics.

As mentioned before, since we split the tasks into smaller sub-tasks, participants could, in theory, have skipped through each task and generated a complete solution at the end with the full set of instructions. However, we did not observe any such behavior in the screen recordings or the size and timings of the accepted code snippets.

When reporting statistical testing in the following, we always picked the appropriate test given the normality of the data test using the Shapiro–Wilk test. Based on that, we run Friedman tests with Wilcoxon signed-rank test as post hoc tests or repeated measures ANOVAs with t-test as post hoc tests using R. Moreover, we adjusted the p-values of the post hoc test using Bonferroni correction. We performed all statistical tests using R.

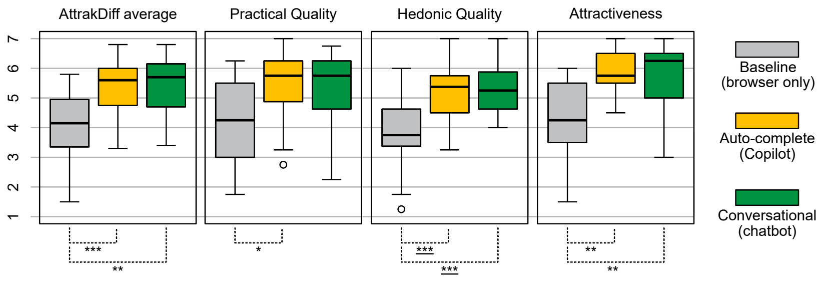


Figure 2.8: The AttrakDiff questionnaire shows some significant differences across its categories, with a consistent picture of both AI assistants similarly outperforming the baseline. (*: $p < 0.05$, **: $p < 0.01$, **: $p < 0.001$, pairwise Wilcoxon test/t-test, Bonferroni adjusted).

2.2.3.1 Satisfaction

User satisfaction, as recorded via the AttrakDiff questionnaire, varied most notably between the baseline and the auto-complete condition, where we found significant differences in all three sub-categories (Pragmatic Quality ($p = 0.027$, Wilcoxon test), Hedonic Quality ($p < 0.001$, paired t-test), and Attractiveness ($p = 0.001$, Wilcoxon test)). The conversational test condition showed similar significant differences for Hedonic Quality ($p < 0.001$, paired t-test) and Attractiveness ($p = 0.003$, Wilcoxon test), but none for practical quality. We could not determine any significant differences between the two AI assistants, with their responses being distributed quite similarly, as shown in Figure 2.8.

2.2.3.2 Performance

To assess the participants' performance, we analyzed the code they produced and determined its quality. The majority (control: 14, auto-complete: 12, conversational: 13) of participants managed to implement the requirements so that each fulfilled our testing criteria. Based on this, we could not determine any significant differences between the three groups. Likewise, the Maintainability Index for the code from all three groups was

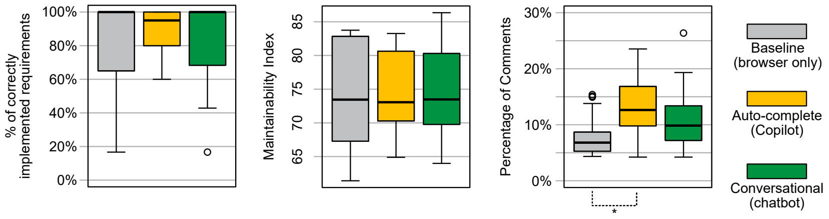


Figure 2.9: With or without AI support, there are no significant differences in the quality of the produced code. However, using the auto-complete AI tool yielded code with significantly more comments. (*: $p < 0.05$, pairwise Wilcoxon test, Bonferroni adjusted)

consistent in the interval from 61 to 86, with each having a median of 73$\pm$1. In fact, a pairwise Kolmogorov Smirnov test (see Table 2.3) further supports the assumption that the Maintainability Index for all three groups follows the same distribution.

Within the code structure of the task results, there are some minor differences, though: using the auto-complete mechanism of GitHub Copilot, participants submitted code that contained significantly more comments than the code from the control condition. Code from the conversational condition contained more comments on average, as well, but this difference was not significant.

2.2.3.3 Activity

While we set the time limit per task to 15 minutes, some participants took less time, while others exceeded it. Yet, we did not abruptly stop participants who required more time, but let them complete their current activity. This results in a distribution of task completion time as seen in Figure 2.10 with the AI assistants displaying significantly different task completion times than the control condition (auto-complete: $p = 0.028$, conversational: $p = 0.025$, pairwise Wilcoxon test, Bonferroni adjusted). Given these differences, we normalized any time-based metric to take into account the time each participant required for the tasks.

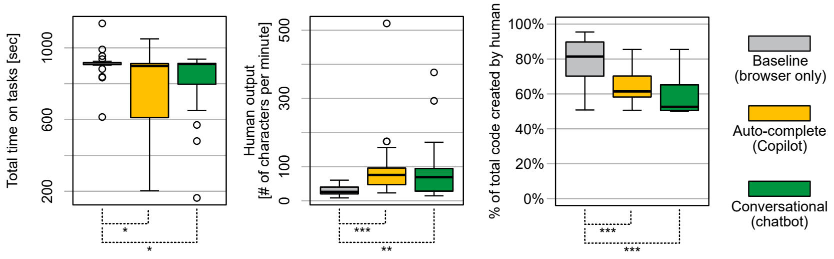


Figure 2.10: Using AI support, participants produced significantly more code, even though the AI contributed an increasing percentage to it. (*: $p < 0.05$, **: $p < 0.01$, **: $p < 0.001$, pairwise Wilcoxon test, Bonferroni adjusted)

Based on this, we determined a significant increase in how many characters each of the participants typed out per minute in the two AI assistant conditions, compared to the control condition (auto-complete: $p < 0.001$, conversational: $p = 0.002$, pairwise Wilcoxon test, Bonferroni adjusted).

While this only considers the characters a user has entered, we also saw an overall increase in code volume for the AI assistants, which matches the fact that even though the character count increased, the percentage of the code originating with the user decreased in the AI-assisted conditions (auto-complete: $p < 0.001$, conversational: $p < 0.001$, pairwise Wilcoxon test, Bonferroni adjusted). Note, though, that the source of the external code differs, as highlighted in the following section.

2.2.3.4 Communication

As previously mentioned, communication in our study setup happens exclusively between humans and available support mechanisms. These support mechanisms can either be the two AI assistants, which were available only in their respective study conditions, or the browser, which was available in any of the three conditions. Participants did use the browser in all three conditions. Still, when the AI assistants were available, this was done only by a minority (four participants in the auto-complete condition, and two participants in the conversational condition). Thus, we observe a significant difference between the control condition and the AI assistants for the number of code snippets from the browser, the total volume of code, and the average size of each snippet ($p =< 0.001$ for each, pairwise Wilcoxon test, Bonferroni adjusted), but no difference between the two AI assisted conditions.

We see a different pattern when investigating AI as the source of code snippets. While, naturally, the control condition cannot provide code from an AI, we still observe that participants accept significantly more code snippets in the auto-complete condition (mean: 12.6, median: 11.5) compared to the conversational (mean: 5.0, median: 4.5; $p < 0.001$, Wilcoxon test, Bonferroni adjusted). While the total number of characters created by the two AI assistants does not significantly differ (auto-complete mean: 514.0, median: 482.0; conversational mean: 818.4, median: 596.5), it still results in a significant difference for the average size of individual code fragments (auto-complete mean: 42.9, median: 41.8; conversational mean: 205.4, median: 110.9; $p =< 0.001$, Wilcoxon test, Bonferroni adjusted).

Taking those two sources of code together, we see that the number of code snippets from an external source differs between the control condition and when using the auto-complete support ($p =< 0.0001$, Wilcoxon test, Bonferroni adjusted) and between the two AI assisted conditions ($p =< 0.001$, Wilcoxon test, Bonferroni adjusted), but not between the control condition and the conversational AI assistant ($p = 0.083$, Wilcoxon test, Bonferroni adjusted). Overall, the use of AI does result in more code being produced, which matches the previous finding that the volume of code increases, but the percentage created by the user decreases.

Finally, the average size of accepted code snippets from any source differs significantly between all three conditions, with the smallest average snippet size in the control condition (mean: 36.1, median: 31.2), closely followed by the auto-complete condition (mean: 42.4, median: 41.8), and the conversational support tool resulting the largest snippets on average (mean: 204.6, median: 110.9).

2.2.3.5 Efficiency

Lastly, to assess productivity, we consider how many of the given requirements participants were capable of implementing. Since not all participants used the allotted 15 minutes exactly, we compared the average number of implemented requirements

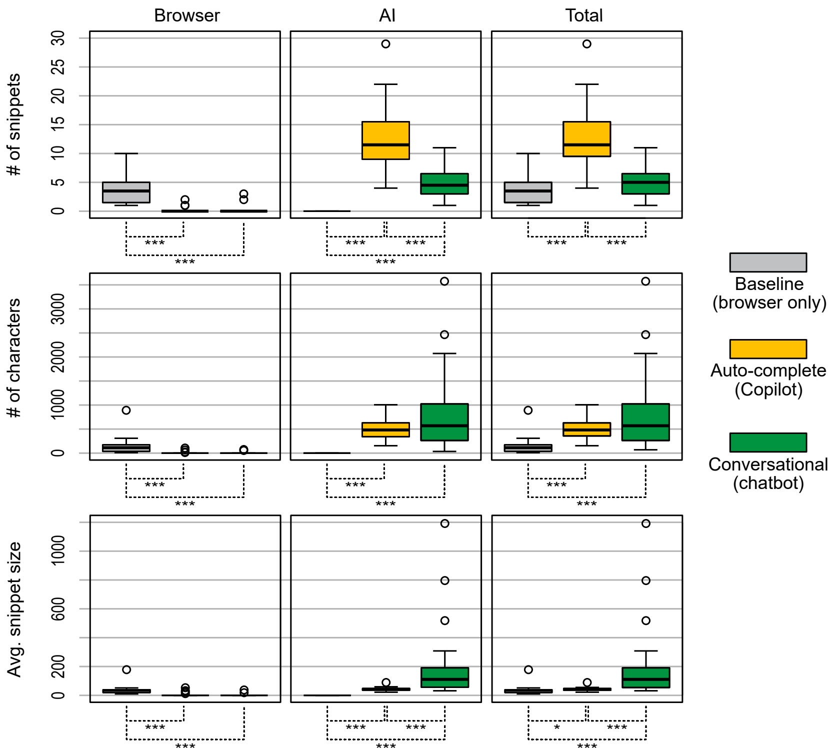


Figure 2.11: Participants communicated with their support mechanisms to a varying degree between the different environments, highlighting how the AI assistants essentially replaced the browser as the source of information. The significant differences additionally show distinctive usage patterns between conversational and auto-complete AI. (*: $p < 0.05$, **: $p < 0.01$, **: $p < 0.001$, pairwise Wilcoxon test, Bonferroni adjusted). The first column of graphs indicates how frequently the browser was used in the three different study conditions. The second column indicates how frequently AI assistance was used. While the AI was unavailable in the baseline condition, the zero value is left for completion. The third column is the sum of the previous two.

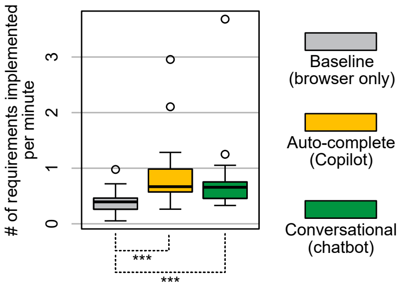


Figure 2.12: The assistance of AI tools allowed participants to complete the implementation of the provided requirements significantly faster. (*: $p < 0.05$, **: $p < 0.01$, **: $p < 0.001$, pairwise Wilcoxon test, Bonferroni adjusted)

per minute. Here, both AI assistant conditions performed significantly better than the baseline with only the browser. Notable are the outliers, where some of the participants used the AI tools to great effect, implementing up to three times as many requirements per minute as their peers.

While we set the task to take approximately 15 minutes, not all participants took this much time. While the majority spent about this much or slightly longer on the tasks, two participants took less than half the time. P19 completed the task in the auto-complete condition in just shy of four minutes, which can be attributed to the fact that they copied the feature description and let the system generate the majority of the code. Additionally, from what we could tell, they did not thoroughly verify whether the code worked and trusted the AI instead. Unfortunately, they provided no additional feedback about this. The other participant, P24, took even less time, three and a half minutes for the auto-complete condition and less than three minutes with the conversational system. This participant did state they had very high prior knowledge of both AI tools.

2.2.3.6 Self-Assessment

The responses to the Likert scale questions, where participants rated how they perceived each of the interfaces in some of the SPACE categories, generally match the previous findings: both AI assistants significantly outperform the baseline, while there are no significant differences between how the two AI tools are rated. As seen in

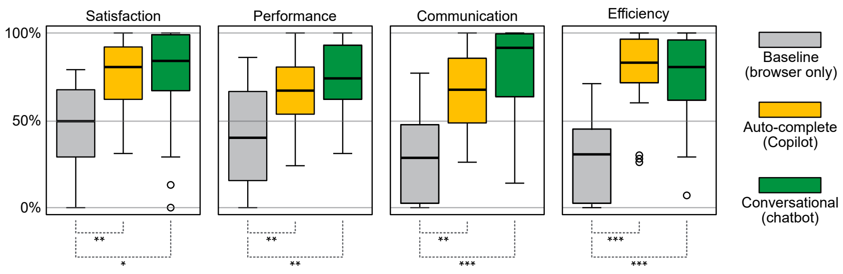


Figure 2.13: The subjective assessment paints a consistent picture, with the AI-supported conditions being rated more positively. This highlights the perceived benefits of AI assistants for programming. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$, pairwise Wilcoxon test, Bonferroni adjusted)

Figure 2.13, there are some outliers, though, with some participants being very unsatisfied with the conversational interfaces. Regarding efficiency and speed, both AIs also received a sub-par rating from some participants.

2.2.3.7 Qualitative Feedback

After each study condition and at the end of the survey, participants had an additional opportunity to provide additional, unstructured feedback on the different support tools.

For both AI-supported study conditions, 21 participants highlighted some positive aspects, while only four did so for the baseline with just the browser. In their feedback, they typically emphasized how the AI assistants were beneficial for productivity (auto-complete: 11, conversational: 13) or pointed out how these tools affect their workflow. P6, for example, noted that the auto-completion *"(increases their) productivity [sic] since (they) don't have to open Stack Overflow anymore."* In the case where participants did not have AI assistance, just three participants noted the familiarity of the interface as a positive aspect.

Five participants also highlighted that using AI support tools positively affects the entry barrier and reduces the effort to get started with a coding task. P4 stated that they *"had no clue about the imported modules" but with the help of AI, they were "able to start working on the code immediately [sic]."* However, this meant that they *"tended* *to put less effort (into understanding) the code."* P11 meanwhile opted to first generate the code and only *"read over (it) afterward to understand it."* P12 also preferred *"to see the whole generated code at once"* first using the conversational system. However, this tends *"to generate more content than needed."* (P12)

P12 also pointed out that in a regular context, they rarely *"have the coding tasks layed [sic] out in a way that (they) can just copy (them and thus writing) the input that's needed for the generative tool also takes time."*

In addition, participants were also critical of the two AI interfaces, highlighting a number of issues. Using the auto-complete feature to generate code led to some participants (5) struggling to understand the generated code quickly. They also were sometimes unhappy with the interaction, where lines would not be generated as expected (5) or the model generated clearly incorrect code (2). According to the feedback, the conversational system suffered from being a separate interface, which required a context switch from editor to chat (6) and providing too much information (3). The elaborate answers of the chatbot were criticized for being distracting and can *"hinder (developers from focusing) on the code itself."* (P6) Given the speed at which code was generated, P4 also noted that they *"tended to put in less efforts [sic] to understand the code."*

Finally, participants also highlighted that the perception of the AI tools might greatly be influenced by individual differences, e.g., "*a user [who has] gotten used to using AI tools [for] coding [relying again on] Google searches [can feel] very inefficient*" (P21). Meanwhile, using the editor without AI assistance may "*(require) a skilled developer who new [sic] Python code [...] inside and out*" (P21). P13 also suggested that the more elaborate completion of GitHub Copilot and the longer answers of GPT-3 that may be beneficial for novices. P29 meanwhile summarized that their experience "*depends on what [they] expect from the [tool]*" and that without AI experience, they would "*not expect [their editor] to solve [...] coding tasks for [them].*"

However, during our analysis, we investigated whether prior knowledge significantly affected the different productivity metrics and whether there were any differences between experienced and novice or professional and amateur users. During this analysis, we could only identify a significant effect of experience on the task completion time, with more experienced programmers being slightly faster (two-way ANOVA,

$F = 7.029, df = 1, p = 0.01$). For all other metrics, there was no significant effect of or notable correlations with the prior experience across the whole group. Instead, we noted more individual differences.

2.2.3.8 Individual Differences

Given this potential for individual differences and the number of outliers in a number of metrics above, we inspected these participants' feedback and performance to determine whether the outliers allow us to learn more about individual, distinct usage patterns. We consider any datum an outlier that falls outside 1.5 IQR. With this, we found 72 outliers from 20 participants across all metrics and study conditions. While some of these are simply participants who happened to yield an outlier datum in one or two categories, more than half (40) of these outliers stem from just six participants (namely P2, P4, P14, P19, P22, and P24) suggesting a more fundamental difference. We thus screened their behavior and their responses for additional insights.

In terms of prior knowledge, on average, the group of outliers did not differ from the remainder of the participants in any significant way. Regarding prior knowledge about the AI assistants, the group split into two camps, one with very high levels of prior knowledge (P2, P14, P19, P24) and one with barely any prior experience (P4, P22). However, these groups exist across all participants, albeit not as pronounced.

In the latter group, the two participants employed two distinctly different approaches to using the AI: P22 barely used it in either study condition. Instead, they relied mostly on their own knowledge and used AI assistance only when stuck. P4, on the other hand, actively used the AI tools but struggled when the generated responses did not fit immediately.

For the participants who indicated a high degree of prior knowledge, we also noted some behaviors that differentiated them. Since these participants were familiar with both types of interfaces before the study, we observed a mixing of interaction patterns. P2, for example, attempted to use the auto-complete interface in a conversational style by typing questions for the AI as inline comments. Generally, though, three of the four participants (P2, P14, P24) used the auto-complete interface to generate code quickly line by line, with short prompts, and the conversational interface to generate functions or whole scripts with longer, more elaborate prompts. Only P19 did this the other way around. For using the conversational system, all four participants generously copied the task description to provide context for the system. This also shows in the metrics for snippet size (cf. Figure 2.11) where the outliers in the conversational system P14, P20, and P24, all indicated a high degree of ChatGPT experience. However, we observed that higher experience also led to them accepting the results with little checking, particularly P19. P24 meanwhile postulated that with code writing moved to the AI, the task of the developer should shift more towards supervision.

Knowledge of the AI tools also helped the participants complete the tasks quickly, with P14, P19, and P24, all experienced with AI, completing both AI-supported tools faster than most of the other participants. Only P24 showed a similar speed in the control condition. This pattern is also reflected in the number of correctly implemented requirements, where P24 again excelled in all three categories, although more so with AI support, and P19 also performed above average.

2.2.4 Discussion

Our collected results paint a clear picture and allow us to draw some conclusions as to the developers' behavior and how AI assistants can be utilized for productive programming. Next, we will outline some of these patterns, the potential reasoning behind them, and what these findings mean in the context of the study and for software development more broadly.

2.2.4.1 Programming Productivity

The participants' performance and their feedback are very consistent across most of the metrics we recorded. From this, it is quite clear that with either form of interaction, AI assistants provide a strong benefit for developer productivity overall and many of the individual aspects of the SPACE framework. The relative improvements, e.g., the fact that developers implemented about 65% more requirements with either AI assistant, also match the findings from prior work, for example, the 56% improvement in task completion time found by Peng et al. [229].

Overall satisfaction with the system and the coding experience with them is high, and the self-reported satisfaction closely matches the results of the AttrakDiff questionnaire. While the conversational system has an ever so slightly higher mean and median satisfaction, it also has two outliers who rated this tool rather poorly (0 and 29 out of 100). However, neither of these participants had any notable trouble with the system. Based on the qualitative feedback, one reason for this may be the comparison to the auto-complete condition, where one participant noted that this method was perceived to be “more efficient than using (the conversational interface)” (P20) as it did not require a context switch. Similarly, the same participants who were unsatisfied also provided a low rating for the perceived efficiency of the conversational system, noting that the volume of information “hinders (them) to focus on the code itself” (P6). P2 also highlighted that typing out a prompt can take quite some time, and the response may not always be satisfactory or provide an “overload of reply info, [and] too much [...] to read [...] to find specific answers.” It appears that in order to use conversational systems efficiently, some work still needs to be done to focus the output of LLMs on what is actually relevant. There are many approaches one can take to this, e.g., training these systems to give more concise answers, different forms of presentation that lead the attention to the relevant parts, or personalization of the output, reducing the information to what the individual user typically focuses on.

2.2.4.2 Usage Strategies

For the most part, participants used the AI tools fairly conservatively, in that they followed the intended steps of writing a prompt, inline or in the chat, and then accepted, rejected, or refined their prompt. Considering the relative novelty of these tools, it is quite likely that users have not yet developed strong, individualized usage patterns or personal preferences. Overall, the use of AI resulted in a larger code base, but an increasing part of it originated with the AI and not the user. This is more pronounced and explicitly noted by the participants for the conversational system.

In the categories of *Activity* and *Communication* of the SPACE framework, we saw considerable differences between the two types of AI tools, though. Based on the data on how much and how frequently it was accepted, we clearly see that code snippets from the chatbot tend to be larger, while the auto-complete interface seems to favor more but smaller code snippets. The typically single-line, auto-completed code snippets are still quite quick to grasp and understand, while larger snippets require more dedicated mental effort to read through them and determine whether they can be accepted or not.

This indicates distinct usage strategies for the two systems: for the auto-complete interface, participants appear to use high-frequency, short snippets, while the conversational system is used for longer blocks. While this skews the acceptance rate in favor of the auto-completion interface, the total volume of generated code remains quite similar since the conversational code snippets, while fewer, are larger in size. These strategies were present across the spectrum of prior experience, suggesting that this comes to programmers naturally and does not require much learning. Only at the higher levels of experience did we see that participants started to mix the paradigms, using the chatbot for short, rapid-fire requests or the auto-complete interface like a conversational system. To us, this indicates that the different types of interaction may have different use cases. The larger code snippet, often with explanations, of the conversational systems appears to be an alternative to the conventional method of searching for code via the browser. Rather than querying an online search engine for information, participants seem to prompt the chatbot instead. This is indicated by the similar frequency in accepted code snippets between the conversational system and browser, which was noted in the comments of seven participants. This also aligns with broader changes in the software development landscape, e.g., the decrease in traffic on Q&A sites like StackOverflow, which is in part attributed to them being replaced by tools like ChatGPT [130]. The two different usage patterns in our study also provide quantifiable evidence for the qualitative findings by [19], who also found different usage modes, an “explorative mode”, equivalent to how participants used the chat interface for more explorative search queries, and an “acceleration mode” for more productive coding through more, rapid-fire completions.

If the conversational system is not available, experienced users will try to emulate the interaction with the chatbot rather than switch to the browser. At the same time, sometimes the participants used the chatbot for quick, line-by-line suggestions but then noted that the context switch was an unnecessary overhead. These two strategies further suggest that both interfaces have distinct use cases, which warrants the existence of both. Combining the two, e.g., an auto-complete interface that allows inline chatbot-style interaction, may be a hybrid solution. However, trying to cater to both use cases may decrease the quality of the model and may incur unnecessary usability overhead. Another way to combine the interaction patterns may be in computational notebooks, where the code is structured in distinct cells anyway with dedicated textual cells in between. Presenting the conversation with an AI as a sequence of textual and code cells that can be executed at will, may be an avenue worth exploring.

While these types of interaction may come naturally, experience with AI systems does seem to affect working practices. While the less experienced participants sometimes struggled to get the AI to generate well-integrated code, the more experienced users appear to be familiar with the idea that they must provide a certain level of context to the AI. This demonstrates that there is a learning curve to these tools. Learning to write good prompts for conversational systems and good prompt comments for auto-completion will likely become an essential skill. At the same time, an understanding of how these systems work can also be beneficial to understanding how best to use them.

While these results already look promising and support the hypothesis that AI will yield a considerable productivity benefit, we are likely only at the beginning of a shift in work practices in software development. Not only does this mean that new tools may emerge that upset established patterns again, but it will also affect how developers use these tools. Given more time to get familiar with them, developers may explore and refine usage patterns, learn the intricacies of these systems, and generally learn how to use them better. Thus, productivity may even further increase in the future.

2.2.4.3 Code Quality

The one aspect where the use of AI assistants did not lead to much of an increase is the performance category, where we measured objective code quality metrics. The very similar quality of the resulting code can be considered a positive, though, meaning that the AI assistants produce code that is about equally good as what a human can produce. Furthermore, when software projects increase in size, it becomes challenging for developers to maintain an overview, especially when they rely on code from legacy projects, coworkers, libraries, etc. Consolidating this large volume of information is one of the strengths of LLMs. Thus, in larger projects, they can help find existing functionality, reduce code duplication, provide suggestions in a consistent code style, and much more. Considering that LLMs reproduce patterns found in their training data, and they are typically trained with human-written code, this should not be too surprising.

While the objective code metrics did not vary much across the different test conditions, the subjective perception of this did. Participants tended to rate the performance and quality of the AI assistants as superior to the baseline, which may suggest a degree of overtrust in the AI. While the quick and often functioning responses from AI tools can be impressive, it appears necessary to temper expectations. A broader knowledge of how these systems generate code may be one way. Still, it may also simply be a matter of habituation, i.e., people getting used to these tools and then losing their novelty factor. In addition, any effort to increase the quality of these systems, i.e., bringing their quality closer to the users' expectations, will also alleviate this.

Additionally, mechanisms to quickly and easily verify the quality are probably desirable either way. One way for this may be to give users a way to quickly verify whether the code snippets do what they claim to do. There are already some changes that can facilitate this, e.g., the ability to execute code directly in ChatGPT [78, 207]. Automatically generating easy-to-understand unit tests along with the code may be another approach, which will not only verify that the code works upon generation but also increase overall test coverage and can thus provide a long-term benefit for the software project. Code documentation is another aspect where using LLMs can help and contribute to better and particularly easier-to-understand code. While the significantly higher amount of comments in the auto-complete condition may suggest that this is already the case, it needs to be noted that this may be a consequence of the tools function: to trigger GitHub Copilot to generate code, one way is to describe the desired functionality in a comment first. This naturally leads to more comments, then. However, looking at the code of the participants, it becomes clear that the comments are more descriptive and less explanatory. This means that these comments contain very similar information to the following line of code, not an explanation or a rationale behind the code, which would offer greater value.

With advances in many of these areas, it is no surprise that many vendors of software development tools are already actively working on integrating them into their tool suites. As our findings demonstrate, the form of interaction plays an important role in how people use these tools, though. The seeming popularity of some interface types should not be construed as a superiority, though. The different usage patterns between conversational and auto-completion in our study strongly suggest the different forms of interaction have specific situations where they excel. Thus, it is advisable not to focus on just one form of presentation and interaction for LLMs in software development, but rather offer developers the flexibility to choose their tools according to their needs from situation to situation. However, this also means that a transition between different interaction forms should be relatively seamless to minimize overhead and the need to learn the idiosyncrasies of different tools.

Thus, open research remains, since overcoming these challenges and improving the user experience may lead to wider adoption of AI support systems and thus increase productivity. However, other challenges play a similarly important role, e.g. technical improvements of the underlying models to create suggestions with better fit and quality or legal requirements to ensure that generated code does not introduce, e.g., licensing issues. Addressing these challenges will also be necessary for a broad adoption and long term benefits of AI-powered support tools.

2.2.4.4 Limitations

Considering that the sample of participants in our study leans towards the younger side, they may be particularly affected by these current developments: If younger developers learn to program with these tools early on, they may closely integrate them into their workflows. More senior developers could be reluctant to use AI tools. Still, it may also be the case that they see different usage patterns when combining their conventional experience with the new options in AI-assisted programming. While we recorded prior knowledge and expertise, we could not determine any effects or notable correlations between the level of expertise and productivity. Considering that LLMs for coding have only been available for a relatively short time, any effect of their novelty seems to be fairly limited or at least consistent across the whole sample. It may also be the case that even more experienced users have not had the time to develop enough expertise to have an effect. Future studies specifically targeting long-term users may yield further insights. Aside from the effect of expertise on the individual, it may also be interesting to see how development teams with multiple humans and AI assistants collaborate and how both expertise in AI and traditional development change the group dynamics.

Conducting studies like this in the field may not just yield results that are closer to group dynamics but also give the opportunity to use real-world coding tasks. While we picked the tasks in our study to reflect common development tasks, they were limited in scope to accommodate the study setup. Since the tasks are based on prior work, this helps contextualize the results with prior research, but field studies may still present different observations. Recording the activities and output of different development teams in the field, with or without AI support, over a longer period will likely emphasize the productivity benefits. Still, it may also show larger interaction patterns, e.g., during which phases of development AI can be most useful or how human-AI teams collaborate. Additionally, not all requirements will be as clearly structured into sub-tasks as they were in our study. In fact, properly translating high-level requirements into concrete programming steps is an essential aspect of computational thinking [323] and an important part of the development process. Here, too, LLMs could assist by translating vague requirements into concrete steps, which humans can check before being translated into code. For now, our study focused on the production of code, thus deferring this part of the development process to future studies. While this is a limitation, since the activities in the study were not an exact representation of open, creative programming, the constrained nature of the tutorial was consistent across all study conditions. By permuting the tasks, we expect no adverse effect or bias from excluding the task decomposition step from the development process and thus expect the comparison between the study conditions to be valid

Finally, since LLMs are inherently probabilistic, the specific model we used in our study may be a factor that influences the generated code and how useful it was. Additionally, the AI-assisted conditions used related but not identical models, which can further skew the data. Still, since the two underlying models share a base, and based on the fact that the generated code was quite similar in its general functionality and according to a number of quality metrics, it does not appear that this played any notable role. Finally, current LLMs only present a snapshot in a quickly evolving field. It is very likely that technological improvements will further impact what is possible and thus affect developers' behavior.

Table 2.1: Overview of previous studies that evaluate the impact of LLM-based programming assistants on developer productivity.

Author	Study	Assistant	Productivity Dimensions	Productivity Results
Peng et al. [229]	Between-subjects study that contrasts no assistant with GitHub Copilot	GitHub Copilot (autocomplete interface, context-aware)	*Task Completion Time* (Efficiency)	The group with access to GitHub Copilot completed the task 55.8% faster. In an additional self-assessment, the developers slightly underestimated the measured effect by suspecting a 35% increase in speed.
Vaithilingam et al. [298]	Within-subjects study that contrasts IntelliSense and Copilot	GitHub Copilot (autocomplete interface, context-aware)	*Task Completion Time* (Efficiency), *Interaction Patterns* (Communication), *Tool Preference* (Satisfaction)	No significant effect on task completion time since over-reliance lead to developers not reviewing code and facing debugging rabbit holes later. Developers preferred Copilot over IntelliSense.
Qureshi [239]	Between-subjects study that contrasts printed-out documentation with ChatGPT	ChatGPT (conversational interface, context-aware)	*Task Completion Time* (Efficiency), *Code Correctness* (Quality)	The group that utilized ChatGPT as a programming assistant achieved higher scores in less time. ChatGPT requires a deep understanding of the tool's capabilities and prompting skills to generate solutions for complex problems.
Sandoval et al. [262]	Between-subjects study that contrasts no assistant with GitHub Copilot	GitHub Copilot (autocomplete interface, context-aware)	*Functional Correctness* (Quality), *Absence of Security Bugs* (Quality)	The incidence rate of severe security bugs does not increase when developers are assisted by GitHub Copilot in the context of C programming.
Barke et al. [19]	Observation of how developers use GitHub Copilot	GitHub Copilot (autocomplete interface, context-aware)	*Interaction patterns* (Communication)	The communication happens either in acceleration mode, where developers know what to program and prefer small code-based suggestions, or in exploration mode, where developers conceptualize the solution and communicate in natural language more to find abstract approaches.
Jiang et al. [137]	Observation of how developers use a self-made LLM-based assistant	Self-made snippet generator generates HTML snippets from comments in code (autocomplete interface, context blind)	*Interaction patterns* (Communication)	Participants find the tool useful for facilitating API look-ups and generating boilerplate code They use it primarily to generate small code snippets equivalent to a single line of code. Prompts were reiterated to improve suggestions.
Mozannar et al. [196]	Observation of how developers use GitHub Copilot	GitHub Copilot (autocomplete interface, context-aware)	*Interaction patterns* (Communication)	Copilot-related activities account for more than half of the total time and most time is spent verifying suggestions. Furthermore, the time spent looking at documentation is low.
Ziegler et al. [343]	Comparison of a developer survey to telemetry data gathered by GitHub Copilot	Github Copilot (autocomplete interface, context-aware)	*Acceptance Rate of Suggestions* (Communication)	The acceptance rate of recommendations is a strong predictor of the perceived productivity and increases in the usual working hours of a single developer.
Ross et al. [257]	Observation of how developers use a self-made LLM-based assistant	Self-made chatbot powered by react-chatbot-kit and Codex (conversational interface, context-aware)	*Interaction Patterns* (Communication), *Tool Preference* (Satisfaction)	The conversational interface facilitates programming discussions, generates relevant code, and promotes the use of follow-up questions. Participants perceived the assistant as beneficial for their productivity. Autocomplete, conversational, and search interfaces offer complementary support.

Table 2.2: After using each environment and support mechanism, participants rated them with respect to different productivity metrics using quasi-continuous Likert scales.

SPACE Category	via	Shapiro-Wilk Test W	Shapiro-Wilk Test p	Friedman Test / ANOVA χ^2 / F	df	p		Kendall's W / η^2
Satisfaction	AttrakDiff (Avg.)	0.953	0.009	21.447	2	< 0.001	***	0.447
	Pragmatic Quality	0.917	< 0.001	7.467	2	0.024	*	0.156
	Hedonic Quality	0.979	0.259	19.764	2	< 0.001	***	0.349
	Attractiveness	0.925	< 0.001	18.782	2	< 0.001	***	0.391
Performance	% of correctly implemented requirements	0.724	< 0.001	0.636	2	0.728		0.013
	Maintainability Index	0.950	0.007	0.065	2	0.967		
	Percentage of comments	0.925	< 0.001	8.583	2	0.014	*	0.179
Activity	Character output per minute	0.627	< 0.001	23.083	2	< 0.001	***	0.481
	% of code created by human	0.931	< 0.001	27.750	2	< 0.001	***	0.578
Communication	# of Snippets (Browser)	0.690	< 0.001	41.325	2	< 0.001	***	0.861
	# of Characters (Browser)	0.470	< 0.001	41.325	2	< 0.001	***	0.861
	Avg. Snippet Size (Browser)	0.576	< 0.001	41.325	2	< 0.001	***	0.861
	# of Snippets (AI)	0.854	< 0.001	42.750	2	< 0.001	***	0.891
	# of Characters (AI)	0.700	< 0.001	36.333	2	< 0.001	***	0.757
	Avg. Snippet Size (AI)	0.450	< 0.001	42.750	2	< 0.001	***	0.891
	# of Snippets (Total)	0.874	< 0.001	31.043	2	< 0.001	***	0.647
	# of Characters (Total)	0.695	< 0.001	27.583	2	< 0.001	***	0.575
	Avg. Snippet Size (Total)	0.418	< 0.001	25.083	2	< 0.001	***	0.523
Efficiency	# of requirements implemented per minute	0.714	< 0.001	6.583	2	0.037	*	0.137

Table 2.3: Results of a pairwise Kolmogorov Smirnov test suggest that the Maintainability Index of the code in all test conditions follows the same distribution.

D	Control	Auto-complete
Auto-complete	0.208	—
Conversational	0.167	0.125

p	Control	Auto-complete
Auto-complete	0.679	—
Conversational	0.898	0.994

Table 2.4: Participants provided a self-assessment for multiple productivity metrics in a survey after testing the three test conditions. A Friedman test indicates significant differences between the three conditions.

	Shapiro-Wilk Test		Friedman Test			
	W	p	χ^2	df	p	Effect size
Satisfaction	0.914	< 0.001	21.894	2	< 0.001	0.456
Performance	0.954	0.001	18.087	2	< 0.001	0.377
Communication	0.925	< 0.001	18.083	2	< 0.001	0.377
Efficiency	0.901	< 0.001	25.333	2	< 0.001	0.528

2.3 Summary

This chapter demonstrates how the integration of AI features can enhance and extend existing practices and development tools, using adaptive UIs for IDEs and GenAI for coding as case studies. The subjectively perceived and objectively quantifiable benefits from both studies highlight how AI can benefit developers, regardless of whether it happens via a compeltely new system (RQ 2) or as an extension of existing and familiar tools (RQ 1).

With the prototypical implementation of the gaze-adaptive IDE, we demonstrate how existing tools can be enhanced with relatively little effort and how even limited data can produce meaningful functionality. This was confirmed by the participants in the evaluation. However, they highlighted important aspects that should be considered when enhancing existing tools with AI. Most notably, the emphasized the balance between automation and for staying in control of the tools with which they are already familiar. Thus, they suggested hybrid approaches that utilize automation for context detection coupled with rule-based or human-in-the-loop feedback.

Secondly, using LLMs for programming as an example for an new and emerging type of systems, we see both subjective and objectively quantifiable benefits on developer productivity (RQ 2.1). The feedback from our user study shows how this new AI support tool can successfully enhance developer productivity. Still, some of the technical metrics, like code quality, were relatively unaffected. Subjective perception and behavioral patterns, on the other hand, are very much affected by this new AI tool. Additionally, interface design also plays an important role in how the benefits of AI tools reach their users. The inline auto-complete appears to favor many shorter code snippets, while developers use the chatbot for longer interactions, resulting in more elaborate code suggestions, seemingly replacing traditional online search. While both use cases result in about the same level of productivity in our study, the larger volume of information and the longer turnaround time for the chatbot can occasionally lead to lower user satisfaction. Still, in their current state, new AI tools like LLMs already provide a clear benefit for productivity and will likely make a lasting impact on software development and the ecosystem of development tools.

Considering how both of these studies demonstrate the importance of interface design, human perception and the effect on the developers' behavior, it is clear that human factors are an essential non-functional requirement for applying AI in existing and new software development tools.

3 Software Development for AI

As the previous chapter demonstrated, data-driven application and AI features offer considerable benefits to developers. However, these systems introduce additional complexity, not just in the interface but also in their implementation. While the latter does not affect most people who just use these systems, developers are in a unique situation where they not only use data-driven systems but are also the ones building them. Thus, this increased complexity affects them two-fold: as users of these systems, but also as their developers.

At the same time, the increasing ubiquity, particularly in critical domains (e.g, [179, 197, 241]), continues to drive an interest in high-quality data-driven software. Understanding how this type of software is different and how these difference affect developers during development is a crucial step for understanding what is necessary to support them and enable them to build high-quality software.

While one may argue that data-driven systems are also just another kind of software, as described before, they operate in a fundamentally different way. Karpathy [142] also refers to data-driven software "Software 2.0" to denote this fundamental shift in operating principle and development practice. While the technological differences between the traditional "Software 1.0" and "Software 2.0" are relatively well understood [18], it is not fully clear how this paradigmatic shift changes the work of developers. To support developers of data-driven applications, we must first understand the different aspects of the developers' behavior and how development has changed. Thus, this chapter will first explore whether this difference in operation also leads to developers perceiving these systems differently. If developers were to view data-driven systems the same way the view traditional software, this would imply that much of existing research in software engineering could be applied here as well. However, as Section 3.1 will highlight, this is not the case. Instead, developers are well aware of the differences and, in consequence, behave differently when working with data-driven code. As a consequence, the remainer of this chapter will then explore what this means for tooling to build data-driven applications.

3.1 Developer Behavior during Data-Driven Development: Code Reading

This section is based on the following publications:

Thomas Weber, Christina Winiker, and Heinrich Hussmann. “A Closer Look at Machine Learning Code.” In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI EA ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380959. DOI: 10.1145/3411763.3451679
Thomas Weber, Christina Winiker, and Sven Mayer. “An Investigation of How Software Developers Read Machine Learning Code.” In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’24. Barcelona, Spain: Association for Computing Machinery, 2024, pp. 165–176. ISBN: 9798400710476. DOI: 10.1145/3674805.3686678

To answer RQ 3, i.e. to explore the effect which data-driven software can have on the developers’ behavior, we first explore how software developers read code in this context. Famously, most code is read more than it is written, which makes code reading a crucial part of software development. Prior work has, so far, investigated code reading behavior for traditional code, e.g., [37, 228, 254]. In the following, we go beyond this related work and compare the reading behavior for traditional code with that for code that implements Machine Learning models.

To this end, we first investigate how traditional and data-driven code differ on a structural level using static analysis on publicly available code repositories that contain traditional and machine learning code.

Research Question 3.1:
How does traditional code differ from code that implements data-driven systems on a structural level?

Informed by the structural differences which this showed, we investigate whether these differences have an impact on the reading behavior of developers. We utilize eye tracking to record the developers’ gaze to investigate the following questions:

> **Research Question 3.2:**
> What structures in the code are of interest to developers in Machine Learning code compared to code that does not use Machine Learning?

Since reading behavior can be greatly impacted by prior knowledge and experience, we furthermore compare the reading behavior between participants with different levels of expertise expertise:

> **Research Question 3.3:**
> Does the expertise of experienced Machine Learning developers lead to them focusing on different aspects of the Machine Learning code compared to novice developers?

To address the latter two questions, we conducted a user study (N=18) in which we asked participants to engage with different Python code fragments (see Figure 3.2 for examples). Our results showed differences in the reading behavior, which show that some syntactic structures are read differently in code that implements Machine Learning models. This also implies that developers are aware of the differences and adapt their behavior accordingly. From this, we conclude that for supporting developers, Machine Learning code should not be treated the same as traditional code. We also uncovered differences in expertise, which further underpin the need for approaches specific to the target group and context rather than one-size-fits-all solutions.

3.1.1 Related Work

While the field of Artificial Intelligence is many decades old, more recent advances like deep learning, i.e., Machine Learning with highly complex models, and an increase in computational power have propelled it into the mainstream for software development. As previously discussed, this brings with it changes to software development. The following section will briefly outline how literature has addressed these changes. Beyond that, we will also provide some background about the methodology we used to advance the state of research in this field, eye tracking.

3.1.1.1 Developing Machine Learning Systems

Developers now use Machine Learning (ML) in many areas; thus, it is essential for software engineers to be familiar with the general concepts. However, ML adds additional complexity to the development since developers must create and improve not just the code but also the underlying data and ML models [124]. Thus, making these complex ML systems easier to understand is important. While there are successful efforts to make traditional code easier to understand, e.g. using ML [24], the different level of complexity in code of data-driven applications remains an open challenge.

Prior work has recognized developers as a valuable target group for improving the ML experience [11, 38, 222, 223, 237, 244]. Consequently, researchers and industry have produced several methods, tools, and support mechanisms to assist software developers in creating data-driven applications. Here, they have created tools to support ML programming in a more user-friendly manner for decades. One of the earlier, well-known ones is *RapidMiner* (originally *YALE*) [247], which leverages graphical programming to present complex data processing pipelines in a simple, visual format. This graphical approach has been extended and adopted in many tools since, like *Orange* [68], *Darwin* [100], *KNIME* [26], and many more.

Nonetheless, textual programming remains the predominant way of creating ML systems. At this level, the most substantial improvements have come more from the now ubiquitous libraries and frameworks (e.g., *scikit-learn* [226] or *TensorFlow* [1]), which take over the low-level details and aim to provide easy-to-use abstractions. Their development often happens in an ad-hoc fashion and still needs more validation, e.g., whether the provided abstractions are appropriate. From the language design perspective, there is a growing effort to optimize various aspects of programming languages for ML, e.g., with the *Compilers for Machine Learning* workshop series[1]. However, so far, they have focused more on technical implementation. Meanwhile, improving the literal syntax or creating ML-specific languages specific remains an interesting avenue to make ML programming effective or efficient.

For now, Python is arguably one of the most widely adopted programming languages in the area of data-driven applications, particularly in combination with com-

[1] www.c4ml.org/

putational notebooks [278] like Jupyter notebooks[1] and Google Colab[2]. This type of editor is a traditional code editor at its core but offers increased interactivity and allows developers to mix documentation, code, and outputs in one document. Naturally then, there are efforts to improve these notebooks as well. The mage [151, 152] extensions for Jupyter notebooks, for example, adds interactive widgets which let data workers modify the output of their code. These changes are automatically reflected, similar to*Programming-by-Example* systems [200]. The feedback from a subsequent user study was positive, highlighting some of the challenges of ML code which can be complex, particularly for novices. Therefore, they suggested hiding parts of the code that would confuse them. More experienced developers, on the other hand, viewed the presentation in code more favorably.

The interactivity of computational notebooks greatly facilitates the exploratory behavior of developers. This also brings challenges; since ML systems are a tight interplay between models and data, maintaining an overview of various configurations can be difficult [66, 147, 148]. The evaluations conducted for some of these systems show their potential benefits, but these are difficult to separate from the concrete implementations. Creating fully working software tools like that requires much development upfront. Rigorous requirement elicitation is desirable to motivate this kind of effort. Yet, while some of these projects start with an initial phase of gathering user requirements [99, 147, 151] the reporting typically emphasizes summative evaluations. Additionally, they often also build on anecdotal evidence and the experience of their creators. This limits how well other researchers can rely on the insights that informed the original design. We argue that this common top-down approach, i.e., evaluating a full, working system with a focus on summative results, can be supported by a bottom-up perspective, i.e., early formative assessment, where one investigates various low-level aspects of the development first to get a foundational understanding of the requirements of data-driven development (cf. [156, 202]). These insights then inform further design decisions, especially with existing knowledge from traditional software development. Instead, we can first determine where these development paradigms match and differ. To get a comprehensive picture of these differences researches, so far, investigated

[1]https://jupyter.org
[2]https://colab.research.google.com

multiple different aspects that play a role during software development, from development processes to collaboration or how developers write code. Hesenius et al. [120], for example, investigate these differences on the process level and highlight where current development processes do not accommodate data-driven development. In consequence, they propose an extension to established development processes, the EDDA (Engineering Data Driven Applications) process, which contains additional steps specific to data-driven development.

Given the number of publicly available software projects on online collaboration platforms, they have also become a prominent source for further low-level insights into how developers work. For example, Simmons et al. [281] compare roughly 2000 public repositories, half with ML and half only using traditional code, concerning code quality. Their analysis concludes that there are some similarities but notable differences between traditional and ML codes. While the complexity of the code base, as measured by McCabe's Cyclomatic Complexity [184], is similar, it appears that repositories of data-driven applications adhere less to existing coding and naming conventions. While using the code repositories alone, they could not determine the reason for these differences, they did highlight differences in the developers' behavior, which warrant further investigation. However, the use of code quality metrics leaves it unclear whether the repositories are, in fact, also structurally different or whether the same structures are just present in a different quality.

In this paper, we will continue this line of research to determine differences between ML and traditional code. However, we will primarily focus on human behavior using eye tracking, a method frequently used in empirical analyses of software developers.

3.1.1.2 Software Development Research using Eye Tracking

Sharafi et al. [271] provide an overview of several such publications which emphasize the need for a uniform nomenclature to carry out eye-tracking studies, particularly for behavioral observation studies for software engineering. Their paper provides an overview of terms such as "fixation rate," "saccades," and their use in existing eye-tracking studies in software engineering. We will follow the terminology by Sharafi et al. [271] throughout this text. In addition to that, Sharafi et al. [272] expand on this with a systematic review of eye tracking in software engineering, showcasing the benefits of the method for investigating several questions, including but not limited to debugging [22, 118], code comprehension [23, 254], and collaboration [273], all of which could contribute to a better understanding of software development [133].

Beyond eye tracking as a general methodology, code reading behavior – the topic of this paper – has also been discussed in a series of publications, including the study by Busjahn et al. [37]. They compared the linearity of the reading order in software code structures of inexperienced and experienced software developers. Busjahn et al. [37] analyzed this by first having inexperienced programmers read natural language text, then programming code, and then having experienced programmers read code. They then compared the order of the content sequence and the order of code execution [37]. The results of their study show that inexperienced programmers read code less linearly than natural text but still more linearly than experts [37]. Peachock et al. [225] replicated this study. Their results show that developers read source code less linearly than natural text but that there are no differences in linearity regarding the level of experience or programming skills.

Peitek et al. [228] conducted a study to investigate the linearity and reading order of programming code. They conclude that "the linearity of source code is a major driving factor that determines programmers' reading order, while experience and comprehension strategy seems to play more minor roles" [228]. Since the functionality of Machine Learning systems, particularly those using Neural Networks, depend less on the linearity of instructions in the code and more on implicit factors like the data, weights, etc., this effect may not translate to this different development paradigm and thus warrants further analysis.

These differences in the code are subject to another eye-tracking study by Weber et al. [C7], who leveraged the structure of code to determine which elements in ML code are areas of interest for novice programmers. They found reading patterns that suggest that there are benefits to linearity in this context. Since the study by Peitek et al. [228], which suggests that prior knowledge is only of minor importance for code reading behavior, was on traditional code only, and the study by Weber et al. [C7] was on novices only, it remains still unclear how ML professionals and experts are affected. Particularly for a paradigm like Machine Learning, which may not yet be part of the early education for all programmers, expertise may play a more significant role.

From these previous works, we can conclude that the reading behavior of developers can but does not have to vary based on multiple factors like code structure and prior knowledge. In the context of Machine Learning code, these effects have only been explored for novices [C7]. So, we expand on this in this work by collecting new data from a study that includes experts. This should increase generalizability and provide insights into an important target group.

3.1.2 Static Analysis of Code Repositories

To answer RQ 3.1, we analysed publicly available code repositories to determine whether there were quantifiable differences between the two types of repositories – repositories that implement machine learning functionality and without. For this, we selected 3515 code repositories from the literature [281] plus popular more recent additions and counted the different grammatical constructs in the code. Of these repositories 1534 contained applications that utilize machine learning, while the remaining 1981 were traditional applications. These repositories all used the Python programming language, since it is both general purpose language with wide adoption but is also very popular for machine learning [301], making it an ideal candidate for comparative studies.

After downloading the current state of these repositories, we used the Python *ast* package to construct the Abstract Syntax Trees for the Python code in the repositories. We accumulated the number of occurrences for each grammar construct from Python's formal grammar[1] for all valid ASTs (1431 traditional repositories, 885 with ML) and then normalized that by the total repository size. Thus, we received a relative frequency for each construct for every repository.

We conducted a statistical analysis on the log-transformed data using Student's t-test, where we compared the relative frequencies between the machine learning and

[1] https://docs.python.org/3/reference/grammar.html

the traditional repositories. Of the 86 different code constructs as part of the formal grammar, we found significant differences ($p < 0.05$) for 79 different code constructs (see supplementary for the complete listing).

From this we concluded that there are, in fact, structural differences between the two development paradigms. Certain code elements seem to appear more frequently while other seem to be less common in machine learning code. Thus, we continued to investigate whether developers are implicitly or explicitly aware of these differences and how they incorporate this in their behavior. Thus, we conducted the following eye tracking study.

3.1.3 Methods

To determine the reading behavior of developers for Machine Learning code, we adopt and adapt the methodology proposed by Weber et al. [C7], which relies on the structured nature of software code. We expand upon their methodology, though, with a systematic selection of the code fragments to enhance comparability. The following section describes this general methodology and any changes to the process which we took to address our research questions.

3.1.3.1 Study Design

We performed an in-person, within-subject user study using eye-tracking data from developers while they read a selection of code fragments. We used a graphical user interface nearly identical to that in the study of Rodeghero et al. [254], which uniformly displayed the code. Like in the original interface, we used no syntax highlighting or other visual support mechanisms, as this could impact the gaze, e.g., via preattentive perception. At this point, we do not yet know which code structures we could or should highlight, as this is part of the potential implications of our study, so we opted for the highlight-less approach. Nonetheless, the UI displayed the code in a monospaced font, as is typical for code editors. This also helped with defining clearly distinct areas of interest later on. We also excluded any comments from the code, as they could draw a disproportionate amount of attention and thus skew the results. See the work of

Rodeghero et al. [254] for a further discussion of the benefits and drawbacks of this presentation. As a *within-subject* design, each participant used the UI to read each code fragment described below.

3.1.3.2 Procedure

Besides displaying the code, the interface served a secondary function: to create engagement of the study participants with the code. The participants had the task of summarizing each code fragment's key functionality and purpose in their own words. In this way, we required participants to read the code carefully. We asked them to type their summary directly in the interface, which offered an input field next to the code on the right half of the screen. We also used the think-aloud protocol [275] to understand their behavior better. After fragment summarization, they moved on to the next code fragment. We limited the total duration of the study to 40 minutes, i.e., five minutes per fragment, to prevent effects due to exhaustion. However, this only served as a guideline for the participants; they were free to move on from a code fragment if they believed they had sufficiently summarized it. Beyond this, we also collected additional feedback about the code fragments, demographic data, and prior knowledge with a questionnaire at the end.

3.1.3.3 Selection of the Code Fragments

In total, we selected eight Python code fragments for our study. We only used code written in the Python programming language, which we previously also selected for the static analysis due to its popularity in general but also for machine learning [301]. To facilitate the comparison in *RQ 3.2*, half of these was code that uses Machine Learning in some capacity, while the other half was traditional code without Machine Learning.

Since code quality can differ widely in software projects, we selected our code fragments from Python libraries with a large user base. The idea behind this is that these libraries typically have fairly high standards concerning code style and quality and act as a role model for other developers. Therefore, we picked our fragments evenly from the repositories of the TensorFlow[1] [1] and scikit-learn[2][226] libraries for ML code and from the matplotlib[3], flask[4], and request[5] libraries for the non-ML code. For the ML code examples, we made sure to pick code that uses multiple different ML models using both unsupervised and supervised learning. We looked for similar pairwise ML and non-ML fragments to enable gaze pattern comparison. The first criterion for this was the length in lines of code where we paired fragments of equal length. We also had to consider our screen size since we wanted to minimize interfering variables, such as scrolling through long code examples. Therefore the code fragments did not exceed 60 lines of code.

We followed the approach by Shaffer et al. [269] and Weber et al. [C7], who used the constructs from the programming languages' formal grammar to determine areas of interest in the gaze data. While we had previously determined that different constructs occur at different frequencies, picking code that would accurately reflect these differences would skew the gaze dwell times. Thus we aimed for a similar distribution of the constructs for each pair of ML and non-ML code fragments while keeping distinctly ML and non ML functionality in the code. Table 3.1 lists the frequency of grammatical construct in each pair of code fragments. Since not all 84 grammar constructs were present in the code, the following evaluation will deal exclusively with those present. For each participant, we displayed ML and traditional code fragments in an alternating sequence.

Finally, we selected two sets of code fragments to account for the different expertise levels of novices and experts. This allows us to probe **RQ 3.3** further. As Kahney [140] discuss, choosing a single, simple task can easily lead to task performance becoming indistinguishable between the groups. In a study by Gugerty and Olson [102], they compared novices and experts using the same LOGO code fragments with only minor modifications for the experts, but also discuss that this approach has the issue that with such similar code, one cannot fully explore the experts' skills. Using two structurally similar sets of different complexity, we hoped not to overwhelm the

[1]https://www.tensorflow.org
[2]https://scikit-learn.org
[3]https://matplotlib.org
[4]https://flask.palletsprojects.com/
[5]https://docs.python-requests.org/

novices and, at the same time, ensure that the experts read source code appropriate for their prior knowledge, thus avoiding the effects of mental over- and underload. To gauge how effective the selection was, we collected feedback on the code fragments using the questionnaire, qualitatively using free text answers, and quantitatively using a five-point Likert scale, where participants provided their perspectives on how complex and challenging they considered the two types of code fragments. Furthermore, we collected the code summarizations each participant wrote to gauge whether they understood the key concepts and actions in the code fragments.

3.1.3.4 Apparatus

While reading and summarizing the code, we recorded the gaze of our participants using a *Tobii 4C* on-screen eye tracker, which provides data at a sample rate of 90 Hz. The tracker was attached to a 20” monitor, set up in a well-lit room without outside distractions. We calibrated the tracker for each participant individually before reading the code.

3.1.3.5 Participants

We invited 18 participants from local companies and academic institutions via personal and professional contacts. For their participation of up to one hour, they received compensation equivalent to 10 USD. Of the twelve experienced and six novice developers who participated in our study, half were female, and half were male, with a mean age of 26 years (SD: 2.78). With a background in computer science, they were either in the final year of their Master’s program or had a completed Master’s degree and were now either working in industry or pursuing a Ph.D. In addition, two-thirds of the experts reported currently working on one or more projects involving a substantial amount of ML. The remaining third was not actively working on such a project but had prior experience with the technology.

Our distinction between novices and experts is based on participants’ self-assessment during recruitment and is corroborated by additional background information which the participants provided in the questionnaire. For example, all novices but one listed their programming experience to be between three and five years, while most experts have at least five years of general programming experience. Focusing on the Python programming language, all participants were familiar with it, although all experts indicated a high or very high level of expertise (5-point Likert scale), while the novices rated their knowledge as low to average. The same applies to prior knowledge about Machine Learning, where the two groups are discretely split along the middle of the Likert scale.

3.1.3.6 Data Processing

The structured nature of software code and its formal division into defined structural elements allows for very defined areas of interest. In addition, the presentation using a monospaced font gives clear bounding boxes for gaze targets. After collecting the gaze data in the user study, we applied an analytical approach as follows, in line with Weber et al. [C7]. For this process, we recorded how many gaze points fell into each bounding box and could then determine what grammatical construct in the code and, thus, to which programming concept each gaze point belongs. Figure 3.2 shows an example of the subdivision. We then counted the number of recorded gaze points per area, thus giving us a measure of the dwell time for each area and, when accumulated, for each type of code element.

Not all participants spent the same time reading each code fragment, so we consider only the relative percentage of gaze points per participant. Furthermore, different structural code elements take up more screen space, so naturally, they attract more gaze points than smaller code elements. Thus, we also divided the gaze point percentage by each code fragment's relative area. Since we use a monospace font, this automatically also normalizes the data against character count. Thus, the resulting data is normalized against temporal and spatial factors and allows for a comparison between groups and code elements to answer the research questions.

3.1.4 Results

In the following section, we will summarize the results of the user study and the analysis of the recorded gaze data using the methodology described in the previous section and the literature. To determine differences in the participant's code-reading behavior, we first used heatmaps and a replay of the participants' gaze paths to gain a high-level overview of the data. Second, we used a quantitative method to determine the dwell time for each area of interest, i.e., each syntactical code structure, which allowed us to find general reading patterns and specific parts of the code which caught the developers' attention. After normalizing the gaze to accommodate user-specific offsets, we visualized the heatmaps using the kernel density estimation (KDE) of the seaborn library[1] with the Epanechnikov Kernel. We used Silverman Algorithm [4] for bandwidth optimization, leading to heatmaps, as shown in Figure 3.1. We overlaid each heatmap with its respective code fragment to better determine which part of the code participants looked at.

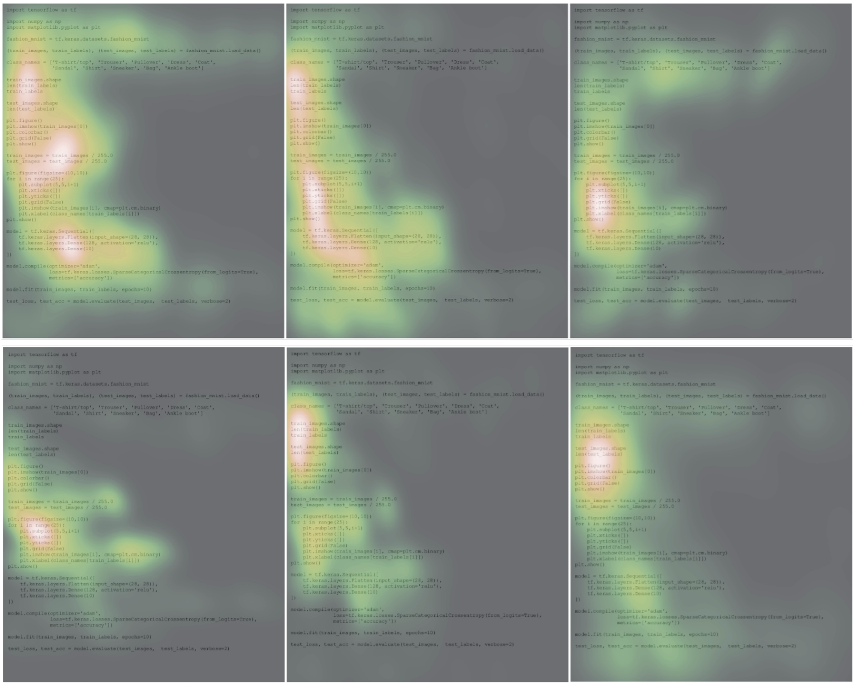
While these visualizations already suggest some interesting patterns, purely accumulated dwell times may not accurately reflect, for example, frequent jumps between the same points, thus, we also employed a tool for replaying the gaze tracks. With this tool, we could visualize the path of gaze points across time to review the eye movement across code fragments. This showed the reading progression of our participants, clearly indicating, e.g., what elements participants looked at first. It also provided a first glance at which elements were most relevant, as indicated by a long dwell time.

3.1.4.1 Reading Behavior for Traditional and Machine Learning Code

To address the first question at the core of this investigate, we investigate the differences in the reading behavior for different code fragments, specifically whether code with ML elements is read differently to code without. When we compared the reading behavior of different developers, we could already see different search strategies being used by participants. Replaying the participants' gaze, we saw that several participants read code fragments sequentially, top to bottom, showcasing patterns akin to Nielsen's F-patterns [209]. The F-pattern means that the reader first performs horizontal scans of the document and then continues with vertical scanning. Those who do not show the F-pattern behavior ignore or skim the majority of the code and jump to areas of seeming interest, where they remain for a considerable amount of time.

Reading behavior without F-patterns is much more prominent in code that implements ML models, while the traditional code is read sequentially for the majority of

[1] https://seaborn.pydata.org



```
import tensorflow as tf

import numpy as np
import matplotlib.pyplot as plt

fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

train_images.shape
len(train_labels)
train_labels

test_images.shape
len(test_labels)

plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()

train_images = train_images / 255.0
test_images = test_images / 255.0

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=10)

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

Figure 3.1: Heatmap visualization of different search and reading strategies of the developers. A replay of the gaze tracks showed that the novices (top) showed mostly sequential reading, while the experts (bottom) jumped to selective points of interest.

the time. Of the eight code fragments per participant, we observed sequential reading in the traditional code 46 times and 28 times in ML code. Selectively reading ML code fragments also includes backtracking, i.e., jumping back to parts of the code that participants had previously read. This is also much less pronounced for the traditional code fragments.

To further investigate which parts of the code are seemingly more important, we used code's grammatical structure to analyze at which elements the participants look at the longest and, thus, may be particularly interesting. Therefore, we subdivided the total text area into small rectangular areas of interest for each type of code element and determined the dwell time for each, see Figure 3.2. We then normalized the dwell

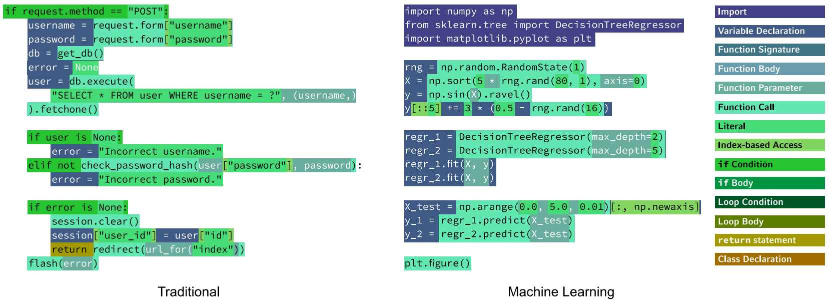

```
if request.method == "POST":
    username = request.form["username"]
    password = request.form["password"]
    db = get_db()
    error = None
    user = db.execute(
        "SELECT * FROM user WHERE username = ?", (username,)
    ).fetchone()

    if user is None:
        error = "Incorrect username."
    elif not check_password_hash(user["password"], password):
        error = "Incorrect password."

    if error is None:
        session.clear()
        session["user_id"] = user["id"]
        return redirect(url_for("index"))
    flash(error)
```
```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt

rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))

regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_1.fit(X, y)
regr_2.fit(X, y)

X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)

plt.figure()
```

Figure 3.2: Examples from each type of code and how they were divided into discrete areas of interest based on the formal grammar of the Python programming language.

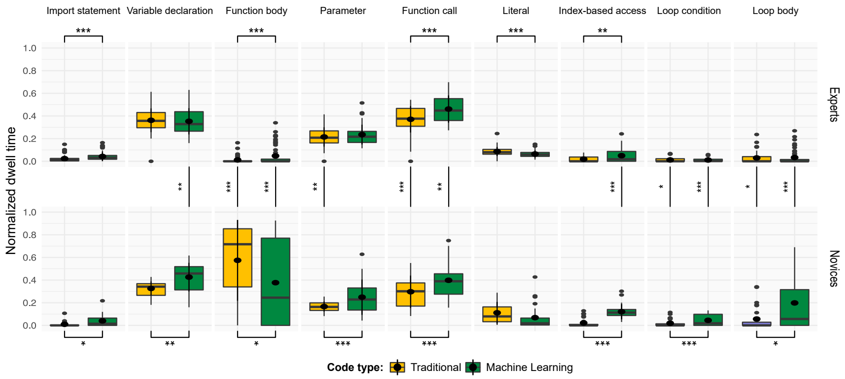


Figure 3.3: Distribution of the relative number of gaze points per area of interest for both participant groups and both types of code fragment. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$)

time to account for differences in area and how long different participants spent for the whole code fragment. Figure 3.3 shows the normalized dwell time for the two groups in our study and the different code elements for source code that does and does not use ML. This already visualizes that certain fragments in the code receive relatively more attention than others.

Considering that the two groups, experts and novices, had different code fragments according to their skill levels, we analyzed the two groups separately. In addition, a first inspection also showed that participants entirely ignored some of the code structures in some test conditions. Thus, we first created a binary distinction at which code fragments novices and experts had looked at. This analysis showed that participants ignored some code elements, namely conditional *if*-statements, class definitions, and function definitions. However, this is an infrequent behavior. Return statements, while similarly infrequent, received more attention. Here, one of the participants even mentioned that they consider return statements an important indicator for the function of a code fragment. Based on these initial insights, we will not interpret behavior for code that participants did not read in any of the study conditions in the further analysis since we cannot determine meaningful statistical results beyond the binary distinction, see Figure 3.3. This resulted in nine types of code elements for further analysis.

To systematically investigate the question of whether there are differences between the two types of code, we compare the mean normalized dwell time for each code element between the code fragments with and without ML, see Table 3.2. Based on the results of a Shapiro-Wilk test, we then performed a paired t-test or Wilcoxon test for all remaining code elements to determine which statistically significant differences between *Traditional* and *ML* exist for the two groups, *Novices* and *Experts*, independently. There are several significant differences between the types of code for both groups, see Table 3.2. We note that these differences are not the same for both groups.

We found similar differences for function, import statements, and index-based data access, with which both groups spend significantly more time. The experts also spend significantly less time on literals in ML code. On the other hand, the novices looked more at function parameters and variable declarations in the ML fragments.

This effect of the type of code appears to be consistently present for both novices and experts alike. Which code elements appear relevant does differ between the groups, though, which warrants further analysis of the two groups.

3.1.4.2 Reading Behavior between Experts and Novices

To answer the second research question, we compare the six novice developers and their gaze behavior with that of the twelve that report a higher degree of experience with Machine Learning. While these two groups read two different types of code fragments, we ensured to select syntactically similar code fragments of roughly equal length and structure (see Table 3.1). Based on their feedback, both groups also considered the code fragments semantically to be of similar, medium difficulty, as indicated by the median of the Likert scales being the neutral option and no significant differences (Mann-Whitney-U-test, $p = 0.294$) between the study conditions.

Concerning the search strategies, it was noticeable that of the six novice developers, four showed F-patterns across both types of source code, while the other two sometimes followed this strategy and sometimes jumped with their gaze. We could, however, not determine how the code influenced the pattern concerning the type of code when they would choose one or the other strategy. In comparison, the experienced developers also applied the sequential, F-pattern-based reading strategy for the traditional Python code but for the Machine Learning code only at most half, typically three to four, were scanning through the code fragments while the other half was selective in their reading. It is worth mentioning that all experts applied both strategies at some point. This indicates that expertise may come with knowledge about which parts of the code are worth looking at and which are not.

When asked what aspects of the code the participants deemed most helpful to foster understanding, almost all novices mentioned clear names of the declared variables. Comments were also listed, although, as previously mentioned, we removed them from the code fragments. For the experts, seven mentioned the names of certain models in the code and especially in the import statements to be helpful. This also showed in the reading behavior where the less experienced participants, who selectively read the ML fragments, often focused not on ML-specific keywords, unlike the expert, but instead paid attention to more commonly understood terminology like the use of plots to visualize results. This effect was visible in relatively longer dwell times and was corroborated by the feedback in both the code summarization and the questionnaire. In comparison, these parts were almost always skipped by the expert developers who were, as they mentioned, more interested in more advanced terminology, like the names of the ML models used in the code. Based on the feedback in the questionnaire, the novices did, however, notice some of these ML-specific keywords but, as they were sometimes unfamiliar with them, looked elsewhere for cues to understand the code.

Beyond this qualitative feedback, we also performed the quantitative analysis of the gaze data for differences between expertise, as listed in Table 3.3 and visualized in Figure 3.3. One difference that immediately becomes apparent is the fact that the distribution of gaze data of the novices is more spread across the different code elements. Experts, on the other hand, focus the majority of their attention on function calls and their parameters, as well as variable declarations, while the other elements play a relatively little role. Still, function parameters and variables declarations are similarly important for the novices, but the function calls receive significantly more attention from the experts, both in traditional and ML code fragments. For function parameters and variable declarations, there appears to be a certain dependence on the type of code: experts value function parameters significantly more than novices in traditional code, while novices spend more time with variables in ML code. Beyond this, control flow via loops offers consistent differences, where novices look at loop conditions and the loop body significantly longer. The same also applies to the encapsulation of code in functions, with novices spending a significantly larger portion of their gaze on the body of such functions. Lastly, the parts to do with index-based data access in lists or similar data structures are notable. Here, novices spend little time reading them in the traditional code examples, just like the experts, but spend significantly more time on them in the ML code. The answers in the questionnaire provide some indication as to why this might be: the Python programming language has a syntax that goes beyond simple numerical access of list elements and allows, for example, to select ranges or “slices” of data. While this syntax is often used in ML code, it has less frequent applications in traditional code. For this reason, two novice participants mentioned that they found this unfamiliar syntax challenging.

3.1.5 Discussion

Our results show differences between the development paradigm and the respective source code and between the developers of different levels of expertise, thus providing answers to RQ 3.1 – 3.3. The data suggests that developers are aware of the structural differences in machine learning code and have developed behavioral strategies for different scenarios and code types. Specifically, we saw different search strategies when reading code, depending on expertise, where sequential scanning in F-patterns was more prominent for novice developers and depending on the development paradigm. Here, code without ML was more likely to be read in order from top to bottom. These two observations may very well be connected since the expert could be aware that ML code warrants a distinct reading strategy, while the novices fall back to the same way of reading they are used to from traditional code and other areas. If this is the case, it implies that certain aspects of the ML code are either trivial or entirely irrelevant to people with prior ML knowledge. In turn, this has implications for tool and framework design, which should strive to eliminate irrelevant steps by abstraction so that developers do not spend time on them. However, as this distinction was not completely binary, it may also be the case that some information is only unnecessary in some use cases and for some people, so flexibility and control over what is hidden should be considered. An example from which we can take inspiration is how modern IDEs handle import statements: since they are often automatically generated and considered to be of little relevance, they are typically “folded,” i.e., displayed as a group in a single line. The developer can expand this presentation, offering the necessary flexibility should a developer be interested in specific import statements. Similarly, an IDE for data-driven development could automatically detect a sequence of preparatory or visualization steps and then collapse them into a single line labeled “data preparation” or ”data visualization,” respectively. An alternative to this would be automatic refactoring, which likewise detects these common blocks and suggests moving them into a separate function, increasing encapsulation and, thus, adding to code readability and maintainability.

Sequential reading behavior may lead to issues with attention in our study but also in general: in many of the code examples we surveyed for our study, the first part of the code is preparatory while the latter parts deal with the Machine Learning parts. This is a widespread pattern, but the nature of F-pattern-shaped reading could mean that the novices spend considerable attention on early, less interesting parts and have little attention left for the learning models at the end. At the same time, this could explain why experts often skip early parts and directly jump further toward the end. As mentioned above, one option is to minimize the space these parts take up could alleviate this. Additionally, by extracting common steps (e.g., data preparation, model configuration, training), an editor could provide a table of contents, allowing one to quickly jump to the relevant parts without the need to scan the complete source code. In notebook-style editors, we could take this further, since documentation and code are often mixed, by automatically generating fitting headings for the steps in the process.

Another question regarding the sequence of the code and the relevance of those steps remains whether this observation depends on the Python programming language and the summarization task. In different situations, developers may have different goals when reading code, e.g., finding errors, finding new solutions, or communicating with collaborators. All of these might come with different reading strategies. Thus, the differences we observed in our study may be specific to the code summarization task and other activities will show different reading patterns. Regarding the impact of the Python language, we think its popularity makes Python a good choice for a first analysis. Furthermore, its simple syntax makes it fairly approachable and no participant severely struggled with the task. Given that it was designed as a general-purpose language and not specifically for data-driven development, some of the steps, e.g., the re-restructuring of data, can sometimes be fairly verbose without offering new insights. We argue that this might be why experts can use their experience to identify these areas as irrelevant quickly. Meanwhile, novices needed help understanding the specific syntax and spent more time with these parts without getting more valuable information. A syntax that is easier to understand or explore would be the obvious solution but needs to be embedded in the existing programming environment. Of course, *tool designers must strike a balance to ensure the interaction is sufficiently versatile, but the generated code remains understandable and ideally helps novices learn what is going on*. Furthermore, *any change in syntax should be flexible enough to facilitate learning so that novices can, over time, gain the expertise to use more compact, efficient means of data access*. Thus, UI designers should refrain from forcing novices into using certain presentations without alternatives. Beyond language and tool design, there may be similar effects due to sub-optimal framework design, where steps that could be abstracted need to be explicit in the code. A next step could be a comparison of popular libraries and how they are received and used. Particularly those that offer the same functionality across different programming languages, like TensorFlow, could yield interesting insights into the effect of the underlying language.

Another factor influenced by high prior knowledge is which code structures is of interest, where experts appear to know the important terms that allow them to quickly grasp the meaning of a code fragment, for example, via the imported libraries and the ML terminology. Meanwhile, novices need significantly more time, relying on control structures, e.g., loops, to follow what the code does. Because often Machine Learning code uses fewer of these control structures — after all the decision logic, and thus conditions are learned from the data — this new development paradigm may offer new challenges for understanding unknown code. Developers who cannot rely on the high-level structure must either understand the data, which often requires domain knowledge, or focus on specific details like known terminology, as we saw from our expert developers. However, the appropriate presentation could help reinstate high-level structures to data-driven development. Since many ML systems follow a general structure of a data pipeline, with data sources as input on the one side and a model as output, we could leverage the sequential nature for presentation: graphical programming already commonly uses blocks, which represent individual steps in the data processing, and connections, which denote the data-flow. However, the existing graphical tools (e.g., [26, 68, 247]) for data-driven application development are more niche. In principle, this should also be easily possible in computational notebooks, but the experience shows that maintaining an overview of data flow in notebooks, where no execution order is enforced, can be challenging [66, 147, 148]. *Explicitly visualizing data flow will assist developers and mitigate the reliance on certain code structures for novices and support debugging for both novices and experts*.

When data defines how the software works, it is not surprising that knowledge of the data itself and the code that manipulates said data becomes more relevant, which might be why participants looked more at index-based accesses in the code. When asked what parts of the code were interesting or helpful in the post-study questionnaire, participants mentioned the names of the data set. Naturally, in our study, the experts would have an advantage here, as they are typically intimately familiar with common data sets like IRIS or MNIST, but many of the novices also voiced familiarity with those data sets, and none of them listed the data set as detrimental to their understanding. For real-world applications, however, data sets are usually more complex and require domain knowledge, so *support for data exploration remains highly relevant for understanding ML systems*. This also increases the reliance on proper documentation, which may explain the increasing popularity of more literate programming.

In addition, one important question is how we can bridge the divide of expertise. Since the experts learned their more selective and focused reading over the years, we cannot expect novices to pick up these skills immediately. Thus, *developers will benefit from more visual guidance, e.g., via visual highlighting*. This may support novices in acquiring the selection skills of experts more quickly. Our data can be a basis for selecting which code elements need to be visually more salient, e.g., data access, and which can be more subdued or moved into configuration files, e.g., specific parameters and literals. In addition, programming education can also contribute by emphasizing the ability to read code with an eye for the relevant bits. Ultimately, the goal should be that truly irrelevant code is either hidden away or takes up little mental load such that novices are guided to the relevant code, and experts benefit from a less cluttered code base.

3.1.5.1 Limitations

The differences we observed in the code structure level depend on the code fragments' selection. While we chose our code examples from popular libraries and tutorials, these may not reflect the reality of software projects in the field of data-driven development. The same applies, as previously mentioned, artificially picking structurally similar code to facilitate comparability, even though the larger code landscape has known differences. A further, dedicated in-depth study of code structure beyond just grammatical construct might lead to interesting insights. For example, it could tell us whether the structure we noted, with many preparatory steps at the beginning and the seemingly interesting block in the second half, is common and how we can use abstraction to reduce the necessity of parts that expert developers mostly ignore. In addition, as we saw, some of the syntactic structures were disregarded entirely, which may be highly specific for our code fragments. Thus, we argue that the next step will be extending our insights by investigating real-world software projects.

Furthermore, the necessities of an experimental setup, e.g. the removal of syntax highlighting offers a limitation as it means that the presentation deviates from how it would be done in the wild. As this is common practice for experimental setups, as described above, it allows for a reasonable comparison with prior work. Since it is consistent within the study, it likely also does not affect the comparative results, but an additional validation in a more realistic setting is desirable. Likewise, the precision of eye tracking is an important factor, as noise can easily skew the data. However, the qualitative feedback of the participants as well as the replay of the gaze tracks corroborates the differences in the quantitative data.

In addition, the comparison of expertise in these scenarios, as in our study, remains quite challenging, as identical pieces of code may over- or underwhelm developers leading to different reading patterns. While our efforts to select syntactically and semantically similar code fragments appear to be successful, since the feedback on their complexity was comparable across all participants, it limits the generalizability of our results and any interpretation should consider this context. Larger code-bases will make it unfeasible, though, to perform such an analysis as a laboratory experiment and would require a field study, which comes with its own logistical challenges. Our experimental setup, so far, only observes these reading patterns of the code as-is. A way to validate our findings could be to adapt the presentation, e.g., highlighting or selection, or by excluding certain code constructs, to see how this affects reading behavior, allowing us to infer the importance of individual code constructs. A long-term study could also add further insights into the effect of expertise by observing how reading behavior changes over time with growing experience. Lastly, the sample of participants for our study was limited and unbalanced between novices and experts. While the results already show some statistically significant differences, a larger pool of participants would strengthen these findings.

Table 3.1: Absolute number of each grammatical code construct per code fragment for both novices and experts. We paired ML and non-ML code fragments based on a similar distribution frequency of those constructs.

	Code for novice developers								Code for expert developers							
	Pair 1		Pair 2		Pair 3		Pair 4		Pair 5		Pair 6		Pair 7		Pair 8	
	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML
Import statement	2	-	2	2	-	-	-	-	3	5	3	1	8	3	7	2
Variable declaration	4	2	8	2	9	8	7	13	7	22	9	9	19	11	12	14
Function definition	-	1	-	-	2	1	1	3	-	-	-	-	2	-	-	1
Function body	-	1	-	-	2	1	1	3	-	-	-	-	2	-	-	1
Function parameter	2	4	6	4	14	6	28	10	17	30	14	9	18	12	20	26
Function call	3	2	7	5	11	9	39	3	21	28	21	6	18	14	18	27
Literal	3	3	2	3	11	12	70	15	22	41	14	4	56	18	24	36
Index based access	1	-	4	-	3	5	12	-	4	7	2	1	1	-	-	-
if block condition	-	3	-	-	-	4	5	8	-	-	-	-	-	-	-	1
if block body	-	2	-	-	-	1	5	7	-	-	-	-	-	-	-	1
Loop condition	-	1	1	1	2	-	-	-	1	-	-	-	2	1	-	1
Loop body	-	1	1	1	1	-	-	-	1	-	-	-	-	1	-	1
Return statement	-	1	-	-	-	1	-	1	-	-	-	-	2	-	-	1
Total lines of code	9	7	16	9	20	23	38	41	51	59	30	30	60	39	59	60

Table 3.2: Statistical analysis of differences between the dwell time in various code elements in ML code fragments vs. non-ML code fragments for both participant groups. All Wilcoxon-test are indicated with a $^{\triangle}$. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$)

	Novices						Experts					
	Normality test		t-test / Wilcoxon-test				Normality test		t-test / Wilcoxon-test			
	W	p	t/W	df	p		W	p	t/W	df	p	
Import statement	0.866	.059	3.593	5	0.017	*	0.882	0.009	$78^{\triangle}$	11	< 0.001	***
Variable declaration	0.968	.893	4.9127	5	0.004	**	0.970	0.675	-0.507	11	0.620	
Function body	0.823	.017	$0^{\triangle}$	5	0.031	*	0.925	0.076	8.148	11	< 0.001	***
Parameter	0.957	.739	8.3604	5	< 0.001	***	0.986	0.972	1.945	11	0.079	
Function call	0.924	.319	10.931	5	< 0.001	***	0.951	0.281	5.569	11	< 0.001	***
Literal	0.951	.646	-2.3253	5	0.068		0.925	0.077	-5.785	11	< 0.001	***
Index-based access	0.875	.076	7.3625	5	< 0.001	***	0.898	0.020	$78^{\triangle}$	11	0.003	**
Loop condition	0.900	.161	20.759	5	< 0.001	***	0.955	0.353	-0.210	11	0.838	
Loop body	0.824	.017	$21^{\triangle}$	5	0.031	*	0.959	0.427	0.461	11	0.653	

Table 3.3: Statistical analysis of differences in the normalized dwell times between the two participant groups, novices, and experts, for both traditional code fragments without Machine Learning and those with Machine Learning. All Mann–Whitney U tests are indicated with a $^{\triangle}$. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$)

	Traditional Code Fragments						**Machine Learning Code Fragments**					
	Normality		**t-test / MWU-test**				**Normality**		**t-test / MWU-test**			
	W	**p**	**t/W**	**df**	**p**		**W**	**p**	**t/W**	**df**	**p**	
Import statement	0.752	< 0.001	$52^{\triangle}$	17	0.151		0.892	0.042	$38^{\triangle}$	17	0.892	
Variable declaration	0.974	0.873	1.825	15.781	0.087		0.976	0.893	-3.168	10.637	0.009	**
Function body	0.650	< 0.001	$0^{\triangle}$	17	< 0.001	***	0.699	< 0.001	$0^{\triangle}$	17	< 0.001	***
Parameter	0.922	0.143	3.706	11.721	0.003	**	0.969	0.782	-0.919	14.272	0.373	
Function call	0.907	0.077	5.316	15.696	< 0.001	***	0.933	0.223	2.957	15.797	0.009	**
Literal	0.952	0.464	-2.184	11.044	0.051		0.904	0.066	-0.330	7.265	0.751	
Index-based access	0.972	0.828	-0.684	8.269	0.513		0.941	0.302	-5.645	7.296	< 0.001	***
Loop condition	0.958	0.558	-2.464	9.703	0.034	*	0.845	0.007	$0^{\triangle}$	17	< 0.001	***
Loop body	0.935	0.240	$12^{\triangle}$	17	0.024	*	0.751	< 0.001	$0^{\triangle}$	17	< 0.001	***

3.2 Tooling Landscape for Data-Driven Development

This section is based on the following publication:

Thomas Weber and Heinrich Hußmann. "Tooling for Developing Data-Driven Applications: Overview and Outlook." In: *Proceedings of Mensch Und Computer 2022*. MuC '22. Darmstadt, Germany: Association for Computing Machinery, 2022, pp. 66–77. ISBN: 9781450396905. DOI: 10.1145/3543758.3543779

As the results above highlight, data-driven software differs from traditional software in many aspects. Additionally, the previous chapter demonstrated how good tooling can benefit the development experience and performance. Yet, even though the paradigmatic change is apparent, developers often still rely on tools that were build for traditional software development, like simple text editors, command line interfaces and integrated development environments (IDEs). These do offer a wide array of support mechanisms, e.g. for dependency management or debugging. However, the challenges of data-driven development, e.g., management of a large volume of data and its exploration, are often not a focus of these tools and, for example, traditional debugging with breakpoints can only get a developer so far when the programs behavior is embedded in the data and not in the lines of code.

Using existing tools certainly works to a degree, and those who pioneered Machine Learning and those who now use it had and have to build on top of existing technology. However, if the data-driven development paradigm is to be carried into everyday development, now may be a good time to consider and evaluate whether tools that drive software development since its early beginnings are still adequate for this change in paradigm, new practices and a broader audience that may not have a traditional programming background but wishes to participate in the development process.

The fact that the simple text editor and IDEs are very common for data-driven development does not mean that there are no alternatives. Both industry and academia have created a myriad of tools of the years to make software development, including for data-driven software, easier and more accessible. With those tools, the question arises how they can benefit the developers (RQ 4). However, before investigating this, we first explore the tool landscape to understand what types of tools have been researched and are actually available even though they might not have found widespread adoption. Their limited use in practice may have many reasons that are hard to pinpoint, particularly when there is only little effort to evaluate existing tools. Moving forward with new types of tool support in the future, it is crucial to understand what has already been tried, what tools exist, and what issues have already been addressed, though.

To provide an overview of the tooling landscape and particularly systematic evaluations of tools for the development of data-driven applications, this paper reports on a review of the academic literature (Section 3.1.3). Through systematic labeling (Section 3.2.1.2) , we were able to determine some popular avenues that have already been explored and evaluated, representing the state-of-the-art, and areas that still remain open (Section 3.1.4). Based on these, we discuss possible future directions and what upcoming tools may look like and what needs to be done to ensure that they are successful (Section 3.1.5).

3.2.1 Method

In order to gauge the state of the literature and to get an overview of existing development tools for data-driven software and their evaluations, we conducted a literature research, following the PRISMA guidelines [219] which we outline in the following section. The goal of this survey is to determine which aspects of data-driven software development must currently rely on previously existing tools are which are already well supported with novel and dedicated tools. We are particularly interested in indicators that tell us how well tools work, that were specifically created to support data-driven development.

3.2.1.1 Search and Filtering

To determine the efficacy of existing tools, we need to rely on form of analysis or evaluation. We therefore queried large publication databases, specifically the ACM Digital Library, the IEEExplore and the proceedings of the AAAI conference, for publications that matched this research focus. The first two represent the major publication databases for computer science literature, while we included the AAAI proceedings specifically, because the AAAI conference is a major venue for related literature that is typically not listed in either of the other databases.

Arguably, there are also tools coming exclusively from industry that try to address the needs of developers. However, either there is little to no evaluations publicly available for some of these or these tools have been evaluated as part of a publication and should therefore show up when querying the publication databases. We accept the possibility of missing out on tools that have been evaluated and the results being available *somewhere* but not as part of a scientific publication, since comprehensively finding such evaluations is unfeasible and their number should be relatively small. This would include especially in-house tools, which may be used at some industry companies to great success, but as long as they are not public, they offer only limited value for developers at large and thus cannot contribute to this overview.

Given that scientific research in the fields of Machine Learning, Artificial Intelligence and data-driven application is very active, terminology can sometimes be in flux. We therefore decided to start our search with broad queries and refine the results later based on manual selection.

We therefore queried each of the publication databases using the following search criteria:

- The term “tool” had to be present in the abstract to limit the results to publications that had tooling as a focus.
- “data-driven“, “machine learning“ or “artificial intelligence“ as the general research field also had to occur in the abstract.
- “developer” or “development” as our intended target group could be anywhere in the text. We decided for this broader query here because an initial scan of promising literature revealed that the target group was often left implicit in the abstract.
- Considering the major pace at which the field is changing, we furthermore decided to only consider publications of the last ten years at the time when

we coducted the search. Tools that are more than a decade old and have not received any attention in the meantime clearly have limited relevance for current and future research and development.

Joining these criteria was trivial for the IEEExplore and the ACM DL, both of which provide a powerful advanced search. For the AAAI database, no such option was available though, so we resorted to downloading the metadata of 9 717 available publications from the last ten years and filtered them ourselves. Naturally, since we could not download these nearly 10,000 full texts, we filtered the third criterion, i.e. the term “developer” or “development” only in the available metadata (abstract, keywords, etc.) for the AAAI publications. Additionally, we omitted the query pertaining to Artificial Intelligence, as this is the topic of the conference series to begin with and should trivially be the subject of all the publications published there.

The results of our query are listed in Table 3.4. The table also provides the number of results, which remained after screening, which we did in an multi-stage process (cf. [219]).

After extracting the list of meta-data for each publication from their respective databases, we first filtered them by title. Since our search focused on tool support for development experts, the primary inclusion criterion here was whether the publication would introduce or evaluate a new or existing tools in this context respectively. At this stage we also excluded all those texts where the title clearly indicated that the tools were not intended for software development but for example for end users of various domains, e.g. software tools for medical diagnoses.

Given the ambiguity and broad use of the search terms, particularly “tool” and “development”, this already substantially reduced the number of relevant publications to about a tenth of the initial search results.

In the second stage we then read the abstracts of the remaining papers, applying the same filter criteria, i.e. include tools that were specifically concerned with the creation of data-driven software.

We excluded publications that only applied data-driven principles or ML to domain problems, e.g. those that apply Machine Learning to medicine, manufacturing, etc. but also those that apply these techniques to, for example, conventional software repositories to automatically detect defects via ML. While the latter certainly fall in the

Table 3.4: Total number of publications throughout the search and filtering process.

	AAAI Proceedings	ACM Digital Library	IEEExplore	**Total**
Records retrieved from databases	80	839	592	**1511**
Screened by title	9	95	67	**171**
Screened by abstract	1	48	27	**76**

intersection of software engineering and data-driven software, the goal of this survey is to determine how to specifically enhance development experience *of* data-driven, not *using* data-driven software.

Technical publications that reported on the development or improvement of algorithms but did not integrate them or provide them in a dedicated tool, we also removed from the list of relevant texts at this point. This left us with a manageable amount of 76 publications in total. A full list can be found in the addendum to this text.

3.2.1.2 Labeling

Using the full text of those 76, we labeled each publication in two ways.

First, we had a number of pre-defined criteria that we were interested in:

- As a rough classification we looked at what kind of tool support each paper describes? Is it about an existing tools, did the authors improve a tool from prior work or did they creates a completely new solution. This provides us with a general overview of the tool landscape in this area.
- Since, as mentioned, adoption of existing tools from academia is limited, we also recorded whether publications went beyond a purely technical description and reported on on real-world application of their tools or any other forms of practical evaluation in the form of case studies, field studies, etc. to gauge the effectiveness of these tools.
- To determine areas in need of additional research, another focus was what aspect of the development process these tools focus on? Since each tool can

support steps specific to data-driven development but also integration into conventional software engineering, we use the terminology of Hesenius et al. and the EDDA process [120]. The EDDA process is an extension of traditional software engineering processes to capture the additional requirements of data-driven development. Specifically, it adds additional steps for assessing the suitability of Machine Learning, *data exploration* and subsequent *model requirements*, *model development*, and *integration*. By categorizing the tools according to the steps in this process, we can determine whether the whole process is similarly well supported or whether some steps receive more attention than others.

Beyond this we performed open labeling of the publications, which we subsequently clustered into categories. The following chapter will list these labels and further results.

3.2.2 Results

Our search and filtering resulted in 76 publications pertaining to tools for data-driven development. The following section will describe these publications in further detail and highlight shared topics and point out differences.

3.2.2.1 Metadata

First we looked at the metadata of the publications, starting with the publication date. While we specifically selected only publications from the last ten years, it is noteworthy that even during this period of time, we can see a trend of increasing research interest. Figure 3.4 shows this steady increase. This also matches the results of similar surveys of various topics related to Machine Learning, which generally find that this development paradigm is becoming more popular in practice and research [334].

This is also reflected in the breadth of domains that publish about data-driven development tools. Naturally, Machine Learning and data-management venues are represented by our sample of papers (16 times), but also other areas of computing, like general software engineering (12 times), education (5 times), and particularly human-computer-interaction (16 times). Considering that tools a form of human-machine interaction, this comes as no surprise, though.

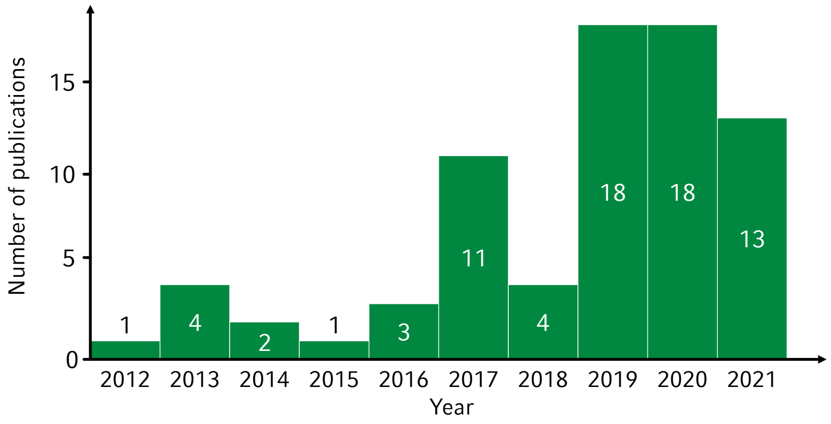


Figure 3.4: Distribution of the publication date of the papers in our selection.

Notable, however, is the fact that our search criteria only yielded a single publication from the AAAI conference series. While, certainly, the complete database of AAAI publications is smaller than ACM DL and IEEExplore, we found the primary reason for this to be that these publications tend to be either highly focused on the underlying technology or report on case studies where Machine Learning was applied to solve a problem in practice.

3.2.2.2 Research Topics

Opposed to that, of the 76 publications we extracted, only one reported on algorithmic improvements in the context of automating and thus simplifying development [73]. **Automating** development in general was a popular topic in the full set of search results, but often with the goal of eliminating the developer from the software creation process – and thus of limited interest to our question of how tools can support developers. However, partial automation also was a topic of interest for 14 of the 76 publications, again with an increase in the recent years, e.g. for entity matching [309], prediction [261] and optimization [214]. Particularly the use of meta-learning [300], or “AutoML” systems, i.e. ML systems that automatically learn their ideal configuration, seem to be of interest for tool developers [195, 270, 309, 325].

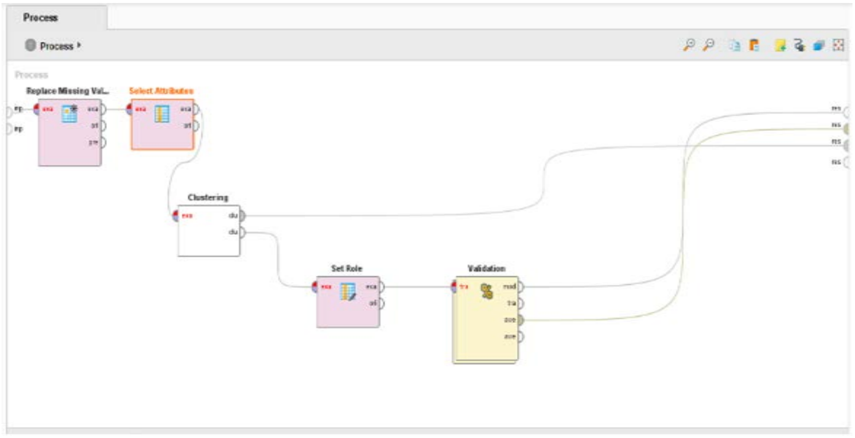
Just as automation is of interest from a technical perspective, they also end up in dedicated tools for developers with the goal of simplifying model development and optimization (e.g. [195, 309]. ATMSeer [310], for examples, extends AutoML systems with visualizations to give developers feedback about the systems performance and progression.

Researchers, however, recognize that complete automation may not always be feasible or desirable [325]. Particularly the general opacity and black-box nature is viewed critically. If even the developers no longer have means for understanding their complex software, it will become hard to debug [158], evaluate [122, 177], and control [274]. Works like “ATMSeer” by Wang et al. [310] or “CertifAI” by Sharma et al. [274] thus attempt to make the advances from the research on **Explainable AI (XAI)** accessible as tools for developers through interaction or examples and visualization respectively (cf. also [97, 158]). In this field, there are also a number of structured evaluations, e.g. a study by Kaur et al. [143] of how XAI tools are used by developers and whether they achieve their goal of informing them.

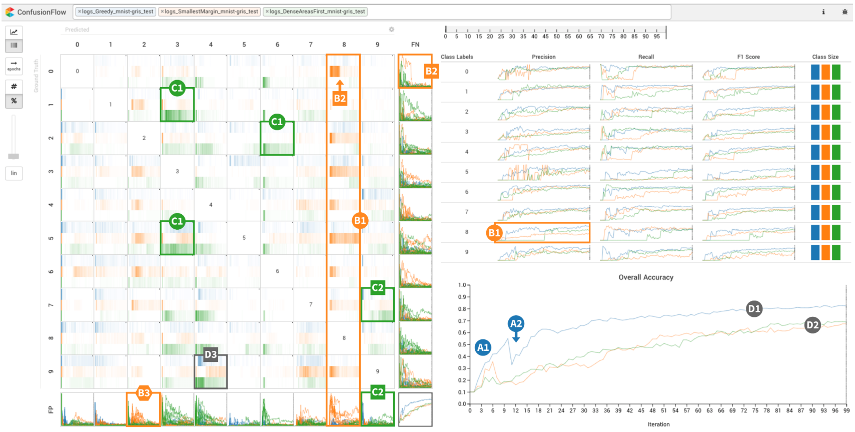
For these explanations, but also for development tools in general, another major topic appears to be how to visualize the complexity of ML. Many researchers recognize that in the traditional, code-based format, it can be hard to understand what is going on. For this reason, 12 publications specifically investigate the use of **visualizations**, e.g. for data preparation [290], development [153, 321] and evaluation [177]. Typically, they use various graphical presentations to offer real-time, and sometimes interactive, visualizations of various metrics of the ML models as shown in Figure 3.5.

In addition, many of the tools mentioned or investigated in 24 of the publications have some visual component a use graphical programming. Especially the graphical programming aspect is often directly inspired by the domain of model-driven development (MDD), like the work by Zhang et al. [336], which aspires to bring MDD and data-driven development together.

This naturally also includes a number of analyses and evaluations of existing graphical tools for data-driven development, like RapidMiner [28, 135, 247] or Orange [276], which Bjaoui et al. [28] consider for novices and Shastri [276] for non-programmers respectively.



(a) RapidMiner (image from [28]) allows the development of a data processing pipeline by composition of functional blocks



(b) ConfusionFlow (image taken from [122]) supports developers with visualizations during performance analysis of classifiers

Figure 3.5: Two examples for tools for the development of data-driven software that heavily rely on a graphical presentation.

Some of these tools have been around for quite a while – in the case of RapidMiner for two decades now [248] – and have been used for various domains (e.g. [104, 159, 189, 204]). It therefore comes as not surprise that the evaluations highlight a number of benefits of these graphical tools, particularly with respect to visual organization and managing complexity. Yet, no single graphical tool has managed to become as popular as code-based tools like Jupyter Notebooks [139].

Kery et al. [153] and Zhao et al. [339] therefore report on attempts to bridge the divide between code and visualization via widgets that allow switching between these representations. The results of their evaluations in the form of a user study indicate that data scientists view this flexibility and the ability to hide and show code on demand very favorable, which may be an important factor for adoption.

3.2.2.3 Supporting the Development Process

As previously mentioned, we also specifically looked into which steps of the development process different tools aim to support. Table 3.5 provides an overview of the different steps of the EDDA process [120] and the publications from our selection that support each step.

Very clearly there are some steps that seem to be of more interest to researchers than others. Especially data exploration, model development and testing and evaluation seem to get a lot of attention and tooling. Some of this may be an effect of our search and screening strategy, since we only queried the databases for tools for data-driven development. Some steps of the EDDA process are in part covered by conventional software engineering, so existing and established tools may suffice.

For data-centric steps in the process there are quite a number of novel tools, however, e.g. for labeling the data where Alaghbari et al. [8] explore the use of gamification or TagRuler by Choi et al. [49], which combines Natural Language Processing and programming by example to automate this usual tedious process.

The concrete implementation of ML models is very heavily supported from various perspectives, be it the selection of the appropriate model [9, 203, 221] and architecture [339] or its implementation, [81, 195, 292] training and optimization [122, 214, 265, 310]. Optimizing ML models is of course also a question of optimizing hardware-specific code

Table 3.5: The steps of the EDDA process [120] and the publications that introduce or evaluate tools that support this step in some way. Publications not listed did not introduce tools to specifically support individual steps of the development but either provided general support or helped with activities related to the development but not captured by the EDDA process.

Activity	Publications
Requirements Engineering	[342]
Specification and Design	—
Evaluation of ML suitability	[72, 342]
Data Exploration	[8, 9, 49, 103, 235, 289, 290]
Model Requirements	—
Model development	[9, 49, 73, 117, 122, 172, 195, 203, 217, 221, 265, 292, 308, 310, 321, 339]
Model integration	[178, 336]
System Test & Evaluation	[42, 97, 122, 123, 158, 177, 274, 310, 338]
Operation & Maintenance	—
End-to-end	[33, 117, 177]

for running the complex models efficiently on GPUs or other specialized hardware. Liou et al's. [172] GEVO-ML is an example for a tool that addresses this particular challenge of ML via automation, detecting and applying certain optimizations.

Optimization of course is also very much related to the testing and evaluation step, where many of the XAI and visualization systems play a role, either in order to give the developers a better understanding of the metrics and machinations [117, 122, 274, 321] or explicitly for debugging [122, 158].

Some of the authors attempt to maintain a holistic view though [9, 33, 206], addressing issues that are relevant throughout the development process like traceability [210] and asset management [132].

Much less in the focus appears to be the question of ML requirements, which according to the EDDA process [120] should include questions about acceptance criteria, i.e. when a ML model is adequate. For the integration of ML systems, we found two publications in our search: Liu et al. [178] which explores how different ML systems and libraries can interoperate via model transformation and Zhang et al. [336] who approach ML tools from the perspective of MDD and highlight the existing integration challenges in the tool infrastructure.

Overall it appears that certain aspects of data-driven development are already very actively explored – whether to a degree of saturation we cannot yet say – while others still provide ample opportunity for improvements. Still, the breadth of tools for many niche and specialized aspects also highlights a certain fragmentation. Instead of integrating solutions into tools with a large user base, researchers currently appear to prefer building their solutions from scratch and as stand-alone solutions. This may be in part due to the nature of research, focusing on small, well defined aspects. Another factor, though, may also be the matter of interoperability and integration, as previously mentioned [336].

There are notable exceptions like Malviya et al. [183], who advocate for a plugin-based architecture, and others [13, 135, 153, 186, 261] who build plugins or extensions to existing tools. However, based on the publications we investigated, the tooling landscape remains in flux, with Jupyter Notebooks – currently – a solid contender as the tool of choice [139]. Alternatively, it is also possible to leverage the existing tooling landscape and adapt it to support the development of data-driven applications too [65, 336], which not only is an efficient reuse of resources but could also lead to a greater integration of software tools and thus counteract a potential fragmentation.

3.2.2.4 Evaluation Methodology

A slow adoption of tools can of course also be the result of insufficient quality and poor usability. We therefore specifically noted the evaluation methods, if any, in the publications. Positively, almost half (31) of the publications we analyzed did perform some for of evaluation.

However, when comparing the method of evaluation, we found that there are three mostly disjoint groups: publications from the more technically inclined venues tend to favor benchmarks with analytical metrics to compare their implementations with prior work (7), while particularly researchers from the field of human-computer interaction rely more on user studies, interviews and surveys to capture to how their tools are perceived (13). A method that both groups occasionally rely on are case studies, i.e. applying their tool in a realistic setting (10). Here we could, however, not reliably determine from the text whether these case studies were selected after the tools were completed or whether the tools were built with the application in mind.

While all these evaluation methods are very much valid to determine the quality of the tools which people build, they each of course only address a limited number of quality criteria. Unfortunately, our analysis shows that only a handful of publications use more than one of those methods, so even though researchers may claim their tool to be high quality, we typically get an incomplete picture, focusing often either on analytical quality or the human perception but rarely both.

3.2.2.5 Related Literature Surveys

Beyond direct evaluation on individual tools, we found 14 publications in our sample that reported not on single tools but on comparative surveys across multiple existing tools, eight of which are from the last two years. This, to us, indicates an increase in reflection on the status quo and a desire to understand the current situation before moving forward [98, 218].

Being from different domains, these publications focused on different aspects of data-driven tools, be it where the tools are used, e.g. in education [139], or how they support specific steps in the development from data preparation [103] to model selection [221] or training and optimization [265]. In contrast to our survey they did not specifically focus on the benefits for development experts but contrasted the expert perspective to that of data science novices [12, 64, 265]. This focus is often motivated by the complexity and opacity of current machine learning models and the process for developing and tuning them [265], which makes existing tools hard to use for less experienced users. At the same time, it is acknowledged that even experts often require a considerable amount of skills and training for effectively using current tools [217] which suggests that there is still much to do to achieve easy to use tools for data science, regardless of the target group.

Nonetheless, these types of comparisons and overviews, particularly when applied to practical, real-word use cases, provide an additional perspective for evaluation the existing tool landscape for data-driven development.

3.2.3 Discussion

From the body of publications, we were able to determine a number of commonalities and shared directions but also highlight some of the outliers and shortcomings in existing scientific literature. The following section will now summarize some of these and draw conclusion regarding tooling for data-driven development in the future.

3.2.3.1 Common Goals

The generally high – and increasing – interest in the topic of data-driven development and tools and methods for its support already suggest that is by now well understood that Machine Learning etc. have moved from academic exercise to be the driver of real-world change and thus needs to be treated as such. This view is shared by many of the reviews and shows in a decent number of evaluations, which also suggest a increase in reflection.

Such introspection will help with transitioning from data *science* to an *engineering* discipline. However, considering how diverse the field still is, with many disjointed methods, processes and a limited consensus on best practices [106], there may be a need for a coordinated effort to consolidate and steer the field similar to the movement that brought us software engineering [21].

Another fact that has been acknowledged is that it can be quite challenging for developers to move from conventional programming to developing data-driven software. While the former builds on a defined sequence of instructions, the latter has many of its functionality and intricacies implicit in the data where it is learned by the software but remains hidden to the developer.

Certainly, the field of XAI is working on this challenge, but, at the current point in time, the preferred target group seems to be end-users [51, 52, 79]. While this yields interesting results, it is still unclear whether end users need or want explainability [80, 312]. Developers, on the contrary, very clearly benefit from opening up the black box that is ML for the purpose of debugging [122, 158], identifying biases [123] and just general fairness [274], transparency [310] and an ethical use of the technology [299]. Focusing some of the XAI efforts on developers first may, therefore, yield more immediate, actionable successes. In addition, given that someone has to develop the end-user explainability, making sure that developers precisely understand what is going on, may also prevent a situation where flawed mental models are propagated and should in turn improve the situation for all (cf. [191]).

3.2.3.2 Common Openings

As highlighted in Table 3.5, not all aspects of data-driven development, in this case represented by steps in the EDDA-process [120], have received an equal amount of attention in the form of publications. As mentioned before some of these areas may already be covered by prior research from traditional software engineering though. In addition, a high number of publications and approaches in one area must not mean that this step has been solved but can also indicate a high need for diverse solutions or many incremental improvements. It is therefore hard to precisely determine, where exactly more needs to be done.

It it, however, noteworthy that the large gaps appear to lie at the interface between data-driven and more traditional software development: there is little to nothing for specifying the requirements of the data-driven aspect in the context of the larger software and similarly little for integrating the data-driven part into the software and pushing it into the world. This may be due to a diffusion of responsibility between traditional and data-driven development, both expecting the other side to handle this. Another possible explanation may be that these are the points of friction where we have not yet found adequate solutions. Both these potential reasons, however, strongly suggests that these areas will require some attention and should not be neglected going forward.

3.2.3.3 Quo Vadis

Research into the development of data-driven explanations certainly is only getting started, though. Nonetheless, the question arises which immediate issues need to be addressed now to steer the field in a good direction. For tooling in particular it is important to make Machine Learning and related technologies accessible to a broad audience to make sure many can participate in a technology that affects everyone. Likewise, we need to determine the major impediments for its common adoption first before we rush into niche solutions for specialized problems.

Based on the literature we described in this paper, we see a number of properties that tools in the nearby future should try to achieve:

Vertical Integration One issue that is not just bothersome but can also introduce an arbitrary number of problems is when a developer has to use a specialized tool for each step in the development. It is no wonder that for traditional software engineering, many professionals use Integrated Development Environments (IDEs), i.e. tools that attempt to integrate support mechanisms for a slew of activities in a single application.

As our literature review shows quite clearly, there are many helpful tools for many important steps in the development process but most of them are stand-alone applications. In order to use them, developers will have to export, import and transform their data, which will be the source of errors.

Arguably, this may very well be an artifact of the fact that academic projects often focus on one aspect only. Still, a growing number of projects, including from academia, provide their support mechanisms integrated into the Jupyter Notebook environment, as one of the most popular tool. Such a close integration into an established platform not only does reduces the likelihood of errors due to for example data-transfer, but also simplified usage and increases usability, which saves time and should positively affect adoption. We therefore encourage this practice for future development efforts.

Horizontal Integration Of course, consolidating tools leads to another open topic: how to integrate data, models, etc. from various tools, libraries and systems. This will require some form of universal interface between applications, which is highly non-trivial, but there are early efforts for interaction between software systems, both in our sample of the literature, e.g. the AI-ESTATE standard [131, 289], or beyond, like the ONNX standard and platform[1] which allows for a shared representation and thus exchange of ML models. Unfortunately, in our literature sample, we could find only very few examples where these, admittedly relatively new, standards are being used.

Currently, the more common practice (e.g. [9]) is to leverage the RESTful architecture, as well established in traditional software engineering, and provide ML functionality as a service with an API. Adherence to this de facto standard method also allows fairly simple interoperability with traditional software systems; a topic that was barely mentioned in our literature sample.

Automation A major trend in the publications from our sample seems to be automation though, with one in five publications introducing a tool that automates some aspect of the development. As mentioned before, complete automation is viewed critically in some of the analyses though, for it will exasperate the opaque nature of ML, which is undesirable for developers and unacceptable in many domains like for example healthcare [126, 224, 268].

This balance between automation and control is of course not new, but in this case of software and data-driven development is somewhat reflexive in that the developers of the automation would potentially automate their own work.

To ensure this balance, we should strive to make the transition across the spectrum of automation as easy as possible. This means that for any step that can be automated, developers should have the choice of letting the computer solve the problem for them, manually solve it themselves or rely on any form of collaborative interaction in between with little overhead for switching between these options.

Graphical and Code-based Tool Another very common theme is the use of visualizations, which were used in a number of publications. They way they are often presented suggests that for many data-driven development has a strong graphical component, be it as a means of tackling its complexity or as a property inherent to this development paradigm.

[1] https://onnx.ai

Naturally, many have leveraged this in some form or another, yielding a number of graphical programming tools for data-driven development. Still, Jupyter notebooks, which are at their core code-based, remain among the most popular tools [139, 149].

Various evaluations in our sample suggests, though, that the graphical aspects provides a number of benefits, particularly when dealing with complexity at the scale of many data-driven applications. Code, on the other hand, is very concise and expressive, allowing a high degree of control and flexibility. Both therefore are valid presentations.

Work like that of Kery et al. [153] already show that these two options must not stand distinct from each other but can be combined. Solutions like that, which allow a seamless transition from established to novel interfaces may convince more people to try interfaces beyond what they already know and use – particularly when integrated into a platform that developers already use, like Jupyter notebooks. The division of Jupyter Notebooks into individual blocks that individually can be switched between graphical and code-based view lends itself especially well for this, but this notion of decomposition has been used to a further degree already. Silva et als. [280] DBSnap++ uses graphical programming, where queries of data are built from individual blocks, or consider RapidMiner [28], which builds the whole data processing pipeline by composing blocks.

In fact, the pipeline-nature of data-driven software lends itself very much to this idea inspired by prior work in MDD, since the data-flow from a high-level perspective is straight forward. To still facilitate the aforementioned control and flexiblity, tools could allow a fluid switch to code [153] for individual steps or even a hierarchical structure where developers can drill down through multiple levels of abstraction per block until they reach the level of detail they require.

There are already a number of approaches for extending computational notebooks beyond their linear presentation [110, 111, 316]. By placing cells as blocks arbitrarily in 2D space alows for more flexibility. Additionally, when each cell is a distinct step in the processing pipeline, this already moves in the direction of combining graphical, block-based programming with computational notebooks.

However, one should not forget that this way of abstraction, composition and general approach to complexity has been state-of-the-art for many years in MDD but did manage to convince the majority of software engineers to use it in everyday practice.

Holistic Evaluation Why exactly MDD is not as widely adopted as it might be for all its benefits is beyond the scope of this thesis, but some recent trends in that field [17] and some aspects of our data may hold some clues how to prevent this for data-driven development.

To be adopted a tool should fulfill certain criteria, among which are that it should provide sufficient benefits at a low cost. For data-driven tools the benefits can include an increase in model and software quality, while the cost can come in work overhead and general usability.

Part of this is often the topic of an increasing number of evaluations in this field of research – a development that is to be encouraged. Still, looking at how the tools in our sample are evaluated, we saw that the majority is only evaluated for one of these two aspects: either some form of usability evaluation or performance benchmarking. However, even if a tool is superior to other in the benchmarks, it is likely that only few people can adopt it if it is only usable by highly experienced experts. Likewise, if a tool is a delight to use but provides only limited practical value, it will likely also be rejected.

This is not to say that tools from technology experts are unusable or those from HCI are technically unsound – with the current evaluations we cannot tell – but to convince potential early adopters, an holistic evaluation from multiple perspective may be more convincing.

This should also encourage and facilitate collaboration across domains, which generally is desirable and should lead to better, more rounded tools. However, such collaboration must be supported on an organizational level also, for example by encouraging more human-centered work at technical venues and vice versa. Considering how tooling and support for developers is distributed across the different venues in our data set, there still seems be a bit of work left to do.

3.2.3.4 Limitations

At the same time, one would assume that an area like software development, where the target groups is capable of building their own solutions, would be self-regulating in that developers will likely build the tools for their own most pressing issues. So, our selection of scientific literature cannot fully reflect the full breath of tooling for data-driven development, since it will not include all the individual solutions that did not make it into a scientific publication. Likewise we also cannot judge the tooling landscape that large corporations might employ internally, as long as they remain behind closed doors.

These tools, however, often are created in an ad-hoc fashion and without publicly accessible evaluations it is hard to determine their actual value and benefit. Particularly for data-driven software, where many factors like unknown biases can be the deciding factor between impressive results and hidden flaws, a healthy degree of skepticism for un-tested and un-evaluated tools is appropriate. Consequently publication, scientific or otherwise, an open-source mentality and, in general, continuous evaluation are all what we should strive for to ensure that data-driven software increases in quality and its development becomes easier, more accessible and more reliable.

That is, of course, not to say that many of the current tools we looked at are necessarily flawed, but with only about half of them providing an evaluation, there is still some room for making their benefits more convincing and improving transparency.

Naturally, our results for these tools depend on the search and filter criteria we applied and can only reflect a snapshot of the status quo. Given how fast the field of ML is changing, many more tools are probably already being developed to address many of the open issues and at the same time the technology may change such that existing tools become obsolete. Given that software developers do still use tools that have been around for decades, the adoption is not a quick though. So, the question really is not which individual tool will be used by how many people but which overarching theme will be prevalent and can convince developers to change habits. These themes, some of which have been described in this paper, are more stable and much more than just trends, but therefore should also be grounded in the results of systematic evaluations of what works and what does not.

3.3 Hybrid Development Tools for Data-Driven Development

This section is based on the following publications:

Thomas Weber, Janina Ehe, and Sven Mayer. "Extending Jupyter with Multi-Paradigm Editors." In: *Proc. ACM Hum.-Comput. Interact.* 8.EICS (June 2024). DOI: 10.1145/3660247

Thomas Weber and Sven Mayer. "Usability and Adoption of Graphical Tools for Data-Driven Development." In: *Proceedings of Mensch Und Computer 2024*. MuC '24. Karlsruhe, Germany: Association for Computing Machinery, 2024, 231–241. ISBN: 9798400709982. DOI: 10.1145/3670653.3670658

As the literature review in the previous section demonstrates, tooling is an area that still holds a lot of potential for supporting software developers with building data-driven applications. This, this section explores how to create a concrete implementation of tools that address some of the challenges which we found in prior work, thus addressing RQ 4. Specifically, we focus on a human-centered perspective. The goal of this is to build a tool specifically with adoption in mind and continuously evaluate whether it meeds the needs of developers.

As part of this, we take the integration of different features into account, both within the tool but also in the larger ecosystem of software developement tools. Additionally, we include the graphical programming paradigm as an example of a less adopted but potentially beneficial paradigm in our evaluation. We selected this paradigm based on the insights from the literature described in the previous section. Furthermore, visualization plays a prominent role in data-driven development, making this paradigm particularly interesting.

Following these requirement, we implemented a development tool across various iterations, starting with a graphical programming interface and then extending it into a multi-paradigm tool that gives developers an increased level of flexiblity and integrates with existing tool infrastructure for computational notebooks while managing the level of perceived complexity.

We evaluated the first, purely graphical iteration in a user study (N = 20) which demonstrated some improvements. We further iterated on this system based on the feedback to allow users to use different presentations and programming paradigms side-by-side and with the option to transition between them seamlessly.

With this multi-paradigm presentation, we address the desire of developers for flexibility, which was mentioned already in previous studies (e.g., Section 2.1) and was reiterated here. We collected further user feedback during the development from professional developers (N = 12), demonstrating that a multi-paradigm interface offers greater flexibility to developers without negatively impacting their perceived mental load. The feedback from the study participants also included specific use cases and workflows that would benefit from the option to use different paradigms at will. Thus, our work not only demonstrates the technical feasibility of extending the Jupyter infrastructure with different user interfaces but also highlights how different paradigms and even multi-paradigm user interfaces can provide benefits for software developers.

3.3.1 Related Work

Non-textual programming paradigms, e.g., graphical programming, as an alternative to the traditional text-based input method, have been around for decades, effectively ever since computers were powerful enough to support them [201]. In this chapter, we will briefly outline these ideas and some of the existing prior work that informed the design of our hybrid development environment, particularly for creating data-driven applications.

3.3.1.1 Literate Programming and Computational Notebooks

Literate programming as a concept describes the idea that programs should be considered as pieces of literature [155] and thus be highly readable for humans. An essential part of this is adequate documentation in an understandable format. This idea is, in fact, quite old already, having been formulated by Donald E. Knuth as early as 1984 [155]. While the goal of having adequate, human-readable documentation easily available has always been present, surveys of software repositories show that even if documentation is present, it can be spread out across multiple sources [284]. Thus, computational notebooks try to facilitate literate programming by making it easy to keep code and documentation closely intertwined. Lau et al. [161] summarized a wide array of existing approaches and variants of computational notebooks, highlighting that the design space for these systems allows for quite some variance.

One of the most popular literate programming tools is Jupyter notebooks [242], which presents software code in interactive cells interleaved with their output. Their interaction allows for relatively rapid exploration and prototyping, making them attractive to skilled but also less experienced developers [173] and for programming education [212, 303].

Project Jupyter is not just the interface but also acts as a platform for different usage scenarios and different target groups [192, 234, 242]. For one, it supports multiple programming languages — hence also the name, which includes the set of three core programming languages **Ju**lia, **Pyt**hon, and **R**. The underlying infrastructure of code execution kernels has allowed it to be extended to support many additional programming languages. The front end is also extensible, allowing developers to add their own commands, interface modifications, and widgets to replace and enhance the original cell structure. Naturally, this has been leveraged in a number of research projects: In the mage system [150], for example, the output of data operations is processed and presented as an interactive table, and changes in the table are reflected in an update of the source code. Here, the multi-modal presentation is limited to the output. Kery et al. [153] highlight that this concept could be taken even further, e.g., using drag-and-drop in the widget but also between widgets and cells. The ODEN system adds widgets that visualize neural network architectures alongside the code of the network. Using these visualizations allows for additional interaction to manipulate the neural network directly and not just with code [339]. The code cell structure, which is popular in many computational notebook systems, also motivated Watson et al. [311] to develop an extension for Jupyter that allows developers to reuse existing code snippets in the domain of molecular biology.

The computational notebook interface is not without drawbacks, though. Firstly, the goal of making software more human-readable and facilitating documentation is not a necessary consequence of the interface. While it encourages good practice regarding documentation by allowing developers to add their documentation directly alongside the code as Markdown, a survey of 1.4 million notebooks from GitHub by Pimentel et al. [234] showed that this promise does not necessarily hold true. Many notebooks are still lacking when it comes to using cells for documentation. Furthermore, Chattopadhyay et al. [43] report on nine pain points that developers have with them and how they try to mitigate them. Among them, they found that developers are dissatisfied with the support in notebooks for activities such as refactoring. As a result, they regularly copy their code back and forth between tools. Consequently, the authors suggest combining different development paradigms to reduce work overhead across tools. Furthermore, both Head et al. [116] and Lau et al. [161] highlight that the linear, sequential presentation of notebooks, as a consequence of the text-based presentation, can lead to issues since the cells can still be executed in arbitrary order. The result problems, like missing or incorrect values, lack of code dependencies, etc., then easily confuse the user.

This demonstrates that, while they are currently very popular, computational notebooks will likely not be the one paradigm for all programming needs. The extensible infrastructure, however, does lend itself as a platform to iterate on how programming interfaces might be designed, so we utilize it in this paper to test out different programming paradigms.

3.3.1.2 Programming Paradigms

While computational notebooks are popular partially because of their simple and fresh take on a programming user interface, they fundamentally follow the principles of textual programming. However, it is unsurprising that software development can occur in many different ways using different programming paradigms. Over the years, there have been many explorations of programming paradigms, including the following:

Visual Programming [36, 201] or Graphical Programming uses richer representation and interaction to define software behavior. Here, the developer uses a visual representation, e.g., following the visual models of the UML [259], to create software. The visual representation can provide greater flexibility than text because it is not constrained to be strictly sequential and is often supported through direct manipulation for effective interaction.

Programming by Example [56, 171, 201] allows developers to demonstrate their desired behavior, and the computer then infers the required instructions to facilitate this behavior. This paradigm requires relatively little programming expertise but benefits greatly from domain knowledge, making it interesting to work with domain experts.

Natural Language Programming replaces the conventional, rigorously structured programming language and allows developers to describe the software and its behavior in natural language. The lack of precision of natural language requires some translation into a format acceptable for the machine (e.g. [176]). However, the familiar interface provides a very low barrier to entry for this development paradigm. In addition, advances in large language models have made this type of development viable for increasingly complex software [47, 314, 326] as discussed and assess in Section 2.2.

Rule-based Programming substitutes describing the behavior of the software by enabling the developer to define rules to constrain the programs' space via logical expressions [54], data-relations [89], etc.

Dataflow Programming is based on the concept of data flowing from one output to an input of another node. Dataflow languages can be represented in a directed graph. It allows partitioning into components by adding phantom input or output nodes and concurrent execution of nodes as input parameters become available [3]. This gives a better overview of data flow and processing.

These are only some examples, and new paradigms emerge and change all the time, so the exact line between them can be blurry. Furthermore, with multi-paradigm languages, multi-language projects, etc., a single software project increasingly relies on multiple programming and development paradigms. Still, most software development tools, computational notebooks included, rely primarily or exclusively on textual programming. While textual programming has proven successful, these alternate paradigms each have their own advantages. *Thus, we explore how the Jupyter infrastructure can be extended and how it can support different programming paradigms.*

For now, we focus on visual programming, which, while not broadly adopted, has proven successful in some niche areas of professional software development, e.g., embedded programming [15, 245] or high-performance computing [90, 264].

Since these different programming paradigms all have advantages and disadvantages, we also consider how they can be combined. Existing approaches typically only add different paradigms on top of textual programming [83, 211, 253]. However, these focus on using the visual presentation as an alternative, not necessarily in parallel, forcing developers to choose or incurring certain switching overhead and mental load. Lately, game engines (Unity[1] or the Unreal[2]) have added functionality, where developers can choose to write their applications using a text-based paradigm like C# or use a visual programming approach. However, the transition between the two is far from seamless. However, there is already evidence that when it is easy to switch between multiple programming paradigms, they can outperform single-paradigm editors in some areas, e.g., in education [318].

3.3.1.3 Developing Data-Driven Applications

Given that computational notebooks are particularly popular in the domain of data science and for developing data-driven applications, our work also operates within this domain. These types of applications have grown in popularity over the past few decades for many reasons, including increases in computing power and data availability. The many developers now working to create these applications require adequate support. While Jupyter notebooks are quite popular in this domain [242], there are other tools, some of which also explore alternate programming paradigms.

One example, using the graphical programming paradigm, is RapidMiner [28, 190], which provides a process workspace in which the user can compose operating trees and, thereby, the data- and control flow. Data processing steps are displayed as blocks and can be spatially arranged[190]. In contrast to textual programming, this visual presentation provides a better overview of workflows and accelerates redundant steps, e.g., data preprocessing [28]. KNIME [26] and Orange [69] offer similar functionality, including “visual brushing”, i.e., a user can select data and computation representations across different views. Of these views, some allow for textual programming, but this is distinctly separated from the visual presentation and requires a context switch.

[1] https://unity.com/features/unity-visual-scripting

[2] https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/

Weber and Hußmann [C4] provide an overview of tools for data-driven software development from the last decade. Their review shows that there are quite some tools that do not follow the common, text-based paradigm. Based on their literature review, they also provide areas that could improve data-driven development, including visual programming. However, as mentioned above, with the popularity of computational notebooks, textual programming is currently the paradigm of choice for data-driven development. Still, other development paradigms have other advantages, so the authors suggest exploring how we can increase the likelihood of developers trying and using them. *Thus, this paper explores whether we can achieve this by adding different paradigms to the established Jupyter infrastructure. Additionally, we also test whether combining different paradigms side-by-side can further assist with this endeavor.*

3.3.2 Extending the Jupyter Infrastructure

As previously described, the computational notebooks of Project Jupyter are essentially only a frontend, which interacts with a backend of kernels. These kernels execute the actual code, maintain state and scope, and respond with its output. The communication between these components happens via the ZeroMQ message passing system or HTTP on top of it. For a full overview of the Juypter infrastructure and communication protocol, please refer to the documentation[1].

The user interface for Jupyter notebooks can be extended with widgets. Such a widget can extend the functionality of an existing code or output cell or add a new type of cell with novel functionality. However, Jupyter widgets generally adhere to the sequential cell structure of classic computational notebooks. After overcoming some of the shortcomings of this structure, we decided first to build the user interface independent of this structure. Thus, our interface offers essentially a full alternative frontend, which interacts with the Jupyter kernels in the backend for code execution. In later iterations – for the multi-paradigm presentation – we re-introduced the cell structure as a parallel interface option.

We implemented this interface as a (VSC) extension. We chose this method for multiple reasons: Jupyter notebooks are also available for VSC, which gives us a

[1] https://jupyter-client.readthedocs.io/en/stable/messaging.html

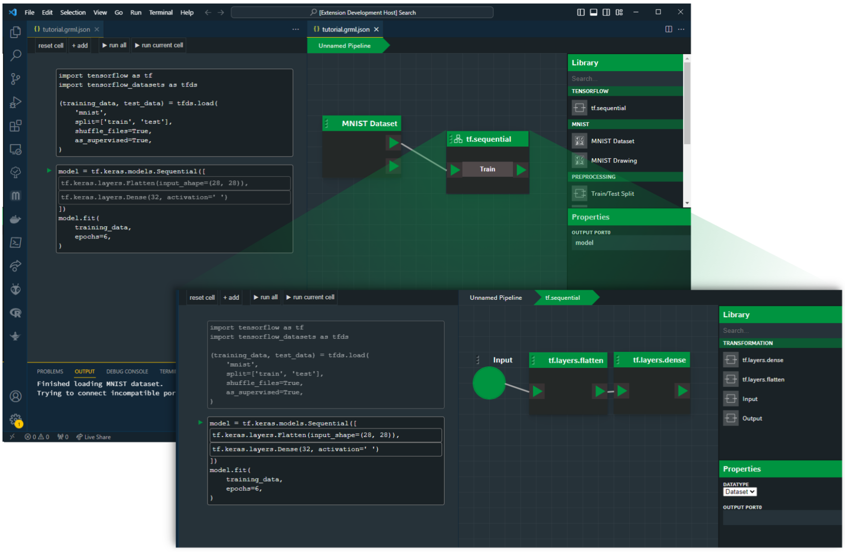
state-of-the-art reference in an environment that is familiar to many software developers. Additionally, VSC provides some functionality, both for the interface, like highly configurable side-by-side views, and for the underlying implementation, like easy file access. Thus, we could simply develop two custom editors in VSC, utilizing its existing functionality, and display them within VSCs interface. This kept the overall structure and appearance very similar and comparable for subsequent evaluations.

Within this context, we implemented a graphical programming environment and added an extended version of the common computational notebooks. This notebook operated in parallel with the graphical interface while keeping the code synchronized between both editors.

3.3.2.1 A Graphical Programming Interface for Jupyter

Our implementation of the graphical programming paradigm roughly follows common patterns found in existing tools for data-driven applications like RapidMiner [28, 190] but also other domains, e.g. programming of embedded systems [288], education [317], robotics [345], etc.

In this system, small system fragments are represented as blocks with inputs and outputs, which can be composed on a single canvas, which represents the software. Each block can be viewed as a function with a fixed number of parameters and a number of return values. From the data-processing perspective, each block is also a step in the data-processing pipeline. Inputs and Outputs can be connected via drag-and-drop (see Figure 3.7), where each output can be used in any number of subsequent inputs, but any input draws its' data from only a single source. Visually, this is indicated by connecting lines. A larger piece of software, thus, is a directed graph where the nodes are the blocks. Since the software of a certain size, and thus a considerable number of blocks, can quickly become visually cluttered, we added the option of composite blocks, i.e., blocks that encapsulate a sub-graph of blocks (see Figure 3.6). In the domain of data-driven applications, this is also particularly useful to represent neural networks, which often are considered a single unit on a higher level but have an internal structure, which can also be represented using blocks for each layer.



```
import tensorflow as tf
import tensorflow_datasets as tfds

(training_data, test_data) = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
)
```

```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(32, activation=' ')
])
model.fit(
    training_data,
    epochs=6,
)
```

```
Finished loading MNIST dataset.
Trying to connect incompatible por
```

```
import tensorflow as tf
import tensorflow_datasets as tfds

(training_data, test_data) = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
)
```

```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(32, activation=' ')
])
model.fit(
    training_data,
    epochs=6,
)
```

Figure 3.6: Certain blocks can be composed of additional sub-blocks, e.g., a neural network can be composed of multiple blocks representing its layers. Double-clicking a block allows the user to navigate through the hierarchy of composite blocks. In the textual view, the sub-blocks are equivalent to groups of lines of code within a cell. The code cell corresponding to the currently active block is highlighted.

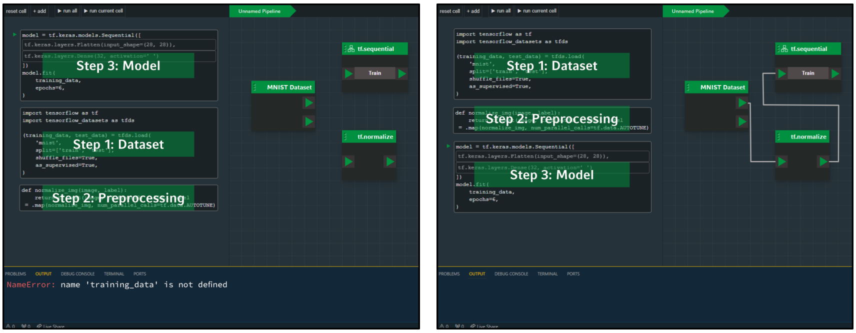
New blocks can be added to the main canvas from a "library" panel, which lists pre-defined blocks. While a block can essentially be an arbitrary piece of code, we focused on data science applications for now as it is one of the most areas where computational notebooks are used. Thus, we implemented a set of essential functionalities from the TensorFlow library [1] and provided them as library blocks. Blocks can also have additional parameters, e.g., type information about their inputs and outputs, which can be set in a dedicated panel. While output in Jupyter Notebooks is typically displayed below a cell, we chose to use the built-in output panel for feedback. This was done mostly because displaying any output next to the blocks would quickly lead to the canvas being cluttered and outputs obscuring blocks or vice versa.

In principle, it would be possible to implement this interface, including execution of the TensorFlow code, as a self-contained extension, using, for example, TensorfFlowJS.

However, some parts of the code execution, like training of neural networks, particularly benefit from running in a more optimized environment. Using the Jupyter infrastructure presents an easy way to push complex computations to a kernel running in the background. Additionally, it also allows for greater flexibility since the kernels can be easily exchanged with little overhead.

To interface with the Jupyter system, the VSC extension started a Jupyter kernel for the underlying programming language, Python, in our case. Aside from the visual presentation, each block is equivalent to a few lines of code representing its functionality. This aligns it with the Jupyter paradigm since each block now corresponds to a single cell in computational notebooks and can be sent to the kernel to be executed. This also means that if, in the future, we want to change the execution environment, we need only replace the code snippets and the kernel. In the graphical programming interface, this code was fully internal and not displayed to the user. Since in Jupyter, data exchange between different cells is done via globally scoped variables, each output port of a block corresponds to a variable for the return value. Whenever blocks are connected, the underlying code is automatically updated so that the correct variable names for inputs and outputs match. This also prevents issues due to variable or variable name re-use in the purely graphical environment.

Serializing Computational Notebooks When executing code in a computational notebook environment, one known issue [116, 161] is that since code is organized in independent cells, it can be executed in arbitrary sequence. This can cause errors when the developers do not take care of variable declaration, assignment, and usage, which happen in the correct order. By arranging the blocks spatially and enforcing explicit dependency between them through the connections, we prevent this issue, since the directed graph of blocks has a definitive order (see Figure 3.7) Of course, this is only the case when there are no circular connections, which we currently do not allow. When blocks are connected in sequence, it is straightforward to determine which blocks need to be executed first by traversing the graph to a block without inputs, typically a data source. This is what happens whenever a user chooses to execute a



```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(32, activation=' ')
])
model.fit(
    training_data,
    epochs=6,
)
```

```
import tensorflow as tf
import tensorflow_datasets as tfds

(training_data, test_data) = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
)
```

```
def normalize_img(image, label):
    return [illegible]
 = .map(normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
```

```
NameError: name 'training_data' is not defined
```

Figure 3.7: When adding new cells/blocks, they will appear in the order in which they were created (left) and can be executed in arbitrary order. Particularly in larger projects without rigorous structuring, this can lead to errors during code execution, for example, because variables are set in later cells but used before. Connecting them explicitly (right) defines the correct execution order, preventing these errors.

block: the graph is traversed, and all blocks that need to be run are added to a queue and then executed in sequence by sending them to the kernel, waiting for completion, and only then sending the next block.

Since ML systems can have computationally intensive steps, particularly the training step, only code should be executed that has been changed or is affected by changes before it in the processing pipeline. Thus, whenever a block in the execution queue is to be sent to the kernel, we first verify whether it or any block upon which it depends has been changed since its last execution. If there is no change, no re-computation is necessary, and we use the values stored unchanged in the output variables. Otherwise, the previous steps in the execution graph are executed as necessary, and then the code of the block is sent to the kernel, and its results are stored. In theory, if the code is known to be deterministic, it would be possible to verify whether a block's output has actually changed; if not, subsequent blocks would not need to re-compute. However, comparing this can potentially be time intensive, e.g., determining whether a trained neural network has changed. Thus, we skip this for now and assume that re-computation is always necessary if the previous steps were also newly calculated. This means that typically, we can only truncate the head of the execution queue, i.e., early steps in the data processing pipelines. Due to the additional information about the execution order, more elaborate optimizations or parallel processing are viable for future implementations.

Interaction with Jupyter Kernels Whenever it is decided that a piece of code needs to be evaluated, it is packaged with some additional meta information and sent from the UI to the VSC extension and then to the Jupyter kernel. Communication between the UI and VSC happens via internal message passing. The VSC extension uses a TCP socket for communication with the Jupyter kernel, which runs in the background. The meta information includes, for example, identifiers to associate a code execution with an environment so that previously computed values are available (we refer to the Jupyter documentation for a detailed listing of the communication protocol). Once the computation is complete, the result is sent back via the same channels and shows up in the output pane in the UI. Subsequent computations of the same block are thus cached and only require re-computation if anything before it in the pipeline has changed.

3.3.2.2 Multi-Paradigm Editor

Building upon the graphical programming environment, we combined it with the conventional Jupyter Notebook interface with individual cells in a multi-paradigm editor. Unlike some previously available programming tools for data-driven applications, which offer interfaces following multiple development paradigms, our version did not require users to switch between them explicitly. Instead, the two interfaces are displayed side-by-side with the same content but different representations (see Figure 3.6).

Furthermore, any editor changes automatically result in an update of the other editor to keep both representations synchronized. This means the previous internal code representation of a block in the graphical programming editor is made publicly accessible and editable for the users. In the future, it may be an option to make this code accessible also in the graphical view for increased flexibility for the user, e.g., via a zoomable interface similar to the ideas by DeLine and Rowan [67] where the code is the lowest zoom level as it is the lowest level of abstraction. An alternative view is that the user in this system can spatially arrange the cells of a computational notebook, thus denoting their order and the dependence of their inputs and outputs.

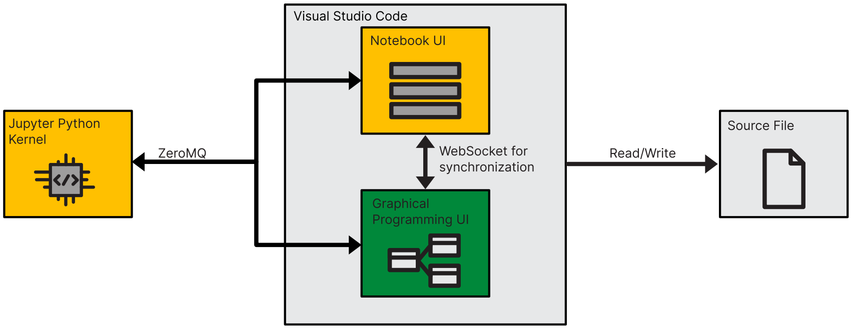


Figure 3.8: Our multi-paradigm setup in VSC uses two different user interfaces to display the same underlying source file. These two interfaces with different programming paradigms used WebSockets to synchronize the user interface. For code execution, the code was sent to a shared Jupyter kernel running in the background.

We facilitate the synchronization using message passing with WebSockets between the different editors: any interaction that results in a change of the program state, e.g., editing the code, triggers a message sent to the other opened editor instances, and they update their UI accordingly. The interaction with the Jupyter kernels is unaffected by this: whenever the user wants to execute code — from either interface — a request with the code is sent to the Jupyter kernel in the background, and any output is displayed in the output panel. Unlike in a conventional notebook, we chose a shared output panel for both editors. Not only was this the way that users in the evaluation said to prefer, but it also avoided mental load or confusion due to redundant information. Additionally, both instances of the editor operate on the same underlying file and are only two different user interfaces for displaying it. While the file could, in theory, also be used as a channel for synchronization, this proved challenging in practice since the editors should remain synchronized even with only temporary changes that the user did not save to the file. See Figure 3.8 for an overview of the different channels through which the system components interacted.

Whenever synchronization between the two editors is impossible, e.g., because the user edited the code but left it syntactically incorrect, we display a warning informing the user of the issue. For this, we highlight the erroneous code cell and display the following icon ❗ next to it. The user can hover over the icon to get a tooltip about the issue. Aside from this and the communication via WebSockets, both editors needed to be fully encapsulated due to the technical limitations of the VSC platform. Thus, it was impossible for us to, for example, allow dragging blocks from the block library in the graphical editor into a cell in the computational notebook.

3.3.3 Evaluation

A series of user studies accompanied the development of this prototype to assess how the alternative Jupyter interfaces and the different paradigms were received. As we use these studies to track progress and changes over multiple iterations, they generally follow a similar study design with minor modifications to accommodate new or changed features. Our evaluation focuses on data science tasks commonly performed in computational notebooks.

First, we conducted a user study (N=20) with only the graphical programming interface. Based on the feedback, we extended our interface with the option to display different interfaces side-by-side, which we further evaluated (N=12) over two months during development to determine the benefits of the multi-paradigm interface and to evaluate additional features.

Our first exploration of how to extend the Jupyter infrastructure was focused on building a graphical programming editor. We evaluated this editor against the baseline of the conventional computational Jupyter notebook. The results of this study informed the design of the multi-paradigm development environment and acted as a baseline for comparing multi- and single-paradigm tools. We designed all studies to follow the same procedure, utilizing the same instructions and tasks, and having a shared set of evaluation metrics. Given the creative nature of programming, it is unlikely that any programming task during the study can be different enough to eliminate learning effects but similar enough to allow for a meaningful comparison, so we instead opted for a between-subjects design with two distinct groups in the first study with the graphical environment and additional groups for each subsequent evaluation of further iterations.

3.3.3.1 Task

We designed the task for each of these studies to follow a typical machine-learning tutorial's general outline and volume. As previously mentioned, very large models can be challenging to handle properly with graphical programming but are also not the common use case for the aforementioned target group. Using a fairly constrained problem with a limited data set and a small model is a more likely scenario. Admittedly, in a real-world scenario, the task typically would be more open-ended, though. The large potential for creative or unexpected solutions to programming tasks makes this undesirable for a study setting, though, as a more open-ended task would make any meaningful comparison impossible. In detail, we took the existing MNIST example from the TensorFlow documentation[1] and adapted it where necessary for our study. The textual programming version used the Python programming language. For the other conditions, the most notable change was that when participants used our tool, we replaced all code examples with screenshots of our tool in the respective state. We had to rephrase some sentences to reflect the difference in interaction, which meant, for example, replacing occurrences of "to type" or "to write" with more appropriate verbs. Because we did not want to bias the participants, we were careful to phrase the instructions in a way that left it up to the user to decide which editor to use when multiple were available. We kept the visual presentation of the tutorial as close to the original as possible. However, we removed any clutter and unnecessary links from the page to ensure that participants focused on the core instructions. We also removed external links to keep participants on the tutorial page.

3.3.3.2 Procedure

After the participants arrived, we gave them a short introduction and asked for their consent to record the data. We began the studies with a brief introduction to the tools the participants were using. We then provided information on how to write and run code and how to add new cells to the notebook. For the graphical interface, we introduced the UI panels and their functionalities.

[1] https://www.tensorflow.org/datasets/keras_example

Participants were then given access to the instructions and asked to work through them at their own pace using the think-aloud protocol. At the end of the programming instructions, we asked participants to complete a survey. For additional qualitative feedback, we conducted semi-structured interviews. We planned the studies to take approximately 45 minutes, but we did not rush them to complete the programming task or interrupt them after 45 minutes.

3.3.3.3 Survey and Interview Guidelines

We divided the survey into three main parts. The first part included the consent form and privacy information, followed by questions about demographics, background knowledge, and expertise. The second part of the survey asked for feedback on the participants' programming tasks and the tool. We focused on workload to determine whether the additional complexity in the interface of graphical programming and multi-paradigm editors incurs a penalty to mental load. To this end, we used the raw NASA-TLX [112], System Usability Scale (SUS) [30], and Technology Acceptance Model (TAM) [62] questionnaires for additional feedback on the user experience.

In the third part, we asked specifically about different features and the hybrid aspect of the multi-paradigm tool using 5-point Likert scales and open text fields. In the first study, which only used single-paradigm tools, the third part instead compared the tool participants used with the one they did not use. To do this, we showed participants a screen recording of a user performing the tutorial task with the alternative tool.

The subsequent interview focused on qualitative feedback about the various features available in the current iteration of the development environment. After some general introductory questions about the tool, we asked participants if they needed additional features or if they would change existing features. When discussing individual features, we used the interview to elicit from users how and when a hybrid editor would be useful and whether it would encourage multi-paradigm software development.

3.3.3.4 Apparatus

Participants completed the survey and the programming task on a computer we provided using a mouse and keyboard. This allowed us to ensure that the training phase of the machine learning system used the same hardware, making it more consistent. We pre-installed all the necessary tools on the computer. The instructions page was displayed on a second screen to minimize the need to switch windows.

3.3.3.5 Participants

For the evaluation, we recruited participants with prior programming experience, including in Python, by contacting personal and professional contacts involved in professional software development. We focused on employees, both junior and senior, in medium enterprises where data science and ML start to play an increasing role, as well as students and academics who recently started to use ML in their own personal and professional projects. This target group will need to rely on data science functionality in the future. Still, it may not have the capacity to fully immerse themselves into the field, thus benefiting from easier-to-use development tools. Given the rapidly evolving nature of the field, we did not require expertise in specific fields or with certain tools. We had 20 participants in the first study, followed by five and seven participants in the two subsequent studies with the multi-paradigm tools, respectively. Nine participants identified as female and 23 as male, with a mean age of 28.91 years ($SD = 8.75$). Of these 32 participants, 18 reported working in an IT or computer science position, while the remaining participants described their occupation to involve programming for data-processing tasks, mostly in STEM (science, technology, engineering, math) research (9 participants) or other activities. Participants reported an average of eight years of programming experience ($M = 8.1$, $SD = 8.9$) and, not surprisingly, less experience in data science and data-driven development ($M = 3.4$, $SD = 6.7$). All but one participant had previously used Jupyter notebooks, and only four participants reported occasionally using graphical programming, although not for data science tasks. Since our task is based on a public tutorial, we also inquired whether participants were familiar with it. While all participants had at least heard of the MNIST dataset, none reported having previously worked on this tutorial.

3.3.4 Results

The following summarizes the findings of two user studies. First, the comparison of using the graphical paradigm versus standard computational notebooks, and second, the evaluations of the multi-paradigm interface.

3.3.4.1 Graphical Programming and Computational Notebooks

Of the 20 participants, we excluded two (one per group) from the study as their responses strongly suggested that they misunderstood parts of the instructions. The remaining participants provided feedback for purely graphical programming and the standard Jupyter environment.

Quantitative Results Each tool for the two paradigms scored reasonably well on the Systems Usability Scale [30], 81 of 100 ($SD = 11.7$) for the graphical tool and 74 ($SD = 10.4$) for Jupyter notebooks. The workload, as measured by the raw NASA TLX [112], is significantly lower for the graphical tool (Shapiro-Wilk test: $W = 0.934$, $p = 0.287$, t-test: $t(17) = -4.027$, $p < 0.001$). Moreover, each of the six subscales of the NASA TLX is lower for the graphical tool individually, highlighting the reduction in workload even further. Additionally, we compared the recorded task completion time for all participants, where completing the task using the conventional Jupyter notebook takes significantly longer on average than when using the graphical programming frontend (Mann-Whitney test, $p = 0.040$).

Qualitative Results However, participants in both the questionnaire and the post-study interviews stated that they still prefer the conventional test-based notebook, even though visual programming may have advantages. The participants even pointed out additional advantages of the graphical environment, such as a higher level of error prevention. Yet, they gave various reasons for their preference for text, the most common (six participants) being the higher level of flexibility. Four participants also explicitly considered graphical programming a more suitable tool for beginners. In contrast, they stated that textual programming requires some learning but should be superior in the hands of an expert.

At the same time, the participants viewed the ability to assemble the software from pre-existing building blocks using the graphical programming interface as positive, as it reduced the need to write boilerplate code and recall API functionality from memory. This also applies to using parameters, of which ML libraries can have many. According to the participants, presenting the existing parameters with their possible values helped to minimize the mental load. Seven participants mentioned using drag and drop to create and arrange the sequence of operations positively.

Based on this feedback, we asked in the post-study interviews whether participants would use a graphical tool for productive use. While participants found some of the aspects mentioned potentially helpful, e.g., for setting up a general structure, none considered a graphical programming tool a viable option as a primary development tool. However, two participants suggested combining different types of presentation. The other participants at least expected to use visual programming more often if it were a readily available alternative in their familiar work environment. Overall, participants emphasized their desire for flexibility in their work, which text editors covered best.

To address the issue of easy availability and flexibility, we created our multi-paradigm editor (as shown in Figure 3.6) as it would give users additional, easily accessible options. We evaluated whether the expectations voiced by the participants matched the actual user feedback and whether the additional user interface complexity would be detrimental to the workload.

3.3.4.2 Evaluation of the Multi-Paradigm Editors

First Iteration In this earlier evaluation of the interface that allowed users to use both graphical programming and the conventional Jupyter notebook, the notebook editor was read-only to avoid synchronization issues between the two interfaces, i.e., participants performed the task in the visual programming environment, as they did in the study with only the graphical UI, but had an automatically updated code representation side-by-side with the visual presentation.

Quantitative Results All five participants completed the study tasks in 52 minutes on average. Overall, participants viewed the tool positively in both quantitative and qualitative feedback. The responses to the TLX questionnaire show a slightly raised

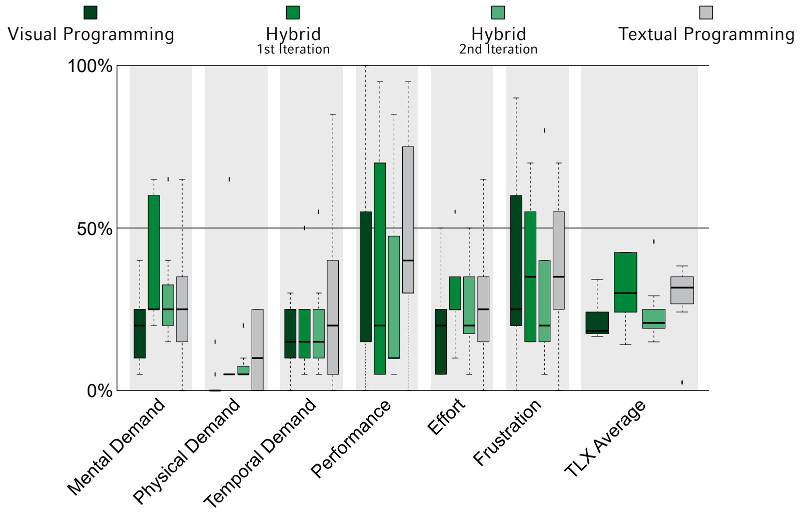


Figure 3.9: The raw NASA-TLX results for visual and textual editor in the first study as well as both iterations of the hybrid development environment.

workload but not significantly higher than the baseline from the first study, at an average of 35.1% workload (see Figure 3.9). The strongest deviation comes from the "Mental Demand" category, while the other categories fall somewhere between the results for pure textual and pure visual programming. The multi-paradigm editor received a SUS score of 81%, very close to the 79% of the graphical tool from the first study, but only 58.2% for *Usefulness* in the TAM and 70.5% for *Ease of use*.

Qualitative Results Additional feedback from the survey and the interviews was overall positive, with a wide range of additional feature requests and general usability feedback. According to them, having parallel presentations helped the participants to understand some of the steps in more detail than with just the high-level block representation. At the same time, the graphical presentation gives a quick and easy overview of the different steps. Participants commended the fact that the graphical editor forces developers to define the sequence of operations, thus preventing the known issue of computational notebooks where blocks can be executed in arbitrary order. However, our system was, at this point, only able to translate a single pipeline into a corresponding code representation. One participant noted that it might be valuable to allow for multiple pipelines in parallel, e.g., “one for playing around, one for actual use” (P2.1, translated). The fact that existing functionality is presented as choices in a library panel also led the participants to comment that this reduces the need to look up the same information online. However, they would still rely on external resources in some cases, e.g., in our study when a participant expected parameters for a block but found none. Additionally, participants wished for the option to extend the library with custom blocks that they would implement using the textual editor. The most common request, at this stage, was to allow for full duplex editing, though.

Second Iteration For the second study, we extended the editor to allow for duplex editing, i.e., reflecting changes in one editor in the other.

Quantitative Results This additional functionality does not affect task completion time, with 51 minutes on average. When comparing the workload responses, this added functionality has a positive impact, putting this iteration of the hybrid environment ahead of the textual paradigm and on par with the pure graphical programming environment, see Figure 3.9. This is a consequence of the fact that the previous increase in Mental Demand appears to be tempered in this iteration and the Frustration category shows slightly lower values. The SUS score remains largely unaffected at 82%. For the TAM, the *Usefulness* too remains at a similar level of 56.9% on average, but the *Ease of Use* increased to 82.7%, although not significantly (Shapiro-Wilk test: $p = 0.99$, Mann-Whitney-U test, $p = 0.180$). For the TAM responses, we need to note that participants were aware and reported after the study that a more mature implementation would likely receive higher scores.

Qualitative Results In the interviews, participants mentioned many general feature requests or usability improvements, emphasizing the early, prototypical nature of the implementation. Of the implemented functionality, all but one enjoyed the addition of the graphical editor in parallel with the notebook-like editor they were already familiar with. While the instructions were left open, which editor the participants used, they typically chose the graphical environment most of the time. However, they also noted in subsequent feedback that the scope of the task allowed this since the primary goal was to build a working system simply. They said that the graphical programming environment allowed them to do this fairly quickly. For a task that required a lot of fine-tuning, they expected to use the graphical programming paradigm for the initial setup and switch to textual programming for detailed adjustments. Similarly, debugging was another activity that participants expected to be easier in the textual editors, since it displayed more details. One example of this in the study was when four participants wanted to change the number of training epochs; however, the visual editor did not implement this parameter, so they quickly switched to the textual editor. On the other hand, each participant commented positively that a non-textual presentation allows interactions that are not possible in text. For example, for the MNIST task in the study, a drawing widget in the visual or hybrid environment allows users to test the predictive capabilities of the model they have trained directly.

Three out of seven participants highlighted the time-saving potential of multi-paradigm editors. Especially during the initial setup, it helps to avoid writing boilerplate code but then allows for an easy transition to fine-tuning in code. Two of the participants also see applications in education. They suggest learning to work with ML with visual programming first, and then moving to code after gaining some experience. One participant (P8) even went so far as to say that he would use graphical programming exclusively to develop data-driven applications because they felt it could help with the complexity of these systems. On the contrary, another participant (P9) felt that adding the graphical programming paradigm was unnecessary for experienced programmers and would feel like an “unnecessary burden.” Discussion adoption, participants were also aware that current graphical programming tools have little adoption but saw the potential of multi-paradigm editors since they make it fairly easy to switch on demand, which might encourage developers to use alternative paradigms more frequently.

3.3.5 Discussion

From this feedback, we concluded that the overall attitude toward a multi-paradigm environment is positive. However, the divergence of extreme opinions also shows that no tool will satisfy all developers. Thus, we argue that providing the infrastructure for developers to switch if they want to try alternative paradigms easily is worthwhile as long as the added overhead is unobtrusive.

3.3.5.1 Challenges in Bringing Multi-Paradigm Editors to Jupyter

Building on the existing Jupyter infrastructure greatly simplified this process. The Jupyter backend with exchangeable kernels essentially meant that we only had to provide a frontend that sends code snippets to the kernels and displays the output. However, different programming paradigms have different requirements and challenges, some of which are not trivially mapped to Jupyter. In the case of our graphical programming environment, early on, we decided that each block should be roughly equivalent to a cell in the computational notebook. This also simplified the mapping between the different paradigms down the line. However, the normal text-based programming mode shifts responsibility to the user: for example, as was also highlighted in prior work, cells can theoretically be executed in an arbitrary order, and it is up to the user to ensure this order works. The graphical programming paradigm with blocks has an inherent order, namely the sequence in which blocks are connected. In theory, we are thus able to determine which code needs to be executed first and can do so if the user has not run it. However, it is a fairly strong assumption that previous code blocks must be executed every time. Consider, for example, the case where the first block initializes some data, and the second block applies a transformation. Executing the transformation multiple times in the conventional notebook means that it is applied multiple times. When the graphical programming environment enforces the execution of every block in a sequence, we will execute the first block each time as well, which would result in a re-initialization and the transformation effectively being applied only once. Another edge case where this fails is with cyclic structures. Additionally, some computations may be time-intensive or expensive, considering the training of a neural network, so re-execution should also be kept to a minimum. Since the data flow is fairly well defined in the block-based interface, tracing input changes is one way to automate parts of this, but user input may remain necessary in some cases. Deciding when certain behaviors are desired will differ from situation to situation but especially from programming paradigm to programming paradigm, so the system's behavior needs to be communicated very clearly.

Just as some properties of the programming paradigms, like sequencing, are challenging to fit into the Jupyter environment, some properties of the Jupyter environment must also be considered when implementing different programming paradigms. One example in our case is the matter of naming and scope: since the Jupyter kernels essentially expect that code can be executed in arbitrary order, variables that are used across cells are typically globally scoped. This increases the potential for name clashes and similar undesirable effects. Meanwhile, the block-based graphical presentation implies strong encapsulation where information is exchanged only via clearly defined ports. In our implementation, we solved this by enforcing that each port has a unique name, which can then be used as a variable name without the risk of name clashes. Additionally, the code of each block can be automatically extended in the background to alleviate this, e.g., using closures to constrict its scope.

Finally, not every interaction is easily translatable between paradigms. In our study, participants had to train a handwriting classifier. In a more graphical presentation, adding a widget allowing users to test their classifier with their handwritten input is easy. This is not as simple in a purely textual presentation. However, Jupyter notebooks have the aforementioned flexible option to add widgets to cells and for user interaction. Aside from these more specialized cases, the Jupyter infrastructure, with a backend of kernels that simply require a message with the code that is to be executed, makes it straightforward to implement an interactive interface for executing code. Thus, we consider it a viable platform for future experiments on interface paradigms for programming.

3.3.5.2 Potential of Multi-Paradigm Editors

Looking at the participants' behavior with our prototype, they used the graphical programming style for most tasks, even in the multi-paradigm test condition where the familiar text-based UI was available. If this effect continues in the real world, multi-paradigm environments may achieve the goal of making alternative paradigms more accessible. This offers a positive perspective on RQ 4. Considering the potential benefits of other paradigms, e.g., the reduced workload we observed for graphical programming and could also achieve with our hybrid environment, there is great potential beyond our concrete implementation. The reduced level of frustration and perceived effort, which we observed in our study, may also help this cause. This is also consistent with participants' feedback that they enjoy being able to switch paradigms freely and their ideas about situations in which they would take advantage of graphical programming. However, there are additional effects, such as bias due to novelty and study context, which the participants also noted. Thus, we need to investigate further whether this behavior would lead developers to try and integrate alternative paradigms into their workflow or whether they would fall back on familiar patterns once the novelty wears off. Nevertheless, the feedback from all studies underscores the need for flexibility if we want developers to try out the alternative paradigms that have emerged from research and more niche domains. The scenarios participants suggested, such as graphical programming in early project phases to create high-level architecture and limit boilerplate and textual programming for fine-tuning, show that multi-paradigm development can be productive as long as the switching overhead is kept at a reasonable level. The differences in the mental demand during the different iterations also highlight that the concrete implementation, the number of features, and their usability can have a considerable impact. From a technical perspective, the extensibility of the Jupyter environment, coupled with its increasing popularity, makes it an excellent candidate to facilitate efforts towards multi-paradigm tools.

Besides the overhead of switching between two paradigms, side-by-side in the UI adds visual stimuli and functionality options that need to be processed by the user. Additionally, if models become increasingly complex, graphical presentation may reach its limits. Thus, graphical or even hybrid presentations may not be the first choice for teams of data science professionals. However, for smaller ML systems or projects where existing models are reused, a use case more common for smaller, less experienced teams, the fact that the first iteration of our hybrid environment did not add a significant workload, and the second iteration actually improved upon it, this is a positive sign. It suggests that hybrid environments are feasible without adding too much overhead. Of course, our prototype uses only two paradigms in parallel. Combining even more paradigms may change this situation, so the question of adequately representing three or more paradigms remains open. Since Jupyter notebooks can also have widgets to replace individual cells, it may also be an option not to replace the full UI and have different paradigms integrated at a small scale. It remains to be seen whether users can deal with this or whether separating different paradigms into clearly delineated editors is necessary.

Using multiple paradigms also presents additional challenges to ensure a good user experience. Since the paradigms may have different approaches to different aspects of programming, mechanisms must be in place to allow the paradigms to interact transparently with the user. One example we encountered in our implementation is the naming of values and variables. Text-based programming languages have well-defined rules for namespaces and scope. For the type of programming we used, the naming of return values and parameters is not strictly necessary since data is transferred via defined connections. Combining these paradigms side by side requires a compromise between these approaches. In our case, we named the connections and used those names for the corresponding variables. We also asked participants about this issue and how to deal with these inconsistencies between paradigms. The consensus among participants for these scenarios was that the tool should attempt any straightforward translations. However, it should also maintain transparency and avoid complex and far-reaching renaming or restructuring schemes, as they will likely confuse users. In cases where direct translation between paradigms is impossible, simple warnings were the suggested response. Expecting the user to fix these cases manually adds work; however, it was considered an acceptable trade-off since any confusing changes would require effort to understand anyway. This was also the common opinion for challenges between multiple paradigms in general: it appears to be better to request explicit interaction from the user if it reduces the potential to be not transparent and confusing, even if that means a small overhead.

3.3.5.3 Limitations

Compared to many existing single-paradigm tools, our prototype still needs to be improved in functionality. Some feedback from participants and feature requests are a natural consequence of this. Additional development time and effort would alleviate this, but if the implementation becomes more elaborate, this may affect aspects like the perceived workload. Yet, the main activities during development were already covered, so we expect changes to the user experience to be limited. Additionally, in data-driven applications, a great deal of complexity comes from the domain and data, not the tool itself. While this means that the choice of tooling cannot mitigate all complexity, the fact that multi-paradigm editors offer different presentations can benefit different target groups. For example, domain experts, who deal with domain complexity, and software developers, who deal with system complexity, can use and communicate using the same tool but use the paradigm that suits them best in any given situation. Working in the same environment can help with communication and understanding. Deployment in group scenarios can provide these types of insights in the future.

With the complexity of the data and domain, the task we chose for the user studies for the reasons described above is comparatively small and constrained. While this allows for a better comparison of the different tool environments and has yielded valuable insights, how well the hybrid approach scales to real-world problems remains to be seen. Since the hybrid environment is an extension of existing and established tools, like Jupyter Notebooks, we expect it to perform well. As the study showed, the additional tool functionality is beneficial, and the added option to view systems from different perspectives with different degrees of abstraction should be beneficial, particularly at a larger scale.

Finally, our hybrid environment has been used only by a few people. We published the tool, thus opening it up to a wider audience. Further, this will help to add and improve functionalities through contributions from the community, which may turn it into a viable alternative for professional developers. It will also give us access to additional feedback, e.g., other usage scenarios, project or domain-specific requirements, etc.

3.4 Summary

What this chapter clearly shows it that, with the pervasiveness of AI, we are truely in a new era of software development. Not only does it enable new types of applications, but their creation fundamentally differs from how software has been created in the past.

While our initial study focused only on code reading, it is plausible to expect similar differences across other developer behavior as well. After all, the reading behavior can be an indicator for the underlying mental processes and how developers perceive the code can also influence how they then interact with it.

Developers may be able to cope with these differences and create data-driven systems without dedicated support, afterall they managed to do so in the past. However, this is also an opportunity to create more tailored support mechanisms that specifically address the needs of developers in a changed and continuously changing world. This matches the general trend in Software Engineering research to collect more empical evidence for various aspects of software development, including tooling as we found in the literature review but also beyond. However, what our results also emphasize is that any of these findings must be viewed in the context in which they were collected. Studies conducted for the devleopment of traditional software may not be equally applicable to data-driven development and vice versa.

The same appears to apply for specific tools as well. Thus, it is no surprise that with the proliferation of data-driven development, tools like computational notebooks have seen a surge in popularity. Some of it may be due to an increase of maturity of these tools, but it may also be an indicator that developers have identified the paradigmatic shift and the need to re-orientat their tool preferences to match this new reality.

At the same time, data-driven systems will likely continue to transition from being a novel and different type of system to being simply another option in a developer's toolbox. Thus, a clear separation of data-driven and traditional systems is likely not sustainable. This will also reflect in the tools, where separate tool eco systems for each development paradigm are not desirable. Afterall, one of the feedback we received was that developers do not like constantly switching betweeen different tools for different contexts. In addition, the new capabilities of data-driven systems, like LLM supported coding as we explored in the previous chapter, will only add new types and variations of tools making the tooling landscape even more diverse. Thus, integrating various different development and tooling paradigms while reducing the switching overhead, as we managed to do with our hybrid tooling approach, may be a necessary direction.

With this, the work described in this chapter provides some first user-centered evidence for both the challenges for data-driven development and for a potential direction how to address these challenges.

4 Discussion

The research described throughout this work very demonstrates that the impact of AI is not only felt by Software Engineers but is demonstrably quantifiable and has observable effects on their behavior and work practices. The following section will discuss the higher-level implications of these developments in the field.

4.1 Software Development with LLMs

In the near future, we will likely see the biggest impact on Software Engineering from the introduction of LLMs. The fact that developers can describe the problems on which they work in relatively abstract terms in natural language and request information about arbitrary existing code can save the developer a considerable amount of work. Even when using auto-completion for generating only small snippets, as we saw in our study, can offer relevant guidance for the developer on how the problem can be solved even if the solution may not be entirely accurate at the end and still needs correcting.

The fact alone that LLM-based support systems are using natural language as the primary level of abstraction will have some far-reaching consequences. For one, natural language is highly accessible for novices and end-user programmers. They are now able to generate up to complete applications from relatively simple descriptions. Similarly, this will impact the interaction between professional developers and nonexperts, e.g. customers. Requirements are already typically described in natural language, which the developers, up to now, had to translate into code. The fact that we can now directly prompt LLMs with any initial vague specification to generate a first prototype, can drastically increase the pace at which software can be iterated, particularly in early phases.

However, while this can enhance productivity, it is unlikely that these AI systems can completely replace a trained developer in the foreseeable future. One reason is that current AI systems are limited in how much context they can take into account. Additionally, current systems are probabilistic in nature and will produce the most likely code as learned from the training data. In consequence, in our study on programming with LLMs, it was clear that the resulting code quality is fairly average. One assumption might be that an improvement in the underlying models can alleviate this. However, at least the current trajectory of training with an ever-increasing amount of data does not support this assumption [128, 220, 294, 304]. Thus, new mechanisms will be necessary, e.g. to automatically steer the systems towards better code, or incorporate human feedback. Additionally, there will be circumstances where a system behaves atypically, e.g. because it is used in highly unusual circumstances or because it needs to accommodate niche und specialized legacy systems. In many of these situations, AI systems may be able to boost performance but will require a human to guide them in the right direction to fill the gap where the AI system struggles. We also see this in the fact that even large vendors for AI systems still build development tools for humans [27, 74, 246, 255]. Yet, even if the development process requires human input, existing approaches like code generation can be a first starting point. By working in this intersection of Software Engineering and Artificial Intelligence, we have many new opportunities to improve the work of developers, while taking a human-centered perspective will be crucial to ensure that these advancements also find practical application.

Code generation with GenAI can make software development more accessible to novices and give professional Software Engineers a means to quickly get started or prototype systems. However, it is likely not a replacement for systematic Software Engineering, particularly for large-scale and specialized systems, but another tool to support developers.

- GenAI programming tools should be built to enhance human abilities rather than as a replacement for them.
- GenAI coding tools should seamlessly integrate into existing workflows, e.g., so developers can easily take them as a starting point before iterating on the generated code. Similarly, it should be seamless to fall back to them when the problem is conceptually understood but needs to be put into a concrete implementation.
- We may require more mechanisms that ensure that generated code reaches a high level of quality.

4.2 Supplementing, Supplanting, or Shifting Software Engineering Skills

While AI-powered development systems like GitHub Copilot Spark [246, 255] are capable of generating even complete applications, they explicitly note that this is meant for "micro apps" [246]. Large-scale software systems will, at least for a while, still require human input. A technical reason for this is the context size of modern LLMs. More importantly, though, Software Engineering is typically not as straightforward as precisely and completely specifying the functionality and then translating these requirements into code. Instead, software development is also an exercise in human communication between the different stakeholders. Requirements can change during development due to technical limitations, a change in perspective, etc. Similarly, the high-level Software Engineering decisions are also typically done independent of the low-level implementation, so code generation systems can contribute little to them. Thus, with the productivity gains of AI, Software Engineers may simply see a shift on which activities they need to spend more time.

Additionally, the usage patterns in our study showed how developers frequently use LLMs, particularly the chat interface, as a replacement for an online search. Thus, some of the benefits of LLMs seem to manifest more as an incremental change in developer activities and less as a radical improvement. While modern AI systems can go beyond retrieving information, their probabilistic nature and risk of hallucinations also mean that their users need to evaluate the responses. Thus, AI support will often not eliminate human work but instead, shift what actions the human performs and then accelerate the overall process. In the case of using LLMs as a search tool, for example, the developer may reduce the workload for aggregating and selecting information but may have the added mental load of detecting and correcting hallucinations or refining the prompt to achieve the desired result. Similarly, when software systems become large and complex, AI systems can explain the code or justify the design but particularly for larger code snippets such "reasoning" becomes increasingly challenging. This means that developers must already have a rough understanding of the systems to assess any generated information and put it into context.

One way how developers can reduce mental load during development with LLM support can simply be that they rely excessively on the LLM and sweepingly accept generated code without thorough review. We saw this behavior from some participants in the study who consistently assumed the generated code to be correct at first. The pilot study in Section 2.2 also highlighted the issue with participants simply submitting the full specification, if available, to the LLM and taking the response as an adequate solution. Typically, developers will not use an LLM system for large-scale problems at first but start with small problems. For these, the LLM may provide suitable and impressive results. This bears the risk that developers expect similar results for larger, more complex problems and may overtrust the system in these cases where the performance may deteriorate. Additionally, some of these models have also seen general quality deterioration over time [45], particularly with an increase of AI-generated examples in their training data [95, 279]. Thus, developers still need to be wary of this and continually question the results. This can further diminish how much these systems can actually reduce mental load. On the other hand, if developers exhibit such an overreliance on LLMs, there is also a risk of becoming out of practice. Maintaining programming skills will be necessary though for instances when the automated systems fail or the output is not as intended.

The act of writing code is not simply a mechanical process of entering characters either. While writing the code, a developer may reflect on it, whether a chosen solution is adequate or how else to solve the problem at hand. This reflection process is, as our participants were aware, an important part of problem-solving and thus Software Engineering. If the writing of the code is outsourced to an LLM, it is unclear whether this reflection happens equally and at a different point, e.g. during the review of the LLMs code, or whether it supplanted by the LLM. Generally, this reflection is desirable as not only does it help find a good solution but it can also be a part of a developer's learning experience. If this reflection is diminished by using LLMs, it may reduce mental load but also the benefits for the software and its developer. If these mental processes happen at a different point this would be an indicator that code generation is not necessarily beneficial in terms of workload, as the mental load simply shifted. Whether the reflection during code reviewing offers the same learning effects as reflection during code writing is also a question that can only be answered by observing the effect of these systems over longer periods of time.

Overall, this suggests that AI will not replace developers but support and supplement them in some activities. Similarly, it may shift the relevance of other skills, e.g., away from encoding instructions in a programming language, towards higher-level activities like understanding requirements and their underlying problems, precisely describing the conceptual functionality of the software, and reviewing generated solutions. Right now, this roughly matches the common distinction between junior and senior developers: junior developers are typically the ones programming the concrete details, while senior developers often work at a higher level of abstraction. However, to reach this seniority, most senior developers have spent time as junior developers, gaining the necessary experience. If AI can replace the activities of junior developers, it also diminishes this phase in a developer's development and the growth and learning associated with it. Whether this means that the developers lose certain skills, as they are taken over by AI, or whether developers maintain a strong but changed skill set and just benefit from added automation and support, remains to be seen.

To determine when and where to productively use AI, it is important that all stakeholders in a software development project have realistic expectations. Knowing when AI can increase productivity, e.g. via code generation, and where human input is required, e.g. the high-level description of a system, allows developers to effectively apply their workload. The perception of these systems and the resulting trust in their capabilities will also require calibration. The rapid changes in the field with the resulting marketing promises as well as early impressive experiences with these systems may lead users to be overconfident. Likewise, if these expectations are not continuously met, it may result in users becoming disillusioned and their perception may shift towards the negative, i.e. the "trough of disillusionment" in the Gartner hype cycle [91]. It remains to be seen when and how the rapidly evolving field of AI achieves a state where the technology is widely adopted for broad practical benefits, i.e. when it will reach a stable "plateau of productivity" [91].

Reaching this will not only require developers and customers to have a realistic estimate of the capabilities of AI, but they will also need new skills. Prompting a generative system in such a way that the generated code matches the functional and non-functional requirements and integrates well with existing systems can be challenging, particularly with the growing system size. Even when iterating this with continuous feedback from the AI system, it will still require mental effort from the developer to select and present the relevant information. Having an understanding of aspects like system design, architecture, and integration, i.e. typical Software Engineering skills, will continue to be helpful, if not grow in importance for prompt engineering.

Beyond this, even if AI systems allow developers to successfully describe the functionality in natural language, this is merely translated and the actual system is typically implemented in an established programming language. Even when working at a high level of abstraction, developers will likely need to look into the actual implementation and thus continue to need general programming skills. Aside from general reviewing whether AI-generated code is suitable and implements the desired features, knowledge of the nuances of the implementation can be particularly necessary for non-functional requirements where reliance on AI-generated code and an AI-generated summary of it are insufficient, like safety, or security. For this, developers need to be able to read, understand, and review the code and address issues when they arise. This again means that the responsibilities of the developers are not replaced by AI systems but are shifted instead.

The proliferation of AI-powered tools does not replace software developers but shifts what activities they perform. This requires developers to acquire new competencies. At the same time, a solid foundation in many of the traditional Software Engineering skills will be necessary to cope with the ever-increasing complexity.

- Developers will still need to learn fundamental Software Engineering skills, e.g., communicating requirements and understanding the problem space, if they wish to effectively leverage AI, e.g., for generating code.
- The design of code generation should still ensure that developers reflect on the code, e.g. to determine whether it is a suitable solution for a given problem and whether the generated code is of sufficient quality.
- We require increased awareness of the capabilities and limitations of AI tools among all stakeholders in the software development process.

This change in activities then needs to be reflected in Computer Science curricula as well: prompt engineering and reviewing generated code will be essential skills and need to be trained. This has the challenge that the field is still rapidly changing, so it is not clear how well skills translate to the next iteration of these systems. Thus, it will be a challenge to find suitable methods to teach them. Additionally, when adding topics to a curriculum, typically other topics have to go to make space. Computer Science educators thus will need to evaluate how and what they teach going forward. Given the potential productivity increase, using LLMs will be a fundamental skill for many professional developers. To accommodate it, educators may consider replacing some of the traditional Computer Science fundamentals in favor of how to use LLMs and the higher-level aspects of Software Engineering necessary for it. However, tweaking a curriculum at the fundamentals will have a ripple effect on everything that depends on them. Additionally, when the LLM system fails to generate the desired results and simply re-prompting does not help, these fundamentals will still be relevant. Likewise, for a sensible review of the LLM-generated code, a firm understanding of Software Engineering principles will most certainly also be valuable. When developers actively use AI support in their professional practice, and thus are less likely to pick up the high-level skills necessary for senior developers in their junior years, education will need to bridge this gap.

Besides this activity-centered perspective, using AI in development tools will also influence the created artifacts. Looking at the data collected during our evaluation, we observed that the quality of the created code was relatively unaffected by the use of AI systems. This is not too surprising though, considering how the underlying LLM works. Effectively all currently relevant LLMs for code generation are based on the Transformer architecture using the Attention mechanism [302]. Thus, they operate primarily to generate plausible syntax without an "understanding" of the semantics of the code of what would constitute code quality. Additionally, their statistical nature also means that they will create code that is shaped by training data. This training data will include both very good but also poorly written code. The code that an LLM generates will typically be between these extremes and likely of average quality. If developers continuously read this average code from LLMs and in the codebases they work on, this level of quality will likely become normalized, by habituation but also because there are fewer excellent or poor examples from which they can learn. A continuous increase in code quality certainly is desirable. This increases the need for measures to mitigate this and ensure a high level of quality. This can be a technical solution, e.g. code generating systems that actively work towards fixing faults in the code (e.g. [329]). Education can play an important role as well, with a greater emphasis on high-level aspects of Software Engineering. Additionally, by automating the lower-level activities like writing some of the code, creating test cases, etc., developers may theoretically also have more capacity to follow known but often neglected best practices like maintaining code quality or effectively communicating with other developers. Because the auto-completion interface relies on comments in the code, there is a risk, though, that developers assume that the mere presence of these comments is sufficient documentation. With this, comments in the code might deteriorate into machine instructions rather than human-readable documentation. Developers could rely on the explanatory capacity of

LLMs to generate documentation after completing the code. However, when reviewing generated code, they will need to understand whether the actual implementation not just the concept behind it is adequate. Thus, code comprehension of the generated code is crucial. If they do not want to blindly trust that generated documentation is correct, this requires reading the actual implementation. This again only shifts the workload from writing to reading. Additionally, this means that the generated code should not just fulfill technical quality criteria but also human-centered ones like readability. Alternatively, we need to enhance programming languages and coding practices to better support these cognitive processes.

> Given the potential productivity benefits, the ability to effectively use AI tools will be an important skill for professional software developers. Training these skills must become part of Computer Science education, but this should not come at the cost of fundamental Software Engineering skills. In fact, Software Engineering principles are increasingly relevant as code creation sees increased automation.
>
> - Computer Science education must give future developers the skills to effectively utilize LLMs and similar systems.
> - By automating low-level activities like typing out code with AI, there is potential to shift the focus more on higher-level activities that contribute to high-quality software.
> - Methods to support code comprehension and legibility will increase in relevance when developers spend more time reviewing generated code.

The impact on the activities of the developers, workload, problem-solving skills, communication with stakeholders, education, etc. once more demonstrates that the proliferation of AI in software development is not simply a technological improvement but has a broad impact on human-centered aspects.

4.3 Supporting Developers with AI Tools

AI has, of course, also given developers a myriad of additional tools beyond pure code generation. These tools can offer support with these new and changed responsibilities and requirements. Arguably, the examples from this thesis can only offer a small selected snapshot of how tools can be extended with AI. As mentioned, many other tools for various development tasks like bug detection, documentation, etc. exist each with their own added benefits. However, the two exemplary topics in this thesis already highlight interesting effects between them. For the adaptive IDE system, as described in Section 2.1, we received feedback that the system was viewed as positive and that participants saw the potential benefits of offering personalization and with it reducing the interaction cost of performing certain tasks. However, developers were apprehensive regarding whether they would actually use it in practice. In our studies, the most commonly cited reason for this was the desire for flexibility and control over the interface and an automated system was viewed to infringe on this desire. This was also feedback we received in other studies, e.g. when exploring graphical programming for data-driven development. Here, using alternative programming paradigms was also considered beneficial in certain circumstances but the traditional way of textual programming was considered to be the one that offered the most flexibility. On the other hand, developers were quick to adopt LLM-based code generation (Section 2.2). In our study on the topic, there were some individual concerns about a loss of control when such a system is used excessively, but overall it was viewed as positive and supportive. This may, in part, be because the LLM-based completions used a relatively sophisticated system with substantially more development work behind it than, for example, our own prototype of an adaptive IDE. However, other effects might be relevant as well.

One point mentioned by some participants was that the overhead of learning and adopting a new tool can simply be too much. In terms of interaction, the LLM-based code completion utilized mechanisms with which the developers were already quite familiar: either the auto-completion, which is merely an incremental improvement over existing completion systems, or the very common chat interface. The adaptive IDE, on the other hand, used a familiar interface but changed it. Considering how adamant developers can be about their tooling choices, it is not too surprising that changes to their tools may be met with some resistance at first. While there may be some resistance to changing development tools, some of the programming subdomains, like Web programming, have a large and active community that develops new libraries and frameworks at a rapid pace – and using these also requires a fair bit of initial learning. Similarly, many of the decades-old programming languages have a steady user base that prefers to continue using them but newer programming languages can sometimes quickly gather a community of eager early adopters. Thus there clearly is no unwillingness to learn among software developers, per se. At the same time as well, the chat-based LLM interface and to a degree also the auto-completion interface, while familiar, do require some new skills though, particularly a knowledge of how to best prompt the LLM for the desired output. Still, developers are more eager to adopt these systems and put in the work to acquire these skills. So this may play a role for some, but cannot be the sole reason for the lack of adoption.

> Software developers have a strong preference for flexibility to adjust their tools and remain in control of the automation of development activities. AI tools can offer both, automation and flexibility, but will need to balance these aspects.
>
> - New software development tools should leverage AI as a means of giving developers the desired degree of flexibility.
> - When aspects of the development process are automated, including with AI, the developers should have the option to maintain a high level of control over it.
> - Developers can have strong opinions when new or changed tools violate their expectations. When developing novel tools, it is therefore crucial to understand developer habits and expectations.

Looking at these interfaces from a point of view of interaction design, most of the systems we explored and many more from the literature promise to simplify interaction or provide shortcuts, e.g. by making features more accessible, as was the case in the adaptive IDE, or by aggregating lengthy interaction flows in semi-automatic tasks, e.g.

for automatic build processes, automated refactoring, etc. However, these simplified interactions provide typically only syntactical support, by which we mean that they help in the mechanical process of getting the abstract programming concepts from the developers' mind to the machine or reduce the actions necessary to achieve a goal. Meanwhile, they often offer little help with semantic tasks, i.e. the cognitive effort of understanding a Software Engineering problem and finding an adequate solution. Particularly when cutting out steps in the interest of efficiency, they may eliminate activities or hide information that might be beneficial for the developers' mental processes. For example, automated build processes can save a lot of interactions but when they fail, they can add additional overhead in identifying the fault. Considering the complexity of modern software systems, the internal, cognitive effort may far outweigh the effort necessary to translate it into code. Controlled studies often use only fairly constrained programming tasks. Thus, they do not suffer from this imbalance. For real-world use cases, however, their benefit for the mechanical processes may simply be negligible compared to the mental effort to cope with the complexity of real Software Engineering problems.

Of course, there are efforts to support these mental processes, e.g., research on program comprehension. However, existing support mechanisms see little adoption. On the other hand, widely adopted mechanisms that could reduce the cognitive effort, e.g. syntax highlighting to guide developers to relevant aspects of the code, seems to have no notable effects [107].

With the introduction of AI support to software development, there is a great chance to go beyond purely mechanical support and achieve such a reduction in mental effort. Generating code from the abstract, natural language description already tackles this to a degree. With this, the workload of worrying about specific implementation details or idiosyncrasies of programming languages is reduced. Unlike with existing systems where code is generated from, for example, formal specifications [14, 29, 145] and where the developer must adhere to their constraints, this also gives the developers the desired increase in flexibility. The ability to then prompt the AI system to provide additional details, rationale, and explanations, i.e. semantic context further allows developers to outsource this semantic effort, potentially reducing mental load. This may be another reason why new AI support tools are being adopted so rapidly and why they manage to provide tangible productivity benefits.

Another interesting aspect that may also contribute to how people perceive AI-powered systems and thus on adoption is the anthropomorphization of AI. In our study, some of the participants referred to the LLM as if it were another person. Attributing an intention to the system and assuming a rationale behind its "decisions" might be a factor that contributes to acceptance. Of course, this also has a risk of inducing overtrust. Further studying the effects of anthropomorphizing AI may not just yield explanations for why it is adopted but can also be a factor in improving the user experience.

> Most software development tools are improving or simplifying the mechanical process of getting abstract programming concepts from the developers' minds into a machine-processable format. However, much of the complexity of software development lies in related cognitive processes, e.g. understanding the core of a problem, abstracting from it, and exploring the solution space to determine a solution on a conceptual level. For these processes, there is still little support, but AI systems have the potential to address this gap.

4.4 Meeting the Demand for Data-Driven Software

In addition to the existing challenges, the ubiquity and increasing demand for data-driven applications will mean that more developers will have to build these systems. This means that they will have to engage with this kind of software, which, as our studies show, has some fundamental differences from traditional software. Additionally, the field is still undergoing rapid changes, which poses additional challenges to developers.

As this domain and the system it produces change continuously, our research on the differences between data-driven and traditional development makes it quite evident that it may not be adequate to simply continue using the same tools that have been used for decades for traditional development. While expertise with traditional Software Engineering skills may certainly be helpful, the activities, behavior, and challenges of data-driven development are in parts different. The question remains how the field will develop going forward and how it affects the importance of these development skills, and how to support them.

Looking at the historical context of computer science and similar fundamental upsets may yield some insights. Assuming AI can provide some of its promised benefits, its importance may be likened to developments like the internet. The advent and subsequent broad accessibility of the World Wide Web has fundamentally changed many industries, many of people's activities, professional and recreational, and also the work of software developers — something AI is already doing and may continue to do as well.

While the historical development certainly cannot be matched exactly, it is noteworthy that for both the internet and AI, the foundational technologies, like, for example, network protocols and neural networks, respectively, are ideas from early computer science. As they have evolved over time and found wider adoption, both have had an important impact on a myriad of fields across society. With this, they have been integrated into software from virtually every field. This meant that developers had to increasingly engage with these technologies over their evolution. At the same time, both technologies have also enabled new tools and support mechanisms that increase productivity, online platforms, and ways to share code via the internet and the tools described throughout this thesis respectively. However, developer-facing improvements are a necessary but not a sufficient reason for the success of both these technologies. Pushing them in end-user-facing systems, like easily accessible personal websites for the internet and systems like ChatGPT which people can access and try out seems to be what has demonstrated the mass appeal of the technologies.

With all these parallels, it will be interesting to see how the field continues to evolve. After all, after an initial phase of excitement for the internet, it resulted in the "dot-com bubble" of the year 2000. After an inflated interest in the technology, the market for internet companies crashed resulting in many of these companies going out of business. Given the massive level of investments in AI technology [181] and some fundamental challenges in their continued development, a similar scenario is a risk for AI as well. There are some differences between the situations though, for example, the fact that much of the investments in new and bigger AI models, e.g., the hardware and training, happens particularly in large businesses [181]. This strong consolidation of investment would likely also affect the spread and impact of any disruptive developments. Thus it remains to be seen how the situation develops.

However, after the dot-com bubble much of the development work for web-based systems did not vanish with the crash. In fact, web development work has grown into a mature industry in the last decades. Thus, independent of whether the market for AI continues on a steady trajectory or hits some bumps on the way, it is very likely that data-driven development skills will grow in importance going forward.

Following these mid- and long-term developments of the historical example, one might extrapolate that the field of AI will continue a decades-long development of becoming a field of software development with many practitioners at every level, from small businesses to large corporations. However, AI has some fundamental differences that need to be taken into account. For one, there are logistical and economic challenges, that ML training at the scale of many successful modern systems currently requires investment in computational resources and data acquisition. For small companies, this presents important hurdles. Additionally, the volume of data that supposedly is necessary for improving current ML models may exceed the amount of data available [304]. There are several potential directions for the field to mitigate this, all of which have implications for software developers. One option may be that the large ML models will remain in the hands of a handful of corporations and developers can merely access them via APIs. This may be the most restrictive approach, putting a lot of emphasis on API design to be useable for developers. However, the restrictive nature also stands in contrast to the desire for flexibility and control, which developers have voiced across our studies. Thus it is unlikely that this is the sole way of using ML going forward. The will likely always be a community of those that want to go beyond what corporations offer them, which likely will open this avenue to other developers as well via, for example, open-source software. Either way, developers will have to learn how to select models based on which is most suitable for a given task.

If developers use their own models, they will have to contend with their resource requirements. In this case, the management of these resources will be a development responsibility that gains in importance. Here, some of the support mechanisms from traditional Software Engineering may certainly come in handy, e.g. tools for automated provisioning of resources. However, the specific requirements of ML infrastructure also have potential improvements. Additionally, decreasing the resource requirements of ML systems is another way to mitigate this potential overhead.

Decreasing the computational overhead will also be an important factor for equitable access to this technology. Training large ML models and the associated costs are not feasible for everyone. However, there is already a possibility that the current approach of scaling models to achieve better results will hit a limit. In this case, a shift from purely data-driven systems to systems that utilize other mechanisms, like rule-based AI, or a combination may be necessary. Coming up with these rules and encoding them may become a skill set for developers, which is closer again to traditional software development. However, it would put also again more emphasis on the rules with which developers come up. This will likely be a mentally challenging task, so mitigating mental load will continue to be critical.

For this, just like the question of how to use AI, the challenges of how to build it are also relevant for Computer Science education as well. While not every Computer Science student will end up becoming a full-time data engineer, having at least some knowledge of the functionality and its strengths and weaknesses is essential to gauge when it may be an adequate solution. This should also include legal considerations, e.g., how to treat sensitive data that may be encoded in data-driven systems. Ethical considerations, e.g., the unforeseeable effects that these probabilistic systems can have, are another important part of this. However, changes in education face the previously mentioned challenges as well, e.g., that the field is still constantly evolving, which makes it hard to reliably determine what aspects are the most relevant.

In which of these directions the field ultimately moves remains to be seen. Regardless of this, understanding the human perspective of these developments certainly is essential to facilitate the success of any technological innovations. While it may be a straightforward approach to try to reuse existing tools, the demonstrated changes in the development paradigm invite innovation and creative solutions to better support developers.

Going by historical parallels, data-driven development and the creation of AI systems will very likely continue to impact society as a whole and developers specifically. Whether this continues at its current trajectory remains to be seen. Additionally, there remain several challenges to making AI a broadly adopted part of developers' toolkits.

- Following advancement for general data-driven development in recent years, there are many specialized use cases that still require specifically tailored solutions.
- Increasing the ease of development of powerful data-driven systems like LLMs and making them widely accessible is an important step to prevent the siloing of these currently computationally expensive and resource intensive systems and their capabilities.
- As we see data-driven systems become part of many software systems, undestanding their general operating principles will become a necessity for all developers. Computer Science education will need to accommodate this.

4.5 Tools for a Multi-Paradigm World

There is already a large body of research on Software Engineering methods and support mechanisms for traditional software development. With data-driven development as an increasingly important paradigm, we may need to re-evaluate whether this prior work is still applicable in this evolving context of data-driven development and beyond. Our hybrid approach of development tools (Section 3.3) is part of this effort, as it was initially motivated by the exploration of how graphical development tools perform for data-driven development. The feedback in our study included comments that the graphical nature is highly suitable for the pipeline-like structure of many ML systems as well as that communication about ML is often visual to begin with. This further highlights that the change in the development paradigm may result in a change of perspective on tools that might have not been successful in the past.

A multi-paradigm approach, like the one we presented, may also be resilient in an environment of changing development paradigms. If, for example, rule-based AI and corresponding tooling paradigms become popular, having the tooling environment for multi-paradigm UIs already might reduce both the implementation and adoption overhead. And even if AI continues on its current path of data-driven development, it is unlikely that it will be the single relevant paradigm. In fact, given the probabilistic nature of AI systems, if a deterministic implementation is possible, it will likely be favorable in many circumstances. Traditionally coded systems simply allow developers to use the wide range of existing supplementary tools for this type of system, e.g. for bug detection, safety analysis, etc. This way the likelihood of an unexpected behavior can be systematically reduced instead of relying on probabilities. Meanwhile, there will also be use cases where a manually coded solution is not tenable. Thus, developers will likely have to work with different development paradigms anyway, so their tools should reflect that as well instead of attempting a one-size-fits-all approach.

What remains to be seen is the path towards this: given the rich landscape of traditional development tools, one option would be to extend them with features for data-driven development. However, some traditional tools that have comprehensive support for traditional development, like IDEs, are already heavily burdened with features. Adding more to that might overwhelm users. Instead, this might also be a great opportunity to take stock of what developers actually require. The rising popularity of tools like computational notebooks [230] is some evidence that there are seemingly many developers who value features like the rapid feedback look they offer over many other features that they would have in a traditional IDE. This offers an alternative path: starting with a relatively simple but extensible tool, like computational notebooks, and adding functionality that developers want and need and that offers a provable benefit. Features like remote collaboration (cf. [305]) or LLM integration [314] may then take greater precedence than some historical features that have become less important, leading to tools that are more tailored towards modern Software Engineers. Whatever path the field takes, though, it is clear that AI will be an essential part of software development going forward.

As Software Engineering evolves and new, different development paradigms like data-driven development appear, development tools must also evolve to accommodate this.

- With new, fundamentally different development paradigms, we must consider for which context past, current, and future Software Engineering research was conducted and whether it applies to development in general or only specific development paradigms.
- To accommodate this multi-paradigm world, novel and evolving tools need to either specialize or consider the requirements of these different paradigms.

4.6 Developing Software beyond Software Developers

Besides the developers that immediately and most directly benefit from these tools, the software development process includes several other stakeholders. Communicating with them, negotiating expectations, eliciting requirements, etc. are an equally important part of Software Engineering. Fortunately, AI systems can support this process as well: they allow stakeholders to translate their natural descriptions to code and vice versa, summarize and explain systems, and generally enable them to engage with the development process in a highly flexible fashion without requiring a lot of technical expertise.

As AI increases the productivity of professional developers, it also lifts novices to a level where they are now able to perform tasks that were previously inaccessible, e.g., by quickly generating simple applications. This can be leveraged by stakeholders to communicate features of software by example, which can be a way to eliminate miscommunication, even if the generated prototype may ultimately not be the foundation of the resulting system.

By lowering the entry barrier to programming, there is a chance that more people give it a try. Generating applications may not necessarily help with understanding, but this lower entry barrier may still motivate more people to inform themselves about the technology. With the line between professional and end-user developers blurring, it might help in the goal of making computational literacy a skill for everyone. However, this also introduces the risk that end-users and customers may assume that their experience of quickly generating small pieces of software scales to larger systems, thus skewing expectations. Since first iterations require little effort, novices and customers might expect that the whole development process becomes little more than prompting an LLM, which may negatively affect appreciation of the trade and, in consequence, compensation. Once more this shows how important it is to understand these developments from a human-centered perspective to ensure all stakeholders in software development understand the challenges and limitations and have the best conditions to apply AI professionally or as amateurs, for small apps or large systems, and in a way that efficiently and effectively offers support.

> By using AI, software development becomes more accessible. This can help with communication and blur the line between professional Software Engineers, novices, and end-user developers. While this can contribute to computational literacy and better understanding, it might as well skew the perception of what software development entails and set false expectations.

5 Conclusion and Outlook

Just as the increasing ubiquity of AI technology affects many other domains, the thesis demonstrates how it also influences software developers and how they work. To this end, we first investigated how AI technology can be leveraged to enhance existing software development tools and create new support mechanisms for developers. We demonstrated the potential of AI technology for personalization and adaptation of IDEs as a common tool used by software developers, by implementing adaptive IDEs using different user data and utilizing different adaptations. While these were viewed generally positively by developers in the evaluation, they voiced concerns about the flexibility of these systems and a potential loss of control.

Furthermore, we analyzed the effect of LLMs as a current example of generative AI, on the behavior of developers and the productivity while writing code. Using contemporary code generation models, we were able to show considerable productivity benefits. Additionally, participants in our study demonstrated distinct usage patterns in different situations, emphasizing the effect that these systems already have on developer behavior.

Since software developers do not just benefit from AI-enhanced tools, but are also able to create and shape AI, we then moved to investigate how data-driven development as a new development paradigm affects software developers and their behavior. For this, we first determined that data-driven development introduces changes to the development process on a technical level, in the form of structural differences in the code. These differences also lead to changes in developer behavior, which we showed by the difference in code reading behavior, which developers exhibit depending on whether they read code of traditional or data-driven software.

As the literature showed, visualization is a frequently used tool in data-driven development. Thus, we finally explored how the new development paradigm of data-driven development can be supported using established alternative tooling paradigms, namely graphical programming. For this evaluation, we implemented a graphical programming tool. Similar to previous evaluations, participants rated the graphical programming favorably but had reservations about adopting it, citing concerns about flexibility. Based on the feedback, we extended the tool into a multi-paradigm editor which not only supports graphical development but also offers a presentation similar to the popular computational notebooks, giving the developers the flexibility to choose the right tool at any given time. Having these two development paradigms side-by-side with minimal switching overhead was able to mitigate some of the participants' concerns. Additionally, participants offered several specific scenarios in which they would benefit from this multi-paradigm approach.

The line of research presented in this thesis clearly shows how data-driven software influences the lives, work, and behavior of software developers. We contribute insights into how existing tools can be extended and how newly emerging tools play an important role in software development. The differences between traditional and data-driven software development are already affecting and will likely continue to affect tool support and the behavior of software developers. Given these effects, these insights further emphasize the importance of a human-centered perspective on these developments in Software Engineering and Artificial Intelligence. Understanding how these developments affect software developers will continue to be an essential part of ensuring the effective, efficient, satisfactory, and overall successful adoption of AI by software developers.

5.1 Outlook

Of course, this work can only present a snapshot of the influence of AI on the state of software developers. As AI and related technologies advance, the work practice too will evolve and change, potentially becoming more abstract and high-level, while the low-level details may be taken over by AI-supported systems. Still, research can continue to shape and inform this process.

As AI systems evolve, how people perceive them will also change, and with it the expectations and requirements for using them. We already see this with LLMs: while users initially were astounded by their impressive functionality, their perception has evolved and LLMs have become a feature that is integrated into existing software where it is often used as yet another productivity tool. However, with the challenges outlined above, the context in which these systems are used will also change.

With an increase in the ubiquity of AI, and in an effort to address its challenges, we will likely see more regulation. The “Artificial Intelligence Act” [82] of the European Union is just one of these examples. Choices made in the regulatory process will have an impact on software developers as well. For example, the AI Act expects certain systems to “be designed in a manner to enable deployers to understand how the AI system works” [82]. This can mean an increased level of explainability also in software development tools. Given the complexity and opaque nature of data-driven systems, means and methods that help developers understand what is happening in these systems will be invaluable for debugging and similar development efforts. However, at the same time, this also introduces new complexity in the form of functional and non-functional requirements to the development process. When software developers create data-driven software, they will have to be aware of these regulations and requirements in order to implement them. In consequence, research must give developers the means to do this without adding or at least by mitigating additional complexity.

Considering the fact that this work has shown that there are some fundamental differences between traditional and data-driven code, both from a technical and a human-centered perspective, it may be necessary to re-evaluate some of the existing research on Software Engineering and broader computer science to determine whether they are equally applicable to this area. If data-driven development is a part of software development as a whole, some assertions from past research and practice may not generally hold. Instead, prior findings and future research will need to consider in which context it has been or will be conducted to make an informed decision whether it applies to software development in general or only to specific development paradigms.

In addition, as data-driven development becomes even more widely adopted and a part of the skill set of a broad majority of developers, the existing infrastructure of developers will likely change and evolve. One example of this are the platforms for developer communities. Aside from the widely used platforms like GitHub or StackOverflow that originated in an era of traditional software, there are also platforms specifically for data-driven development like Kaggle or Hugging Face. As the community evolves and the lines between traditional and data-driven code potentially blur, the landscape of these platforms may also change, e.g., either by consolidation or by separating concerns in specialized platforms. While this is in part a consequence of how the communities interact, other factors, e.g., business decisions, also play a role. This is a point where well-informed research, particularly with a human-centered focus, can provide valuable input. For example, the feedback we received about how developers value control and flexibility or how behavioral patterns emerge for retrieving information with AI tools, as well as research building on this, can inform the decisions of community leaders. How this community infrastructure then evolves can also feed back into how the developer community as a whole changes, e.g., by creating sub-communities.

For this, education will also play a fundamental role. Depending on how educators treat the development paradigms, i.e. either by treating data-driven development as a distinctly different domain, e.g., with separate lectures, or by integrating it as one aspect of holistic software development, will inform how the next generation of software developers treat traditional and data-driven development. As discussed before, this also means giving students the skills necessary to effectively utilize data-driven support mechanisms, like how to program using LLMs, but also the foundational knowledge that allows students to actually understand software and not be dependent on AI.

With the exact future direction of the field cannot be certain, it is clear that the mentioned legal, academic, community, and educational aspects, will continue to change, as well as many more. As this thesis provides only a snapshot of this evolution, it is now up to future research to support and advise how software developers should be and are affected by AI.

Contributing Publications

[C1] Matthias Schmidmaier, Zhiwei Han, Thomas Weber, Yuanting Liu, Heinrich Hußmann. “Real-Time Personalization in Adaptive IDEs.” In: *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization*. UMAP’19 Adjunct. Larnaca, Cyprus: Association for Computing Machinery, 2019, pp. 81–86. ISBN: 9781450367110. DOI: 10.1145/3314183.3324975.

[C2] Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, Sven Mayer. “Significant Productivity Gains through Programming with Large Language Models.” In: *Proc. ACM Hum.-Comput. Interact.* 8.EICS (June 2024). DOI: 10.1145/3661145.

[C3] Thomas Weber, Janina Ehe, Sven Mayer. “Extending Jupyter with Multi-Paradigm Editors.” In: *Proc. ACM Hum.-Comput. Interact.* 8.EICS (June 2024). DOI: 10.1145/3660247.

[C4] Thomas Weber, Heinrich Hußmann. “Tooling for Developing Data-Driven Applications: Overview and Outlook.” In: *Proceedings of Mensch Und Computer 2022*. MuC ’22. Darmstadt, Germany: Association for Computing Machinery, 2022, pp. 66–77. ISBN: 9781450396905. DOI: 10.1145/3543758.3543779.

[C5] Thomas Weber, Sven Mayer. “Usability and Adoption of Graphical Tools for Data-Driven Development.” In: *Proceedings of Mensch Und Computer 2024*. MuC ’24. Karlsruhe, Germany: Association for Computing Machinery, 2024, 231–241. ISBN: 9798400709982. DOI: 10.1145/3670653.3670658.

[C6] Thomas Weber, Rafael Vinicius Mourao Thiel, Sven Mayer. “Supporting Software Developers Through a Gaze-Based Adaptive IDE.” In: *Proceedings of Mensch Und Computer 2023*. MuC ’23. Rapperswil, Switzerland: Association for Computing Machinery, 2023, pp. 267–276. ISBN: 9798400707711. DOI: 10.1145/3603555.3603571.

[C7] Thomas Weber, Christina Winiker, Heinrich Hussmann. “A Closer Look at Machine Learning Code.” In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI EA ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380959. DOI: 10.1145/3411763.3451679.

[C8] Thomas Weber, Christina Winiker, Sven Mayer. “An Investigation of How Software Developers Read Machine Learning Code.” In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’24. Barcelona, Spain: Association for Computing Machinery, 2024, pp. 165–176. ISBN: 9798400710476. DOI: 10.1145/3674805.3686678.

Contributions

	My contributions	Others' contributions
[C1]	In this project, I was responsible for interface and interaction design for the UI adaptions in general as well as the integration into the existing system. I also co-organized an internal workshop to collect the data that is the foundation for the implemented system and the architecture we proposed in this publication.	Zhiwei Han implemented the machine learning pipeline and, through shared discussions, proposed the general concept for the hybrid pipepline. Matthias Schmidmaier co-organized the workshop and co-wrote the publication.
[C6]	I was project lead, supervisor for the underlying Master's thesis as well as first author of the resulting publication. To this end, I conceptualized the project, the user interface and advised the student on system and study design. Finally, I analyzed the collected data.	This project is based on the Master's Thesis of Rafael Thiel. He implemented the ML system and the integration into Visual Studio. He also supervised the data collection and the evaluation. Sven Mayer provided feedback on the Machine Learning implementation.

	My contributions	Others' contributions
[C2]	I led this project and supervised the Master's Thesis which led to this publication. In this position, I advised the student throughout the whole process with a focus study design. I also analyzed the collected data.	This project is based on the Master's Thesis of Maximilian Brandmaier. As part of his thesis, he set up the automatic provisioning of the study environment and supervised the participants during the study. Sven Mayer advised on questions of study design and co-wrote some of the resulting publication.
[C7, C8]	I was project lead and supervisor of the Bachelor's and Project Thesis and first author for both publications. I collected and analyzed the data for the static analysis, supervised parts of the eye-tracking studies and performed the analysis for the collected gaze data.	Both the Bachelor's and Project Thesis were done by Christina Winiker. She set up the studies and supervised them. As part of the analysis, she wrote various scripts for displaying and replaying the gaze data and extracted the participant-specific offsets and the bounding boxes from the code. Both, Sven Mayer and Heinrich Hußmann provieded advice for the thesis and contributed to the publications.
[C4]	I performed this literature review including the collection and filtering of publications as well as summarizing them and extracting common themes and open questions.	Heinrich Hußmann supported this work by providing feedback on the selection process and the themes in the research field.

	My contributions	Others' contributions
[C3, C5]	This work is based on the development tool, which I implemented. In addition, I designed and conducted the initial study and analyzed the data. For the subsequent studies, I supervised the students that worked on extending this tool by providing them with guidance on both the high-level architecture and the concrete implementation as well as study design to evaluate their implementations.	After the initial implementation, the tool was extended by Florian Klement and Janina Ehe as part of the Master's Theses. Florian Klement worked on integrating the tool into the Jupyter infrastructure. Janina Ehe extended the interface into the hybrid system and evaluated it.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.

[2] Rabe Abdalkareem, Emad Shihab, Juergen Rilling. "What do developers use the crowd for? a study using stack overflow." In: *IEEE Software* 34.2 (2017), pp. 53–60. DOI: 10.1109/MS.2017.31.

[3] William B. Ackerman. "Data Flow Languages." In: *Computer* 15.2 (1982), pp. 15–25. DOI: 10.1109/MC.1982.1653938.

[4] Matt Adereth. *Silverman's Mode Estimation Method Explained*. http://adereth.github.io/blog/2014/10/12/silvermans-mode-detection-method-explained/. 2014.

[5] Rajas Agashe, Srinivasan Iyer, Luke Zettlemoyer. "JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation." In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5436–5446. DOI: 10.18653/v1/D19-1546.

[6] Pierre A. Akiki, Arosha K. Bandara, Yijun Yu. "Adaptive Model-Driven User Interface Development Systems." In: *ACM Comput. Surv.* 47.1 (2014), 9:1–9:33. DOI: 10.1145/2597999.

[7] Elena N. Akimova, Alexander Yu. Bersenev, Artem A. Deikov, Konstantin S. Kobylkin, Anton V. Konygin, Ilya P. Mezentsev, Vladimir E. Misilov. “A Survey on Software Defect Prediction Using Deep Learning.” In: *Mathematics* 9.11 (2021). ISSN: 2227-7390. DOI: 10.3390/math9111180.

[8] Sarah Alaghbari, Annett Mitschick, Gregor Blichmann, Martin Voigt, Raimund Dachselt. “Achiever or explorer? gamifying the creation process of training data for machine learning.” In: *Mensch und Computer 2020 - Tagungsband, Magdebug, Germany, September 6-9, 2020*. Ed. by Florian Alt, Stefan Schneegass, Eva Hornecker. ACM, 2020, pp. 173–181. DOI: 10.1145/3404983.3405519.

[9] Maqbool Ali, Rahman Ali, Wajahat Ali Khan, Soyeon Caren Han, Jae Hun Bang, Tae Ho Hur, Dohyeong Kim, Sungyoung Lee, Byeong Ho Kang. “A Data-Driven Knowledge Acquisition System: An End-to-End Knowledge Engineering Process for Generating Production Rules.” In: *IEEE Access* 6 (2018), pp. 15587–15607. DOI: 10.1109/ACCESS.2018.2817022.

[10] Sven Amann, Sebastian Proksch, Sarah Nadi. “FeedBaG: An interaction tracker for Visual Studio.” In: *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*. IEEE Computer Society, 2016, pp. 1–3. DOI: 10.1109/ICPC.2016.7503741.

[11] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald C. Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, Thomas Zimmermann. “Software engineering for machine learning: a case study.” In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE (SEIP) 2019. Montreal, QC, Canada: IEEE / ACM, 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042.

[12] Saleema Amershi, James Fogarty, Ashish Kapoor, Desney S. Tan. “Effective End-User Interaction with Machine Learning.” In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by Wolfram Burgard, Dan Roth. AAAI Press, 2011.

[13] R. Anil, R. Danymol, Harsha Gawande, R. Gandhiraj. “Machine learning plugins for GNU Radio Companion.” In: *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*. 2014, pp. 1–5. DOI: 10.1109/ICGCCEE.2014.6922250.

[14] S. Antoy, P. Forcheri, M.T. Molfino. “ Specification-based code generation.” In: *Twenty-Third Annual Hawaii International Conference on System Sciences*. Vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, Jan. 1990, 165–173 vol.2. DOI: 10.1109/HICSS.1990.205185.

[15] Muhammad Waseem Anwar, Muhammad Rashid, Farooque Azam, Muhammad Kashif, Wasi Haider Butt. “A model-driven framework for design and verification of embedded systems through SystemVerilog.” In: *Des. Autom. Embed. Syst.* 23.3-4 (2019), pp. 179–223. DOI: 10.1007/s10617-019-09229-y.

[16] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, Daqing Hou. “CSCC: Simple, Efficient, Context Sensitive Code Completion.” In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 71–80. DOI: 10.1109/ICSME.2014.29.

[17] Mojtaba Bagherzadeh, Francis Bordeleau, Jean-Michel Bruel, Jurgen Dingel, Sébastien Gérad, Nicolas Hili, Sebastian Voss. “Summary of Workshop on Model-Driven Engineering Tools (MDETools’17).” In: *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017*. CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[18] Abdul Ali Bangash, Hareem Sahar, Shaiful Alam Chowdhury, Alexander William Wong, Abram Hindle, Karim Ali. “What do developers know about machine learning: a study of ML discussions on StackOverflow.” In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR 2019. Montreal, QC, Canada: IEEE / ACM, 2019, pp. 260–264. DOI: 10.1109/MSR.2019.00052.

[19] Shraddha Barke, Michael B. James, Nadia Polikarpova. *Grounded Copilot: How Programmers Interact with Code-Generating Models*. 2022. DOI: 10.48550/ARXIV.2206.15000.

[20] Vivek Basanagoudar, Abhijay Srekanth. “Copyright Conundrums in Generative AI: Github Copilot’s Not-So-Fair Use of Open-Source Licensed Code.” eng. In: *Journal of Intellectual Property Studies* 7.2 (2023), pp. 58–68.

[21] F. L. Bauer, L. Bolliet, H. J. Helms. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. Ed. by Peter Naur, Brian Randell. 1969.

[22] Roman Bednarik. "Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations." In: *Int. J. Hum. Comput. Stud.* 70.2 (2012), pp. 143–155. DOI: 10.1016/j.ijhcs.2011.09.003.

[23] Roman Bednarik, Markku Tukiainen. "An eye-tracking methodology for characterizing program comprehension processes." In: *Proceedings of the Eye Tracking Research & Application Symposium, ETRA 2006, San Diego, California, USA, March 27-29, 2006*. Ed. by Kari-Jouko Räihä, Andrew T. Duchowski. ETRA '06. San Diego, California: ACM, 2006, pp. 125–132. ISBN: 1595933050. DOI: 10.1145/1117309.1117356.

[24] Tal Ben-Nun, Alice Shoshana Jakobovits, Torsten Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics." In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018*. NeurIPS 2018. Montreal, QC, Canada, 2018, pp. 3589–3601.

[25] David Benyon. "Accommodating Individual Differences through an Adaptive User Interface." In: *Human Factors in Information Technology* 10 (1993), pp. 149–149.

[26] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, Bernd Wiswedel. "KNIME - the Konstanz Information Miner: Version 2.0 and Beyond." In: *SIGKDD Explor. Newsl.* (Nov. 2009).

[27] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, Idan Gazit. "Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools." In: *Queue* 20.6 (Jan. 2023), pp. 35–57. ISSN: 1542-7730. DOI: 10.1145/3582083.

[28] Mohamed Bjaoui, Houneida Sakly, Mourad Said, Naoufel Kraiem, Mohamed Salim Bouhlel. "Depth Insight for Data Scientist with RapidMiner "an Innovative Tool for AI and Big Data towards Medical Applications"." In: *Proceedings of*

the 2nd International Conference on Digital Tools & Uses Congress. DTUC '20. Virtual Event, Tunisia: Association for Computing Machinery, 2020. ISBN: 9781405377539. DOI: 10.1145/3423603.3424059.

[29] Marco Brambilla, Jordi Cabot, Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012. ISBN: 978-3-031-02546-4. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[30] John Brooke. "SUS - A quick and dirty usability scale." In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[31] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei. *Language Models are Few-Shot Learners*. 2020. DOI: 10.48550/ARXIV.2005.14165.

[32] Marcel Bruch, Martin Monperrus, Mira Mezini. "Learning from Examples to Improve Code Completion Systems." In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 213–222. ISBN: 9781605580012. DOI: 10.1145/1595696.1595728.

[33] Eli Brumbaugh, Atul Kale, Alfredo Luque, Bahador Nooraei, John Park, Krishna Puttaswamy, Kyle Schiller, Evgeny Shapiro, Conglei Shi, Aaron Siegel, Nikhil Simha, Mani Bhushan, Marie Sbrocca, Shi-Jing Yao, Patrick Yoon, Varant Zanoyan, Xiao-Han T. Zeng, Qiang Zhu, Andrew Cheong, Michelle Gu-Qian Du, Jeff Feng, Nick Handel, Andrew Hoh, Jack Hone, Brad Hunter. "Bighead: A Framework-Agnostic, End-to-End Machine Learning Platform." In: *2019 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2019, Washington, DC, USA, October 5-8, 2019*. Ed. by Lisa Singh, Richard D. De Veaux, George Karypis, Francesco Bonchi, Jennifer Hill. IEEE, 2019, pp. 551–560. DOI: 10.1109/DSAA.2019.00070.

[34] Tyson Bulmer, Lloyd Montgomery, Daniela Damian. "Predicting Developers' IDE Commands with Machine Learning." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 82–85. ISBN: 9781450357166. DOI: 10.1145/3196398.3196459.

[35] Andrea Bunt, Giuseppe Carenini, Cristina Conati. "Adaptive Content Presentation for the Web." In: *The Adaptive Web*. Ed. by Peter Brusilovsky, Alfred Kobsa, Wolfgang Nejdl. Berlin, Heidelberg: Springer-Verlag, 2007. Chap. Adaptive Content Presentation for the Web, pp. 409–432.

[36] Margaret M. Burnett, David W. McIntyre. "Visual Programming - Guest Editors' Introduction." In: *Computer* 28.3 (1995), pp. 14–16. DOI: 10.1109/MC.1995.10027.

[37] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, Sascha Tamm. "Eye Movements in Code Reading: Relaxing the Linear Order." In: *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. ICPC '15. USA: IEEE Computer Society, 2015, pp. 255–265. ISBN: 9781467381598. DOI: 10.1109/ICPC.2015.36.

[38] Carrie J. Cai, Philip J. Guo. "Software Developers Learning Machine Learning: Motivations, Hurdles, and Desires." In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC 2019. Memphis, Tennessee, USA: IEEE Computer Society, 2019, pp. 25–34. DOI: 10.1109/VLHCC.2019.8818751.

[39] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, Jean Vanderdonckt. "A Unifying Reference Framework for multi-target user interfaces." In: *Interact. Comput.* 15.3 (2003), pp. 289–308. DOI: 10.1016/S0953-5438(03)00010-9.

[40] Jose Manuel Cantera Fonseca. *Model-Based UI XG Final Report*. In collab. with Juan M. González Calleros, Gerrit Meixner, Fabio Paternò, Jaroslav Pullmann, Dave Raggett, Daniel Schwabe, Jean Vanderdonckt. 2010.

[41] Stuart K. Card, Thomas P. Moran, Allen Newell. "The Keystroke-Level Model for User Performance Time with Interactive Systems." In: *Commun. ACM* 23.7 (July 1980), pp. 396–410. ISSN: 0001-0782. DOI: 10.1145/358886.358895.

[42] Alberto Castellini, Francesco Masillo, Riccardo Sartea, Alessandro Farinelli. “eXplainable Modeling (XM): Data Analysis for Intelligent Agents.” In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS ’19, Montreal, QC, Canada, May 13-17, 2019*. Ed. by Edith Elkind, Manuela Veloso, Noa Agmon, Matthew E. Taylor. International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 2342–2344.

[43] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, Titus Barik. “What’s Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities.” In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Association for Computing Machinery, 2020, pp. 1–12. ISBN: 9781450367080. DOI: `10.1145/3313831.3376729`.

[44] Hsinchun Chen. “Machine Learning for Information Retrieval: Neural Networks, Symbolic Learning, and Genetic Algorithms.” In: *J. Am. Soc. Inf. Sci.* 46.3 (1995), pp. 194–216. DOI: `10.1002/(SICI)1097-4571(199504)46:3\<194::AID-ASI4\>3.0.CO;2-S`.

[45] Lingjiao Chen, Matei Zaharia, James Zou. *How is ChatGPT’s behavior changing over time?* 2023.

[46] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, Wojciech Zaremba. “Evaluating Large Language Models Trained on Code.” In: *CoRR* abs/2107.03374 (2021).

[47] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov,

Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, Wojciech Zaremba. "Evaluating Large Language Models Trained on Code." In: *CoRR* abs/2107.03374 (2021), p. 35.

[48] Mon Chu Chen, John R. Anderson, Myeong Ho Sohn. "What Can a Mouse Cursor Tell Us More? Correlation of Eye/Mouse Movements on Web Browsing." In: *CHI '01 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '01. Seattle, Washington: Association for Computing Machinery, 2001, pp. 281–282. ISBN: 1581133405. DOI: 10.1145/634067.634234.

[49] Dongjin Choi, Sara Evensen, Çagatay Demiralp, Estevam Hruschka. "TagRuler: Interactive Tool for Span-Level Data Programming by Demonstration." In: *Companion of The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. Ed. by Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, Leila Zia. ACM / IW3C2, 2021, pp. 673–677. DOI: 10.1145/3442442.3458602.

[50] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, Noah Fiedel. *PaLM: Scaling Language Modeling with Pathways*. 2022. DOI: 10.48550/ARXIV.2204.02311.

[51] Michael Chromik. “Making SHAP Rap: Bridging Local and Global Insights Through Interaction and Narratives.” In: *Human-Computer Interaction – INTERACT 2021*. Ed. by Carmelo Ardito, Rosa Lanzilotti, Alessio Malizia, Helen Petrie, Antonio Piccinno, Giuseppe Desolda, Kori Inkpen. Cham: Springer International Publishing, 2021, pp. 641–651. ISBN: 978-3-030-85616-8.

[52] Michael Chromik, Malin Eiband, Felicitas Buchner, Adrian Krüger, Andreas Butz. “I Think I Get Your Point, AI! The Illusion of Explanatory Depth in Explainable AI.” In: *26th International Conference on Intelligent User Interfaces*. IUI ’21. College Station, TX, USA: Association for Computing Machinery, 2021, 307–317. ISBN: 9781450380171. DOI: 10.1145/3397481.3450644.

[53] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, Neel Sundaresan. *PyMT5: multi-mode translation of natural language and Python code with transformers*. 2020. DOI: 10.18653/v1/2020.emnlp-main.728.

[54] Alain Colmerauer, Philippe Roussel. “The Birth of Prolog.” In: *History of Programming Languages—II*. Association for Computing Machinery, 1996, pp. 331–367. ISBN: 0201895021.

[55] Cristina Conati, Christina Merten, Saleema Amershi, Kasia Muldner. “Using Eye-Tracking Data for High-Level User Modeling in Adaptive Interfaces.” In: *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*. AAAI’07. Vancouver, British Columbia, Canada: AAAI Press, 2007, pp. 1614–1617. ISBN: 9781577353232.

[56] Allen Cypher, Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. en. MIT Press. London, England: MIT Press, Aug. 1993.

[57] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, Zhen Ming, Jiang. *GitHub Copilot AI pair programmer: Asset or Liability?* 2023. DOI: 10.1016/j.jss.2023.111734.

[58] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, Michel C. Desmarais. “Effective test generation using pre-trained Large Language Models and mutation testing.” In: *Information and Software Technology* 171 (2024), p. 107468. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2024.107468.

[59] Edwin S. Dalmaijer, Sebastiaan Mathôt, Stefan Van der Stigchel. “PyGaze: An open-source, cross-platform toolbox for minimal-effort programming of eyetracking experiments.” In: *Behavior Research Methods* 46.4 (Dec. 2014), pp. 913–921. ISSN: 1554-3528. DOI: 10.3758/s13428-013-0422-2.

[60] Kostadin Damevski, Hui Chen, David C. Shepherd, Nicholas A. Kraft, Lori Pollock. “Predicting Future Developer Behavior in the IDE Using Topic Models.” In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 932. ISBN: 9781450356381. DOI: 10.1145/3180155.3182541.

[61] Evans Data. *Global Developer Population and Demographic Study 2023*. Aug. 2023.

[62] Fred D. Davis. “Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology.” In: *MIS Quarterly* 13.3 (1989), pp. 319–340. ISSN: 02767783.

[63] Stephan De Spiegeleire, Matthijs Maas, Tim Sweijs. *Artificial intelligence and the future of defense: strategic implications for small-and medium-sized force providers*. The Hague Centre for Strategic Studies, 2017.

[64] Arthur R. T. de Lacerda, Carla S. R. Aguiar. “FLOSS FAQ chatbot project reuse: how to allow nonexperts to develop a chatbot.” In: *Proceedings of the 15th International Symposium on Open Collaboration, OpenSym 2019, Skövde, Sweden, August 20-22, 2019*. Ed. by Björn Lundell, Jonas Gamalielsson, Lorraine Morgan, Gregorio Robles. ACM, 2019, 3:1–3:8. DOI: 10.1145/3306446.3340823.

[65] Rob DeLine. “Modern software is all about data. development environments should be, too. (keynote).” In: *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich, Patrick Eugster. ACM, 2015, p. 4. DOI: 10.1145/2814189.2833213.

[66] Robert DeLine, Danyel Fisher. “Supporting exploratory data analysis with live programming.” In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. Atlanta, GA, USA: IEEE Computer Society, 2015, pp. 111–119. DOI: 10.1109/VLHCC.2015.7357205.

[67] Robert DeLine, Kael Rowan. “Code canvas: zooming towards better development environments.” In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, Sebastián Uchitel. ACM, 2010, pp. 207–210. DOI: 10.1145/1810295.1810331.

[68] Janez Demsar, Tomaz Curk, Ales Erjavec, Crtomir Gorup, Tomaz Hocevar, Mitar Milutinovic, Martin Mozina, Matija Polajnar, Marko Toplak, Anze Staric, Miha Stajdohar, Lan Umek, Lan Zagar, Jure Zbontar, Marinka Zitnik, Blaz Zupan. “Orange: data mining toolbox in python.” In: *J. Mach. Learn. Res.* 14.1 (2013), pp. 2349–2353.

[69] Janez Demsar, Blaz Zupan, Gregor Leban, Tomaz Curk. “Orange: From Experimental Machine Learning to Interactive Data Mining.” In: *Knowledge Discovery in Databases: PKDD 2004, 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy, September 20-24, 2004, Proceedings*. Ed. by Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, Dino Pedreschi. Vol. 3202. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 537–539. DOI: 10.1007/978-3-540-30116-5_58.

[70] Charles-Eric Dessart, Vivian Genaro Motti, Jean Vanderdonckt. “Showing User Interface Adaptivity by Animated Transitions.” In: *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS ’11. Pisa, Italy: Association for Computing Machinery, 2011, pp. 95–104. ISBN: 9781450306706. DOI: 10.1145/1996461.1996501.

[71] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[72] Amanda Gonçalves Dias, Evangelos E. Milios, Maria Cristina Ferreira de Oliveira. “TRIVIR: A Visualization System to Support Document Retrieval with High Recall.” In: *Proceedings of the ACM Symposium on Document Engineering 2019, Berlin, Germany, September 23-26, 2019*. Ed. by Sonja Schimmler, Uwe M. Borghoff. ACM, 2019, 10:1–10:10. DOI: 10.1145/3342558.3345401.

[73] Gonzalo I. Diaz, Achille Fokoue-Nkoutche, Giacomo Nannicini, Horst Samulowitz. “An effective algorithm for hyperparameter optimization of neural networks.” In: *IBM J. Res. Dev.* 61.4-5 (2017), 9:1–9:11. DOI: 10.1147/JRD.2017.2709578.

[74] Thomas Dohmke. “GitHub Copilot X: The AI-powered developer experience.” In: *Github Blog* (Mar. 2023). https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/.

[75] Eladio Domı´nguez, Beatriz Pérez, Ángel L. Rubio, Marı´a A. Zapata. “A systematic review of code generation proposals from state machine specifications.” In: *Information and Software Technology* 54.10 (2012), pp. 1045–1066. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.04.008.

[76] Chenhe Dong, Yinghui Li, Haifan Gong, Miaoxin Chen, Junxin Li, Ying Shen, Min Yang. “A Survey of Natural Language Generation.” In: *ACM Comput. Surv.* 55.8 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3554727.

[77] Steven M. Drucker, Danyel Fisher, Sumit Basu. “Helping Users Sort Faster with Adaptive Machine Learning Recommendations.” In: *Human-Computer Interaction – INTERACT 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 187–203. ISBN: 978-3-642-23765-2. DOI: 10.1007/978-3-642-23765-2_13.

[78] Margaret Efron. *Curious about the new ChatGPT Code Interpreter plug-in? Here’s how to get started.* https://medium.com/learning-data/curious-about-the-new-chatgpt-code-interpreter-plug-in-heres-how-to-get-started-541fd998c017. Aug. 2023.

[79] Malin Eiband, Daniel Buschek, Heinrich Hussmann. “How to Support Users in Understanding Intelligent Systems? Structuring the Discussion.” In: *IUI ’21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13-17, 2021*. Ed. by Tracy Hammond, Katrien Verbert, Dennis Parra, Bart P. Knijnenburg, John O’Donovan, Paul Teale. ACM, 2021, pp. 120–132. DOI: 10.1145/3397481.3450694.

[80] Malin Eiband, Daniel Buschek, Alexander Kremer, Heinrich Hussmann. “The Impact of Placebic Explanations on Trust in Intelligent Systems.” In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems,*

CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019. Ed. by Regan L. Mandryk, Stephen A. Brewster, Mark Hancock, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos, Mark Perry. ACM, 2019. DOI: 10.1145/3290607.3312787.

[81] Alexander Eremeev, Sergey Ivliev, Alexander Kozhukhov. “Tool Environment for Creating Training Prototypes of Intelligent Decision Support Systems.” In: *2020 V International Conference on Information Technologies in Engineering Education (Inforino)*. 2020, pp. 1–4. DOI: 10.1109/Inforino48376.2020.9111781.

[82] European Union. *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence and amending Regulations (EC) No 300/2008, (EU) No 167/2013, (EU) No 168/2013, (EU) 2018/858, (EU) 2018/1139 and (EU) 2019/2144 and Directives 2014/90/EU, (EU) 2016/797 and (EU) 2020/1828 (Artificial Intelligence Act) (Text with EEA relevance)*. Official Journal of the European Union, L 162, 20 June 2024. June 2024.

[83] Stefano Federici. “A Minimal, Extensible, Drag-and-Drop Implementation of the C Programming Language.” In: *Proceedings of the 2011 Conference on Information Technology Education*. SIGITE ’11. West Point, New York, USA: Association for Computing Machinery, 2011, pp. 191–196. ISBN: 9781450310178. DOI: 10.1145/2047594.2047646.

[84] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, Ming Zhou. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. DOI: 10.48550/ARXIV.2002.08155.

[85] Leah Findlater, Krzysztof Z. Gajos. “Design Space and Evaluation Challenges of Adaptive Graphical User Interfaces.” In: *AI Mag.* 30.4 (2009), pp. 68–73. DOI: 10.1609/aimag.v30i4.2268.

[86] Patrick Tobias Fischer, Saskia Kuliga, Mark Eisenberg, Ibni Amin. “Space is Part of the Product: Using AttrakDiff to Identify Spatial Impact on User Experience with Media Façades.” In: *Proceedings of the 7th ACM International Symposium on Pervasive Displays*. PerDis ’18. Munich, Germany: Association for Computing Machinery, 2018. ISBN: 9781450357654. DOI: 10.1145/3205873.3205875.

[87] Nicole Forsgren, Margaret-Anne D. Storey, Chandra Shekhar Maddila, Thomas Zimmermann, Brian Houck, Jenna L. Butler. "The SPACE of Developer Productivity: There's more to it than you think." In: *ACM Queue* 19.1 (Feb. 2021), pp. 20–48. DOI: 10.1145/3454122.3454124.

[88] Krzysztof Z. Gajos, Mary Czerwinski, Desney S. Tan, Daniel S. Weld. "Exploring the Design Space for Adaptive Graphical User Interfaces." In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. Venezia, Italy: Association for Computing Machinery, 2006.

[89] Herve Gallaire, Jack Minker, Jean-Marie Nicolas. "Logic and Databases: A Deductive Approach." In: *ACM Comput. Surv.* 16.2 (June 1984), pp. 153–185. ISSN: 0360-0300. DOI: 10.1145/356924.356929.

[90] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, Jean-Luc Dekeyser. "A Model-Driven Design Framework for Massively Parallel Embedded Systems." In: *ACM Trans. Embed. Comput. Syst.* 10.4 (2011), 39:1–39:36. DOI: 10.1145/2043662.2043663.

[91] Inc. Gartner. *Hype Cycle for Emerging Technologies, 2024*. Tech. rep. Gartner, Inc., 2024.

[92] Marko Gasparic, Tural Gurbanov, Francesco Ricci. "Context-aware integrated development environment command recommender systems." In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 688–693. DOI: 10.1109/ASE.2017.8115679.

[93] Christoph Gebhardt, Brian Hecox, Bas van Opheusden, Daniel Wigdor, James Hillis, Otmar Hilliges, Hrvoje Benko. "Learning Cooperative Personalized Policies from Gaze Data." In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. New Orleans, LA, USA: Association for Computing Machinery, 2019.

[94] Panagiotis Germanakos, Marios Belk, Argyris Constantinides, George Samaras. "The PersonaWeb System: Personalizing E-Commerce Environments based on Human Factors." In: *Posters, Demos, Late-breaking Results and Workshop Proceedings of the 23rd Conference on User Modeling, Adaptation, and Personalization*. Ed. by Alexandra I. Cristea, Judith Masthoff, Alan Said, Nava Tintarev. Vol. 1388. UMAP'15 Workshop Proceedings. Online: CEUR-WS.org, 2015, p. 4.

[95] Elizabeth Gibney. "AI models fed AI-generated data quickly spew nonsense." In: (July 2024).

[96] Fabian Göbel, Peter Kiefer, Ioannis Giannopoulos, Andrew T. Duchowski, Martin Raubal. "Improving Map Reading with Gaze-Adaptive Legends." In: *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*. Warsaw, Poland: Association for Computing Machinery, 2018.

[97] Oscar Gomez, Steffen Holter, Jun Yuan, Enrico Bertini. "ViCE: visual counterfactual explanations for machine learning models." In: *IUI '20: 25th International Conference on Intelligent User Interfaces, Cagliari, Italy, March 17-20, 2020*. Ed. by Fabio Paternò, Nuria Oliver, Cristina Conati, Lucio Davide Spano, Nava Tintarev. ACM, 2020, pp. 531–535. DOI: 10.1145/3377325.3377536.

[98] Danielle Gonzalez, Thomas Zimmermann, Nachiappan Nagappan. "The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub." In: *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim, Georgios Gousios, Sarah Nadi, Joseph Hejderup. ACM, 2020, pp. 431–442. DOI: 10.1145/3379597.3387473.

[99] Jochen Görtler, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, Donghao Ren, Rahul Nair, Marc Kirchner, Kayur Patel. "Neo: Generalizing Confusion Matrix Visualization to Hierarchical and Multi-Output Labels." In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391573. DOI: 10.1145/3491102.3501823.

[100] Stephen Gould. "DARWIN: a framework for machine learning and computer vision research and development." In: *J. Mach. Learn. Res.* 13 (2012), pp. 3533–3537.

[101] Sorin Mihai Grigorescu, Bogdan Trasnea, Tiberiu T. Cocias, Gigel Macesanu. "A survey of deep learning techniques for autonomous driving." In: *J. Field Robotics* 37.3 (2020), pp. 362–386. DOI: 10.1002/ROB.21918.

[102] L. Gugerty, G. Olson. "Debugging by Skilled and Novice Programmers." In: *SIGCHI Bull.* 17.4 (Apr. 1986), 171–174. ISSN: 0736-6906. DOI: 10.1145/22339.22367.

[103] Mazhar Hameed, Felix Naumann. “Data Preparation: A Survey of Commercial Tools.” In: *SIGMOD Rec.* 49.3 (2020), pp. 18–29. DOI: 10.1145/3444831.3444835.

[104] Jianchao Han, Juan C. Rodriguez, Mohsen Beheshti. “Diabetes Data Analysis and Prediction Model Discovery Using RapidMiner.” In: *International Conference on Bio-Science and Bio-Technology, BSBT 2008, part of the Second International Conference on Future Generation Communication and Networking, FGCN 2008, Volume 3, Hainan Island, China, December 13-15, 2008*. IEEE Computer Society, 2008, pp. 96–99. DOI: 10.1109/FGCN.2008.226.

[105] Zhiwei Han, Thomas Weber, Stefan Matthes, Yuanting Liu, Hao Shen. *Interactive Image Restoration*. 2019.

[106] Guy S. Handelman, Hong Kuan Kok, Ronil V. Chandra, Amir H. Razavi, Shiwei Huang, Mark Brooks, Michael J. Lee, Hamed Asadi. “Peering Into the Black Box of Artificial Intelligence: Evaluation Metrics of Machine Learning Methods.” In: *American Journal of Roentgenology* 212.1 (2019), pp. 38–43. DOI: 10.2214/AJR.18.20224.

[107] Christoph Hannebauer, Marc Hesenius, Volker Gruhn. “Does syntax highlighting help programming novices?” In: *Empir. Softw. Eng.* 23.5 (2018), pp. 2795–2828. DOI: 10.1007/S10664-017-9579-0.

[108] Hans-Jörg Happel, Walid Maalej. “Potentials and Challenges of Recommendation Systems for Software Development.” In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. Atlanta, Georgia: Association for Computing Machinery, 2008.

[109] Gunnar Harboe, Elaine M. Huang. “Real-World Affinity Diagramming Practices: Bridging the Paper-Digital Gap.” In: *Proc. 33rd Annual ACM Conf. Human Factors in Computing Systems*. Association for Computing Machinery, 2015.

[110] Jesse Harden, Elizabeth Christman, Nurit Kirshenbaum, Mahdi Belcaid, Jason Leigh, Chris North. ““There is no reason anybody should be using 1D anymore”: Design and Evaluation of 2D Jupyter Notebooks.” In: *Graphics Interface 2023* ().

[111] Jesse Harden, Elizabeth Christman, Nurit Kirshenbaum, John E. Wenskovitch, Jason Leigh, Chris North. “Exploring Organization of Computational Notebook Cells in 2D Space.” In: *2022 IEEE Symposium on Visual Languages and Human-*

Centric Computing, VL/HCC 2022, Rome, Italy, September 12-16, 2022. Ed. by Paolo Bottoni, Gennaro Costagliola, Michelle Brachman, Mark Minas. IEEE, 2022, pp. 1–6. DOI: 10.1109/VL/HCC53370.2022.9833128.

[112] Sandra G Hart. "NASA-task load index (NASA-TLX); 20 years later." In: *Proceedings of the human factors and ergonomics society annual meeting*. Vol. 50. Los Angeles, CA, USA: Sage publications Sage CA, 2006, pp. 904–908. DOI: 10.1177/154193120605000909.

[113] Marc Hassenzahl, Michael Burmester, Franz Koller. "AttrakDiff: Ein Fragebogen zur Messung wahrgenommener hedonischer und pragmatischer Qualität." In: *Mensch & Computer 2003: Interaktion in Bewegung, Stuttgart, Germany, September 7-10, 2003*. Ed. by Gerd Szwillus, Jürgen Ziegler. Wiesbaden: Teubner, 2003, pp. 187–196. DOI: 10.1007/978-3-322-80058-9_19.

[114] Marc Hassenzahl, Andrew Monk. "The Inference of Perceived Usability From Beauty." In: *Human–Computer Interaction* 25.3 (2010), pp. 235–260. DOI: 10.1080/07370024.2010.500139.

[115] Yichen He, Liran Wang, Kaiyi Wang, Yupeng Zhang, Hang Zhang, Zhoujun Li. "COME: Commit Message Generation with Modification Embedding." In: ISSTA 2023 (2023), pp. 792–803. DOI: 10.1145/3597926.3598096.

[116] Andrew Head, Fred Hohman, Titus Barik, Steven Mark Drucker, Robert DeLine. "Managing Messes in Computational Notebooks." In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*. Ed. by Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos. ACM, 2019, p. 270. DOI: 10.1145/3290605.3300500.

[117] Tim Hegeman, Matthijs Jansen, Alexandru Iosup, Animesh Trivedi. "GradeML: Towards Holistic Performance Analysis for Machine Learning Workflows." In: *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021, Companion Volume*. Ed. by Johann Bourcier, Zhen Ming (Jack) Jiang, Cor-Paul Bezemer, Vittorio Cortellessa, Daniele Di Pompeo, Ana Lucia Varbanescu. ACM, 2021, pp. 57–63. DOI: 10.1145/3447545.3451185.

[118] Prateek Hejmady, N. Hari Narayanan. “Visual attention patterns during program debugging with an IDE.” In: *Proceedings of the 2012 Symposium on Eye-Tracking Research and Applications, ETRA 2012, Santa Barbara, CA, USA, March 28-30, 2012*. Ed. by Carlos Hitoshi Morimoto, Howell O. Istance, Stephen N. Spencer, Jeffrey B. Mulligan, Pernilla Qvarfordt. ETRA ’12. Santa Barbara, California: ACM, 2012, pp. 197–200. ISBN: 9781450312219. DOI: 10.1145/2168556.2168592.

[119] Adrián Hernández-López, Ricardo Colomo-Palacios, Ángel García-Crespo. “Software engineering job productivity-a systematic review.” In: *International Journal of Software Engineering and Knowledge Engineering* 23.03 (Apr. 2013), pp. 387–406. DOI: 10.1142/s0218194013500125.

[120] Marc Hesenius, Nils Schwenzfeier, Ole Meyer, Wilhelm Koop, Volker Gruhn. “Towards a software engineering process for developing data-driven applications.” In: *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2019, Montreal, QC, Canada, May 28, 2019*. Ed. by Tim Menzies, Burak Turhan. IEEE / ACM, 2019, pp. 35–41. DOI: 10.1109/RAISE.2019.00014.

[121] Geert Heyman, Rafael Huysegems, Pascal Justen, Tom Van Cutsem. “Natural Language-Guided Programming.” In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 39–55. ISBN: 9781450391108. DOI: 10.1145/3486607.3486749.

[122] Andreas Hinterreiter, Peter Ruch, Holger Stitz, Martin Ennemoser, Jurgen Bernard, Hendrik Strobelt, Marc Streit. “ConfusionFlow: A model-agnostic visualization for temporal analysis of classifier confusion.” In: *IEEE Transactions on Visualization and Computer Graphics* (2020), pp. 1–1. DOI: 10.1109/TVCG.2020.3012063.

[123] Fred Hohman, Andrew Head, Rich Caruana, Robert DeLine, Steven Mark Drucker. “Gamut: A Design Probe to Understand How Data Scientists Understand Machine Learning Models.” In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*. Ed. by Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos. ACM, 2019, p. 579. DOI: 10.1145/3290605.3300809.

[124] Fred Hohman, Kanit Wongsuphasawat, Mary Beth Kery, Kayur Patel. “Understanding and Visualizing Data Iteration in Machine Learning.” In: *CHI ’20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. Association for Computing Machinery, 2020.

[125] Reid Holmes, Gail C. Murphy. “Using Structural Context to Recommend Source Code Examples.” In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 117–125. ISBN: 1581139632. DOI: 10.1145/1062455.1062491.

[126] Andreas Holzinger, Chris Biemann, Constantinos S. Pattichis, Douglas B. Kell. “What do we need to build explainable AI systems for the medical domain?” In: *CoRR* abs/1712.09923 (2017).

[127] Jeremy Howard, Sebastian Ruder. “Universal Language Model Fine-tuning for Text Classification.” In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, 2018, pp. 328–339. DOI: 10.18653/v1/p18-1031.

[128] Krystal Hu, Anna Tong. “OpenAI and others seek new path to smarter AI as current methods hit limitations.” In: (Nov. 2024). https://www.reuters.com/technology/artificial-intelligence/openai-rivals-seek-new-path-smarter-ai-current-methods-hit-limitations-2024-11-11/.

[129] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, Heming Cui. *Bias Testing and Mitigation in LLM-based Code Generation*. 2024.

[130] Kevin Hurler. *Stack Overflow Traffic Drops as Coders Opt for ChatGPT Help Instead*. https://gizmodo.com/stack-overflow-traffic-drops-as-coders-opt-for-chatgpt-1850427794. May 2023.

[131] IEEE. *IEEE Guide for the Use of Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE)*. 2014. DOI: 10.1109/IEEESTD.2014.6922153.

[132] Samuel Idowu, Daniel Strüber, Thorsten Berger. “Asset Management in Machine Learning: A Survey.” In: *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 51–60. DOI: 10.1109/ICSE-SEIP52600.2021.00014.

[133] Toyomi Ishida, Hidetake Uwano. "Synchronized Analysis of Eye Movement and EEG during Program Comprehension." In: *Proceedings of the 6th International Workshop on Eye Movements in Programming*. EMIP '19. Montreal, Quebec, Canada: IEEE Press, 2019, 26–32. DOI: 10.1109/EMIP.2019.00012.

[134] Anthony David Jameson. "Understanding and Dealing With Usability Side Effects of Intelligent Processing." In: *AI Mag.* 30.4 (2009), pp. 23–40. DOI: 10.1609/aimag.v30i4.2274.

[135] Dietmar Jannach, Michael Jugovac, Lukas Lerche. "Supporting the Design of Machine Learning Workflows with a Recommendation System." In: *ACM Trans. Interact. Intell. Syst.* 6.1 (2016), 8:1–8:35. DOI: 10.1145/2852082.

[136] Dietmar Jannach, Ahtsham Manzoor, Wanling Cai, Li Chen. "A Survey on Conversational Recommender Systems." In: *ACM Comput. Surv.* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3453154.

[137] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, Michael Terry. "Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models." In: *CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: ACM, Apr. 2022. ISBN: 9781450391573. DOI: 10.1145/3491102.3501870.

[138] Xianhao Jin, Francisco Servant. "The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-List Length on Its Efficiency." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 70–73. ISBN: 9781450357166. DOI: 10.1145/3196398.3196474.

[139] Jeremiah W. Johnson, Karen H. Jin. "Jupyter Notebooks in Education." In: *J. Comput. Sci. Coll.* 35.8 (Apr. 2020), 268–269. ISSN: 1937-4771.

[140] Hank Kahney. "What Do Novice Programmers Know about Recursion." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '83. Boston, Massachusetts, USA: Association for Computing Machinery, 1983, 235–239. ISBN: 0897911210. DOI: 10.1145/800045.801618.

[141] Eirini Kalliamvakou. "Research: Quantifying github copilot's impact on developer productivity and happiness." In: (2022). https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/.

[142] Andrej Karpathy. *Software 2.0*. https://karpathy.medium.com/software-2-0-a64152b37c35. 2017.

[143] Harmanpreet Kaur, Harsha Nori, Samuel Jenkins, Rich Caruana, Hanna M. Wallach, Jennifer Wortman Vaughan. "Interpreting Interpretability: Understanding Data Scientists' Use of Interpretability Tools for Machine Learning." In: *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, Rafal Kocielnik. ACM, 2020, pp. 1–14. DOI: 10.1145/3313831.3376219.

[144] Simeon Keates, P. John Clarkson, Lee-Anne Harrison, Peter Robinson. "Towards a Practical Inclusive Design Approach." In: *Proceedings on the 2000 Conference on Universal Usability*. Arlington, Virginia, USA: Association for Computing Machinery, 2000.

[145] Steven Kelly, Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. ISBN: 978-0-470-03666-2.

[146] Mik Kersten, Gail C. Murphy. "Using Task Context to Improve Programmer Productivity." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 1–11. ISBN: 1595934685. DOI: 10.1145/1181775.1181777.

[147] Mary Beth Kery, Amber Horvath, Brad Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, 1265–1276. ISBN: 9781450346559. DOI: 10.1145/3025453.3025626.

[148] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, Brad A. Myers. "The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–11. ISBN: 9781450356206. DOI: 10.1145/3173574.3173748.

[149] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, Brad A. Myers. "The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–11. ISBN: 9781450356206. DOI: 10.1145/3173574.3173748.

[150] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, Kayur Patel. "Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks." In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 140–151. ISBN: 9781450375146. DOI: 10.1145/3379337.3415842.

[151] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, Kayur Patel. "mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks." In: *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*. Association for Computing Machinery, 2020.

[152] Mary Beth Kery, Donghao Ren, Kanit Wongsuphasawat, Fred Hohman, Kayur Patel. "The Future of Notebook Programming Is Fluid." In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3383085.

[153] Mary Beth Kery, Donghao Ren, Kanit Wongsuphasawat, Fred Hohman, Kayur Patel. "The Future of Notebook Programming Is Fluid." In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, Rafal Kocielnik. CHI EA '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3383085.

[154] Rex Bryan Kline, Ahmed Seffah. “Evaluation of integrated software development environments: Challenges and results from three empirical studies.” In: *Int. J. Hum. Comput. Stud.* 63.6 (2005), pp. 607–627. DOI: 10.1016/j.ijhcs.2005.05.002.

[155] Donald E. Knuth. “Literate Programming.” In: *The Computer Journal* 27.2 (Jan. 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97.

[156] Amy J. Ko, Htet Htet Aung, Brad A. Myers. “Design Requirements for More Flexible Structured Editors from a Study of Programmers’ Text Editing.” In: *CHI ’05 Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’05. Portland, OR, USA: Association for Computing Machinery, 2005, 1557–1560. ISBN: 1595930027. DOI: 10.1145/1056808.1056965.

[157] Sucheta V. Kolekar, Radhika M. Pai, M. M. Manohara Pai. “Rule based adaptive user interface for adaptive E-learning system.” In: *Educ. Inf. Technol.* 24.1 (2019), pp. 613–641. DOI: 10.1007/s10639-018-9788-1.

[158] Sanjay Krishnan, Eugene Wu. “PALM: Machine Learning Explanations For Iterative Debugging.” In: *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*. Ed. by Carsten Binnig, Joseph M. Hellerstein, Aditya G. Parameswaran. ACM, 2017, 4:1–4:6. DOI: 10.1145/3077257.3077271.

[159] Jesse Lane, Hak J. Kim. “Big data: web-crawling and analysing financial news using RapidMiner.” In: *Int. J. Bus. Inf. Syst.* 19.1 (2015), pp. 41–57. DOI: 10.1504/IJBIS.2015.069064.

[160] Pat Langley. “Machine Learning for Adaptive User Interfaces.” In: *KI-97: Advances in Artificial Intelligence, 21st Annual German Conference on Artificial Intelligence, Freiburg, Germany, September 9-12, 1997, Proceedings*. Vol. 1303. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 1997, pp. 53–62. DOI: 10.1007/3540634932_3.

[161] Sam Lau, Ian Drosos, Julia M. Markel, Philip J. Guo. “The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry.” In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. IEEE, 2020, pp. 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201.

[162] Sam Lau, Ian Drosos, Julia M. Markel, Philip J. Guo. “The design space of computational notebooks: An analysis of 60 systems in academia and industry.” In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. IEEE, 2020, pp. 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201.

[163] Yanhong Li, Nadine Bachl, Michelle Dutoit, Thomas Weber, Sven Mayer, Heinrich Hussmann. “Designing Tangible Tools to Engage Silent Students in Group Discussion.” In: *2022 International Symposium on Educational Technology (ISET)*. 2022, pp. 286–291. DOI: 10.1109/ISET55194.2022.00067.

[164] Yanhong Li, Szilvia Balogh, Armand Luttringer, Thomas Weber, Sven Mayer, Heinrich Hussmann. “Designing a Tangible Interface to “Force” Children Collaboration.” In: *Proceedings of the 21st Annual ACM Interaction Design and Children Conference*. IDC ’22. Braga, Portugal: Association for Computing Machinery, 2022, 576–582. ISBN: 9781450391979. DOI: 10.1145/3501712.3535280.

[165] Yanhong Li, Bill Bapisch, Jenny Phu, Thomas Weber, Heinrich Hußmann. “Study Marbles: A Wearable Light for Online Collaborative Learning in Video Meetings.” In: *Human-Computer Interaction – INTERACT 2021*. Ed. by Carmelo Ardito, Rosa Lanzilotti, Alessio Malizia, Helen Petrie, Antonio Piccinno, Giuseppe Desolda, Kori Inkpen. Cham: Springer International Publishing, 2021, pp. 712–721. ISBN: 978-3-030-85623-6.

[166] Yanhong Li, Aditi Kothiyal, Thomas Weber, Beat Rossmy, Sven Mayer, Heinrich Hussmann. “Designing Tangible as an Orchestration Tool for Collaborative Activities.” In: *Multimodal Technologies and Interaction* 6.5 (2022). ISSN: 2414-4088. DOI: 10.3390/mti6050030.

[167] Yanhong Li, Meng Liang, Julian Preissing, Nadine Bachl, Michelle Melina Dutoit, Thomas Weber, Sven Mayer, Heinrich Hussmann. “A Meta-Analysis of Tangible Learning Studies from the TEI Conference.” In: *Proceedings of the Sixteenth International Conference on Tangible, Embedded, and Embodied Interaction*. TEI ’22. Daejeon, Republic of Korea: Association for Computing Machinery, 2022. ISBN: 9781450391474. DOI: 10.1145/3490149.3501313.

[168] Yanhong Li, Yu Sun, Tianyang Lu, Thomas Weber, Heinrich Hußmann. “GrouPen: A Tangible User Interface to Support Remote Collaborative Learning.” In: *Human-Computer Interaction – INTERACT 2021*. Ed. by Carmelo Ardito, Rosa Lanzilotti,

Alessio Malizia, Helen Petrie, Antonio Piccinno, Giuseppe Desolda, Kori Inkpen. Cham: Springer International Publishing, 2021, pp. 590–598. ISBN: 978-3-030-85610-6.

[169] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, Oriol Vinyals. "Competition-level code generation with AlphaCode." In: *Science* 378.6624 (Dec. 2022), pp. 1092–1097. DOI: 10.1126/science.abq1158.

[170] Meng Liang, Yanhong Li, Thomas Weber, Heinrich Hussmann. "Tangible Interaction for Children's Creative Learning: A Review." In: *Proceedings of the 13th Conference on Creativity and Cognition*. C&C '21. Virtual Event, Italy: Association for Computing Machinery, 2021. ISBN: 9781450383769. DOI: 10.1145/3450741.3465262.

[171] Henry Lieberman, ed. *Your Wish is My Command*. The Morgan Kaufmann series in interactive technologies. Amsterdam, Netherlands: Morgan Kaufmann / Elsevier, 2001. ISBN: 978-1-55860-688-3. DOI: 10.1016/b978-1-55860-688-3.x5000-3.

[172] Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, Carole-Jean Wu. "GEVO-ML: a proposal for optimizing ML code with evolutionary computation." In: *GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume, Cancún, Mexico, July 8-12, 2020*. Ed. by Carlos Artemio Coello Coello. ACM, 2020, pp. 1849–1856. DOI: 10.1145/3377929.3398139.

[173] Eric S. Liu, Dylan A. Lukes, William G. Griswold. "Refactoring in Computational Notebooks." In: *ACM Trans. Softw. Eng. Methodol.* 32.3 (Apr. 2023). ISSN: 1049-331X. DOI: 10.1145/3576036.

[174] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, Lingming Zhang. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2023.

[175] Jiming Liu, Kelvin Chi Kuen Wong, Ka Keung Hui. "An Adaptive User Interface Based On Personalized Learning." In: *IEEE Intell. Syst.* 18.2 (2003), pp. 52–57. DOI: 10.1109/MIS.2003.1193657.

[176] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, Andrew D. Gordon. "“What It Wants Me To Say”: Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models." In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3580817.

[177] Shixia Liu, Jiannan Xiao, Junlin Liu, Xiting Wang, Jing Wu, Jun Zhu. "Visual Diagnosis of Tree Boosting Methods." In: *IEEE Trans. Vis. Comput. Graph.* 24.1 (2018), pp. 163–173. DOI: 10.1109/TVCG.2017.2744378.

[178] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, Mao Yang. "Enhancing the interoperability between deep learning frameworks by model conversion." In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, Thomas Zimmermann. ACM, 2020, pp. 1320–1330. DOI: 10.1145/3368089.3417051.

[179] André Luckow, Matthew Cook, Nathan Ashcraft, Edwin Weill, Emil Djerekarov, Bennie Vorster. "Deep learning in the automotive industry: Applications and tools." In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. IEEE Computer Society, 2016, pp. 3759–3768. DOI: 10.1109/BigData.2016.7841045.

[180] Ronald Mace. "Accessible Environments : Toward Universal Design." In: *Design Interventions : Toward A More Humane Architecture* 156 (1991), p. 2.

[181] Tambiama André Madiega. *AI investment: EU and global indicators*. https://www.europarl.europa.eu/RegData/etudes/ATAG/2024/760392/EPRS_ATA(2024)760392_EN.pdf. Mar. 2024.

[182] James S. Magnuson. "Moving hand reveals dynamics of thought." In: *Proceedings of the National Academy of Sciences* 102.29 (2005), pp. 9995–9996. ISSN: 0027-8424. DOI: 10.1073/pnas.0504413102.

[183] Vikas K. Malviya, Sawan Rai, Atul Gupta. “Development of a plugin based extensible feature extraction framework.” In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright, Richard Chbeir. ACM, 2018, pp. 1840–1847. DOI: 10.1145/3167132.3167328.

[184] Thomas J. McCabe. “A Complexity Measure (Abstract).” In: *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*. IEEE Computer Society, 1976, p. 407.

[185] John McCarthy. *What is AI? / Basic Questions*. http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html. Feb. 2012.

[186] Sofia Meacham, Vaclav Pech, Detlef D. Nauck. “Classification Algorithms Framework (CAF) to Enable Intelligent Systems Using JetBrains MPS Domain-Specific Languages Environment.” In: *IEEE Access* 8 (2020), pp. 14832–14840. DOI: 10.1109/ACCESS.2020.2966630.

[187] André N. Meyer, Thomas Fritz, Gail C. Murphy, Thomas Zimmermann. “Software Developers’ Perceptions of Productivity.” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 19–29. ISBN: 9781450330565. DOI: 10.1145/2635868.2635892.

[188] Nesrine Mezhoudi. “User Interface Adaptation Based on User Feedback and Machine Learning.” In: *Proceedings of the Companion Publication of the 2013 International Conference on Intelligent User Interfaces Companion*. Santa Monica, California, USA: Association for Computing Machinery, 2013.

[189] Marcin Michalak, Katarzyna Dusza, Dominik Korda, Krzysztof Kozlowski, Bartlomiej Szwej, Michal Kozielski, Marek Sikora, Lukasz Wróbel. “Application of RapidMiner and R Environments to Dangerous Seismic Events Prediction.” In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016*. Ed. by Maria Ganzha, Leszek A. Maciaszek, Marcin Paprzycki. Vol. 8. Annals of Computer Science and Information Systems. IEEE, 2016, pp. 249–252. DOI: 10.15439/2016F188.

[190] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, Timm Euler. "YALE: Rapid Prototyping for Complex Data Mining Tasks." In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 935–940. ISBN: 1595933395. DOI: 10.1145/1150402.1150531.

[191] Tim Miller, Piers Howe, Liz Sonenberg. "Explainable AI: Beware of Inmates Running the Asylum Or: How I Learnt to Stop Worrying and Love the Social and Behavioural Sciences." In: *CoRR* abs/1712.00547 (2017).

[192] Michael Milligan. "Interactive HPC Gateways with Jupyter and Jupyterhub." In: *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. PEARC '17. New Orleans, LA, USA: Association for Computing Machinery, 2017. ISBN: 9781450352727. DOI: 10.1145/3093338.3104159.

[193] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, Dan Roth. "Recent Advances in Natural Language Processing via Large Pre-trained Language Models: A Survey." In: *ACM Comput. Surv.* 56.2 (Sept. 2023). ISSN: 0360-0300. DOI: 10.1145/3605943.

[194] J. Mitchell, B. Shneiderman. "Dynamic versus static menus: an exploratory comparison." In: *SIGCHI Bull.* 20.4 (Apr. 1989), pp. 33–37. ISSN: 0736-6906. DOI: 10.1145/67243.67247.

[195] Armin Moin. "Data Analytics and Machine Learning Methods, Techniques and Tool for Model-Driven Engineering of Smart IoT Services." In: *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 287–292. DOI: 10.1109/ICSE-Companion52605.2021.00130.

[196] Hussein Mozannar, Gagan Bansal, Adam Fourney, Eric Horvitz. *Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming*. 2022. DOI: 10.48550/ARXIV.2210.14306.

[197] Hafiz Suliman Munawar. "Flood Disaster Management." In: *Machine Vision Inspection Systems*. John Wiley & Sons, Ltd, 2020. Chap. 5, pp. 115–146. ISBN: 9781119682042. DOI: https://doi.org/10.1002/9781119682042.ch5.

[198] Emerson Murphy-Hill, Rahul Jiresal, Gail C. Murphy. "Improving Software Developers' Fluency by Recommending Development Environment Commands." In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Cary, North Carolina: Association for Computing Machinery, 2012.

[199] G.C. Murphy, M. Kersten, L. Findlater. "How are Java software developers using the Eclipse IDE?" In: *IEEE Software* 23.4 (2006), pp. 76–83. DOI: 10.1109/MS.2006.105.

[200] Brad A. Myers. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '86. Boston, Massachusetts, USA: Association for Computing Machinery, 1986, 59–66. ISBN: 0897911806. DOI: 10.1145/22627.22349.

[201] Brad A. Myers. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Ed. by Marilyn M. Mantei, Peter Orbeton. CHI '86. Boston, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 59–66. ISBN: 0897911806. DOI: 10.1145/22627.22349.

[202] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, YoungSeok Yoon. "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools." In: *Computer* 49.7 (2016), pp. 44–52. DOI: 10.1109/MC.2016.200.

[203] Gopala Krishna Murthy N., V. S. N. Bhushana Rao, Naga Raju Orsu, Ramya Chiluvuri, Suresh B. Mudunuri. "Web-Weka1.0: Online Tool for Comparison of Classification Algorithms of Weka." In: *19th International Conference on Management of Data, COMAD 2013, Ahmedabad, India, December 19-21, 2013*. Ed. by Bipin V. Mehta, Nikos Mamoulis, Arnab Bhattacharya, Maya Ramanath. Computer Society of India, 2013, pp. 95–96.

[204] Gabor Nagy, Gergo Barta, Tamas Henk. "Portfolio optimization using local linear regression ensembles in RapidMiner." In: (2015).

[205] Jeanne Nakamura, Mihaly Csikszentmihalyi. "The Concept of Flow." In: *Flow and the Foundations of Positive Psychology: The Collected Works of Mihaly Csikszentmihalyi*. Dordrecht: Springer Netherlands, 2014, pp. 239–263. ISBN: 978-94-017-9088-8. DOI: 10.1007/978-94-017-9088-8_16.

[206] Varun Natu, Rahul Ghosh. “EasyDist: An End-to-End Distributed Deep Learning Tool for Cloud.” In: *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, COMAD/CODS 2019, Kolkata, India, January 3-5, 2019*. Ed. by Raghu Krishnapuram, Parag Singla. ACM, 2019, pp. 265–268. DOI: 10.1145/3297001.3297037.

[207] Adel Nehme. *How to Use ChatGPT Code Interpreter*. https://www.datacamp.com/tutorial/how-to-use-chat-gpt-code-interpreter. July 2023.

[208] Nhan Nguyen, Sarah Nadi. “An empirical evaluation of GitHub copilot's code suggestions.” In: *Proceedings of the 19th International Conference on Mining Software Repositories*. ACM, May 2022. DOI: 10.1145/3524842.3528470.

[209] Jakob Nielsen. *F-Shaped Pattern For Reading Web Content*. https://www.nngroup.com/articles/f-shaped-pattern-reading-web-content-discovered/. 2006.

[210] Aquilas Tchanjou Njomou, Alexandra Johanne Bifona Africa, Bram Adams, Marios Fokaefs. “MSR4ML: Reconstructing Artifact Traceability in Machine Learning Repositories.” In: *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 536–540. DOI: 10.1109/SANER50967.2021.00061.

[211] Mark Noone, Aidan Mooney. “Visual and textual programming languages: a systematic review of the literature.” In: *Journal of Computers in Education* 5 (2018), pp. 149–174.

[212] Keith J. O’Hara, Douglas S. Blank, James B. Marshall. “Computational Notebooks for AI Education.” In: *Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference*. Ed. by Ingrid Russell, William Eberle. FLAIRS. Hollywood, Florida, USA: AAAI Press, 2015, pp. 263–268. DOI: 10.13140/2.1.2434.5928.

[213] Edson Oliveira, Eduardo Fernandes, Igor Steinmacher, Marco Cristo, Tayana Conte, Alessandro Garcia. “Code and commit metrics of developer productivity: a study on team leaders perceptions.” In: *Empirical Software Engineering* 25.4 (Apr. 2020), pp. 2519–2549. DOI: 10.1007/s10664-020-09820-z.

[214] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, Jason H. Moore. “Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science.” In:

Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016. Ed. by Tobias Friedrich, Frank Neumann, Andrew M. Sutton. ACM, 2016, pp. 485–492. DOI: 10.1145/2908812.2908918.

[215] P. Oman, J. Hagemeister. "Metrics for assessing a software system's maintainability." In: *Proceedings Conference on Software Maintenance 1992*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 1992, pp. 337,338,339,340,341,342,343,344. DOI: 10.1109/ICSM.1992.242525.

[216] Paul Oman, Jack Hagemeister, Dan Ash. "A definition and taxonomy for software maintainability." In: *Moscow, ID, USA, Tech. Rep* (1992), pp. 91–08.

[217] Erol Ozan. "A Novel Browser-based No-code Machine Learning Application Development Tool." In: *2021 IEEE World AI IoT Congress (AIIoT)*. 2021, pp. 0282–0284. DOI: 10.1109/AIIoT52608.2021.9454239.

[218] Szilárd Pafka. "Machine Learning Software in Practice: Quo Vadis?" In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 2017, p. 25. DOI: 10.1145/3097983.3106683.

[219] Matthew J. Page, Joanne E. McKenzie, Patrick M. Bossuyt, Isabelle Boutron, Tammy C. Hoffmann, Cynthia D. Mulrow, Larissa Shamseer, Jennifer M. Tetzlaff, Elie A. Akl, Sue E. Brennan, Roger Chou, Julie Glanville, Jeremy M. Grimshaw, Asbjørn Hróbjartsson, Manoj M. Lalu, Tianjing Li, Elizabeth W. Loder, Evan Mayo-Wilson, Steve McDonald, Luke A. McGuinness, Lesley A. Stewart, James Thomas, Andrea C. Tricco, Vivian A. Welch, Penny Whiting, David Moher. "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews." English (US). In: *Journal of Chronic Diseases* 134 (June 2021), pp. 178–189. ISSN: 0895-4356. DOI: 10.1016/j.jclinepi.2021.03.001.

[220] Stephanie Palazzolo, Erin Woo, Amir Efrati. "OpenAI Shifts Strategy as Rate of 'GPT' AI Improvements Slows." In: (Nov. 2024). https://www.theinformation.com/articles/openai-shifts-strategy-as-rate-of-gpt-ai-improvements-slows.

[221] Ki Sun Park, Kyoung-Soon Hwang, Keon Myung Lee, Chan Sik Han, Jin Han. "Machine learning modeling assistance for non-expert developers." In: *Proceedings of the Conference on Research in Adaptive and Convergent Systems, RACS*

2019, Chongqing, China, September 24-27, 2019. Ed. by Chih-Cheng Hung, Qianbin Chen, Xianzhong Xie, Christian Esposito, Jun Huang, Juw Won Park, Qinghua Zhang. ACM, 2019, pp. 117–122. DOI: 10.1145/3338840.3355680.

[222] Kayur Patel. “Lowering the Barrier to Applying Machine Learning.” PhD thesis. University of Washington, USA, 2012.

[223] Kayur Patel, James Fogarty, James A. Landay, Beverly L. Harrison. “Examining Difficulties Software Developers Encounter in the Adoption of Statistical Machine Learning.” In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. AAAI Press, 2008, pp. 1563–1566.

[224] Urja Pawar, Donna O’Shea, Susan Rea, Ruairi O’Reilly. “Explainable AI in Healthcare.” In: *2020 International Conference on Cyber Situational Awareness, Data Analytics and Assessment, CyberSA 2020, Dublin, Ireland, June 15-19, 2020*. IEEE, 2020, pp. 1–2. DOI: 10.1109/CyberSA49311.2020.9139655.

[225] Patrick Peachock, Nicholas Iovino, Bonita Sharif. “Investigating eye movements in natural language and c++ source code-a replication experiment.” In: *International Conference on Augmented Cognition*. Springer. 2017, pp. 206–218.

[226] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. “Scikit-learn: Machine learning in Python.” In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[227] Matthias Peissner, Rob Edlin-White. “User Control in Adaptive User Interfaces for Accessibility.” In: *Human-Computer Interaction – INTERACT 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 623–640. ISBN: 978-3-642-40483-2. DOI: 10.1007/978-3-642-40483-2_44.

[228] Norman Peitek, Janet Siegmund, Sven Apel. “What Drives the Reading Order of Programmers? An Eye Tracking Study.” In: *Proceedings of the 28th International Conference on Program Comprehension*. Seoul, Republic of Korea: Association for Computing Machinery, 2020.

[229] Sida Peng, Eirini Kalliamvakou, Peter Cihon, Mert Demirer. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. 2023. DOI: 10.48550/ARXIV.2302.06590.

[230] Jeffrey M. Perkel. “Why Jupyter is data scientists’ computational notebook of choice.” In: *Nature* (563 Nov. 2018), pp. 145–146. DOI: 10.1038/d41586-018-07196-1.

[231] Kai Petersen. “Measuring and predicting software productivity: A systematic map and review.” In: *Information and Software Technology* 53.4 (Apr. 2011), pp. 317–343. DOI: 10.1016/j.infsof.2010.12.001.

[232] Ken Pfeuffer, Yasmeen Abdrabou, Augusto Esteves, Radiah Rivu, Yomna Abdelrahman, Stefanie Meitner, Amr Saadi, Florian Alt. “ARtention: A design space for gaze-adaptive user interfaces in augmented reality.” In: *Comput. Graph.* 95 (2021), pp. 1–12. DOI: 10.1016/j.cag.2021.01.001.

[233] Ken Pfeuffer, Jason Alexander, Ming Ki Chong, Yanxia Zhang, Hans Gellersen. “Gaze-Shifting: Direct-Indirect Input with Pen and Touch Modulated by Gaze.” In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST ’15. Charlotte, NC, USA: Association for Computing Machinery, 2015, pp. 373–383. ISBN: 9781450337793. DOI: 10.1145/2807442.2807460.

[234] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, Juliana Freire. “A Large-Scale Study about Quality and Reproducibility of Jupyter Notebooks.” In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR ’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 507–517. DOI: 10.1109/MSR.2019.00077.

[235] Haoyue Ping, Julia Stoyanovich, Bill Howe. “DataSynthesizer: Privacy-Preserving Synthetic Datasets.” In: *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. ACM, 2017, 42:1–42:5. DOI: 10.1145/3085504.3091117.

[236] Rohit Prabhavalkar, Takaaki Hori, Tara N. Sainath, Ralf Schlüter, Shinji Watanabe. “End-to-End Speech Recognition: A Survey.” In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 32 (2024), pp. 325–351. DOI: 10.1109/TASLP.2023.3328283.

[237] Alun Preece, Dan Harborne, Dave Braines, Richard Tomsett, Supriyo Chakraborty. *Stakeholders in Explainable AI*. 2018.

[238] Sebastian Proksch, Johannes Lerch, Mira Mezini. “Intelligent Code Completion with Bayesian Networks.” In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (Dec. 2015). ISSN: 1049-331X. DOI: 10.1145/2744200.

[239] Basit Qureshi. *Exploring the Use of ChatGPT as a Tool for Learning and Assessment in Undergraduate Computer Science Curriculum: Opportunities and Challenges*. 2023. DOI: 10.48550/ARXIV.2304.11214.

[240] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever. “Improving language understanding by generative pre-training.” In: (2018).

[241] Alvin Rajkomar, Jeffrey Dean, Isaac Kohane. “Machine Learning in Medicine.” In: *New England Journal of Medicine* 380.14 (2019). https://www.nejm.org/doi/full/10.1056/NEJMra1814259, pp. 1347–1358. DOI: 10.1056/NEJMra1814259.

[242] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, Christine L. Borgman. “Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study.” In: *Proceedings of the 17th ACM/IEEE Joint Conference on Digital Libraries*. JCDL ’17. Toronto, Ontario, Canada: IEEE Press, 2017, pp. 338–339. ISBN: 9781538638613. DOI: 10.1109/JCDL.2017.7991618.

[243] Veselin Raychev, Martin Vechev, Eran Yahav. “Code Completion with Statistical Language Models.” In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 419–428. ISBN: 9781450327848. DOI: 10.1145/2594291.2594321.

[244] Mireia Ribera, Àgata Lapedriza. “Can we do better explanations? A proposal of user-centered explainable AI.” In: *Joint Proceedings of the ACM IUI 2019 Workshops co-located with the 24th ACM Conference on Intelligent User Interfaces (ACM) IUI 2019)*. Vol. 2327. CEUR Workshop Proceedings. CEUR-WS.org, 2019.

[245] E. Riccobene, P. Scandurra, A. Rosti, S. Bocchio. “A Model-Driven Design Environment for Embedded Systems.” In: *Proceedings of the 43rd Annual Design Automation Conference*. DAC ’06. San Francisco, CA, USA: Association for Computing Machinery, 2006, pp. 915–918. ISBN: 1595933816. DOI: 10.1145/1146909.1147141.

[246] Devon Rifkin, Terkel Gjervig Nielsen, Cole Bemis, Alice Li. *GitHub Spark*. https://githubnext.com/projects/github-spark. 2024.

[247] Oliver Ritthoff, Ralf Klinkenberg, Simon Fischer, Ingo Mierswa, Sven Felske. *Yale: Yet Another Learning Environment*. 2001.

[248] Oliver Ritthoff, Ralf Klinkenberg, Simon Fischer, Ingo Mierswa, Sven Felske. *Yale: Yet Another Learning Environment*. 2001.

[249] Romain Robbes, Michele Lanza. "How Program History Can Improve Code Completion." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 317–326. DOI: 10.1109/ASE.2008.42.

[250] Romain Robbes, Michele Lanza. "Improving code completion with program history." In: *Autom. Softw. Eng.* 17.2 (2010), pp. 181–212. DOI: 10.1007/s10515-010-0064-x.

[251] M Robillard, R Walker, T Zimmermann. "Recommendation Systems for Software Engineering." In: *IEEE Softw.* 27.4 (July 2010), pp. 80–86.

[252] Martin P. Robillard, Robert J. Walker, Thomas Zimmermann. "Recommendation Systems for Software Engineering." In: *IEEE Softw.* 27.4 (2010), pp. 80–86. DOI: 10.1109/MS.2009.161.

[253] William Robinson. "From Scratch to Patch: Easing the Blocks-Text Transition." In: *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. WiPSCE '16. Association for Computing Machinery, 2016, pp. 96–99. ISBN: 9781450342230. DOI: 10.1145/2978249.2978265.

[254] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, Sidney K. D'Mello. "Improving automated source code summarization via an eye-tracking study of programmers." In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, André van der Hoek. ICSE '14. Hyderabad, India: ACM, 2014, pp. 390–401. DOI: 10.1145/2568225.2568247.

[255] Mario Rodriguez. "New from Universe 2024: Get the latest previews and releases." In: (Oct. 2024). https://github.blog/news-insights/product-news/universe-2024-previews-releases/.

[256] Mario Rodriguez. "Research: Quantifying GitHub Copilot's impact on code quality." https://github.blog/2023-10-10-research-quantifying-github-copilots-impact-on-code-quality/. 2023.

[257] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, Justin D. Weisz. “The Programmer’s Assistant: Conversational Interaction with a Large Language Model for Software Development.” In: *Proceedings of the 28th International Conference on Intelligent User Interfaces*. IUI ’23. Sydney, NSW, Australia: Association for Computing Machinery, 2023, pp. 491–514. ISBN: 9798400701061. DOI: 10.1145/3581641.3584037.

[258] Jean Michel Rouly, Jonathan D. Orbeck, Eugene Syriani. “Usability and Suitability Survey of Features in Visual Ides for Non-Programmers.” In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. Portland, Oregon, USA: Association for Computing Machinery, 2014.

[259] James Rumbaugh, Ivar Jacobson, Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. London, UK: Pearson Higher Education, 2004. ISBN: 0321245628.

[260] Mazeiar Salehie, Ladan Tahvildari. “Self-adaptive software: Landscape and research challenges.” In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (2009), 14:1–14:42. DOI: 10.1145/1516533.1516538.

[261] Manuel Martin Salvador, Marcin Budka, Bogdan Gabrys. “Automatic Composition and Optimization of Multicomponent Predictive Systems With an Extended Auto-WEKA.” In: *IEEE Trans Autom. Sci. Eng.* 16.2 (2019), pp. 946–959. DOI: 10.1109/TASE.2018.2876430.

[262] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, Brendan Dolan-Gavitt. *Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants*. 2022. DOI: 10.48550/ARXIV.2208.09727.

[263] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, Ben Zorn. *What is it like to program with artificial intelligence?* 2022. DOI: 10.48550/ARXIV.2208.06213.

[264] Christian Scheidler, Lorenz Schäfers, Ottmar Krämer-Fuhrmann. “TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications.” In: *Parallel Architectures and Languages Europe, 5th International PARLE Conference*. Ed. by Arndt Bode, Mike Reeve, Gottfried Wolf. Vol. 694. PARLE ’93. Munich, Germany: Springer, 1993, pp. 403–413. DOI: 10.1007/3-540-56891-3_32.

[265] Eldon Schoop, Forrest Huang, Björn Hartmann. “SCRAM: Simple Checks for Realtime Analysis of Model Training for Non-Expert ML Programmers.” In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt, Florian ’Floyd’ Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, Rafal Kocielnik. ACM, 2020, pp. 1–10. DOI: 10.1145/3334480.3382879.

[266] Max Schäfer, Sarah Nadi, Aryaz Eghbali, Frank Tip. “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation.” In: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 85–105. DOI: 10.1109/TSE.2023.3334955.

[267] Alex Serban, Erik Poll, Joost Visser. “Adversarial Examples on Object Recognition: A Comprehensive Survey.” In: *ACM Comput. Surv.* 53.3 (June 2020). ISSN: 0360-0300. DOI: 10.1145/3398394.

[268] Arash Shaban-Nejad, Martin Michalowski, John S. Brownstein, David L. Buckeridge. “Guest Editorial Explainable AI: Towards Fairness, Accountability, Transparency and Trust in Healthcare.” In: *IEEE J. Biomed. Health Informatics* 25.7 (2021), pp. 2374–2375. DOI: 10.1109/JBHI.2021.3088832.

[269] Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, Bonita Sharif. “ITrace: Enabling Eye Tracking on Software Artifacts within the IDE to Support Software Engineering Tasks.” In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy: Association for Computing Machinery, 2015.

[270] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, Tim Kraska. “Democratizing Data Science through Interactive Curation of ML Pipelines.” In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, Tim Kraska. ACM, 2019, pp. 1171–1188. DOI: 10.1145/3299869.3319863.

[271] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, Yann-Gaël Guéhéneuc. “Eye-tracking metrics in software engineering.” In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 96–103.

[272] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc. “A systematic literature review on the usage of eye-tracking in software engineering.” In: *Inf. Softw. Technol.* 67 (2015), pp. 79–107. DOI: `10.1016/j.infsof.2015.06.008`.

[273] Kshitij Sharma, Patrick Jermann, Marc-Antoine Nüssli, Pierre Dillenbourg. “Understanding Collaborative Program Comprehension: Interlacing Gaze and Dialogues.” In: *10th International Conference on Computer-Supported Collaborative Learning, CSCL 2013, Conference Proceedings, Volume 1: Full Papers & Symposia, June 15-19, 2013, Madison, WI, USA*. International Society of the Learning Sciences, LuLu, Amazon, 2013, pp. 430–437.

[274] Shubham Sharma, Jette Henderson, Joydeep Ghosh. “CERTIFAI: A Common Framework to Provide Explanations and Analyse the Fairness and Robustness of Black-box Models.” In: *AIES ’20: AAAI/ACM Conference on AI, Ethics, and Society, New York, NY, USA, February 7-8, 2020*. Ed. by Annette N. Markham, Julia Powles, Toby Walsh, Anne L. Washington. ACM, 2020, pp. 166–172. DOI: `10.1145/3375627.3375812`.

[275] Helen Sharp, Jenny Preece, Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2019. ISBN: 9781119547358.

[276] Dvijesh J. Shastri. “Machine Learning for Non-Programmers.” In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt, Florian ’Floyd’ Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, Rafal Kocielnik. ACM, 2020, pp. 1–3. DOI: `10.1145/3334480.3375051`.

[277] Mohammad Shehab, Laith Abualigah, Qusai Shambour, Muhannad A. Abu-Hashem, Mohd Khaled Yousef Shambour, Ahmed Izzat Alsalibi, Amir H. Gandomi. “Machine learning in medical applications: A review of state-of-the-art methods.” In: *Computers in Biology and Medicine* 145 (2022), p. 105458. ISSN: 0010-4825. DOI: `https://doi.org/10.1016/j.compbiomed.2022.105458`.

[278] Helen Shen. “Interactive notebooks: Sharing the code.” In: *Nature* 515.7525 (Nov. 2014), pp. 151–152.

[279] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, Yarin Gal. “AI models collapse when trained on recursively generated data.” In: *Nature* 631.8022 (July 2024), pp. 755–759. ISSN: 1476-4687. DOI: 10.1038/s41586-024-07566-y.

[280] Yasin N. Silva, Anthony Nieuwenhuyse, Thomas G. Schenk, Alaura Symons. “DBSnap++: creating data-driven programs by snapping blocks.” In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018, Larnaca, Cyprus, July 02-04, 2018*. Ed. by Irene Polycarpou, Janet C. Read, Panayiotis Andreou, Michal Armoni. ACM, 2018, pp. 170–175. DOI: 10.1145/3197091.3197114.

[281] Andrew J. Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, Rajesh Vasa. “A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects.” In: *ESEM ’20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*. Association for Computing Machinery, 2020.

[282] Daniel J. Simons, Daniel T. Levin. “Change blindness.” In: *Trends in Cognitive Sciences* 1.7 (Oct. 1997), pp. 261–267. ISSN: 1364-6613. DOI: 10.1016/S1364-6613(97)01080-2.

[283] Dominik Sobania, Martin Briesch, Franz Rothlauf. “Choose your programming copilot.” In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO ’22. Boston, Massachusetts: ACM, July 2022, pp. 1019–1027. ISBN: 9781450392372. DOI: 10.1145/3512290.3528700.

[284] Akhila Sri Manasa Venigalla, Sridhar Chimalakonda. “What’s in a GitHub Repository? – A Software Documentation Perspective.” In: *arXiv e-prints* (Feb. 2021). DOI: 10.48550/arXiv.2102.12727.

[285] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A. Heymann, Massimiliano Di Penta, Daniel M German, Denys Poshyvanyk. *Developer Perspectives on Licensing and Copyright Issues Arising from Generative AI for Coding*. 2024.

[286] Constantine Stephanidis. “Towards the Next Generation of UIST: Developing for All Users.” In: *Proceedings of the Seventh International Conference on Human-Computer Interaction-Volume 1 - Volume I*. HCI International ’97. USA: Elsevier Science Inc., 1997, pp. 473–476. ISBN: 044482183X.

[287] Margaret-Anne Storey, Thomas Zimmermann, Christian Bird, Jacek Czerwonka, Brendan Murphy, Eirini Kalliamvakou. “Towards a Theory of Software Developer Job Satisfaction and Perceived Productivity.” In: *IEEE Transactions on Software Engineering* 47.10 (Oct. 2021), pp. 2125–2142. DOI: 10.1109/tse.2019.2944354.

[288] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, Alois Zoitl, Christoph Sunder, Antonio Valentini, Allan Martel. “Framework for Distributed Industrial Automation and Control (4DIAC).” In: *2008 6th IEEE International Conference on Industrial Informatics*. IEEE, 2008, pp. 283–288. DOI: 10.1109/INDIN.2008.4618110.

[289] Liessman Sturlaugson, Nathan Fortier, Patrick Donnelly, John W. Sheppard. “Implementing AI-ESTATE with prognostic extensions in Java.” In: *2013 IEEE AUTOTESTCON*. 2013, pp. 1–8. DOI: 10.1109/AUTEST.2013.6645086.

[290] Yunjia Sun, Edward Lank, Michael A. Terry. “Label-and-Learn: Visualizing the Likelihood of Machine Learning Classifier’s Success During Data Labeling.” In: *Proceedings of the 22nd International Conference on Intelligent User Interfaces, IUI 2017, Limassol, Cyprus, March 13-16, 2017*. Ed. by George A. Papadopoulos, Tsvi Kuflik, Fang Chen, Carlos Duarte, Wai-Tat Fu. ACM, 2017, pp. 523–534. DOI: 10.1145/3025171.3025208.

[291] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, Neel Sundaresan. “Pythia: AI-Assisted Code Completion System.” In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 2727–2735. ISBN: 9781450362016. DOI: 10.1145/3292500.3330699.

[292] E. G. Sysoletin, Konstantin A. Aksyonov, X. D. Nisskhen, Olga P. Aksyonova. “Development of Components of Multi-agent CASE-System for Describing the Logic of Behavior of Mobile Robots.” In: *2017 European Modelling Symposium (EMS), Manchester, United Kingdom, survey = "•", November 20-21, 2017*. IEEE, 2017, pp. 3–8. DOI: 10.1109/EMS.2017.12.

[293] Attila Szatmári. “Towards Context-Aware Spectrum-Based Fault Localization.” In: (2023), pp. 496–498. DOI: 10.1109/ICST57152.2023.00060.

[294] Andrew Tarantola. "ChatGPT's latest model may be a regression in performance." In: (Nov. 2024). https://www.digitaltrends.com/computing/analysis-suggests-openai-flagship-model-only-performs-like-the-mini-version/.

[295] Kashyap Todi, Gilles Bailly, Luis Leiva, Antti Oulasvirta. "Adapting User Interfaces with Model-Based Reinforcement Learning." In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama, Japan: Association for Computing Machinery, 2021.

[296] Anna Trudova, Michal Dolezel, Alena Buchalcevová. "Artificial Intelligence in Software Test Automation: A Systematic Literature Review." In: *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020*. Ed. by Raian Ali, Hermann Kaindl, Leszek A. Maciaszek. Prague, Czech Republic: SCITEPRESS, 2020, pp. 181–192. DOI: 10.5220/0009417801810192.

[297] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, Neel Sundaresan. *Unit Test Case Generation with Transformers and Focal Context*. 2020. DOI: 10.48550/ARXIV.2009.05617.

[298] Priyan Vaithilingam, Tianyi Zhang, Elena L. Glassman. "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models." In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391566. DOI: 10.1145/3491101.3519665.

[299] Ville Vakkuri, Kai-Kristian Kemell, Pekka Abrahamsson. "Technical Briefing: Hands-On Session on the Development of Trustworthy AI Software." In: *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 332–333. DOI: 10.1109/ICSE-Companion52605.2021.00142.

[300] Joaquin Vanschoren. "Meta-Learning." In: *Automated Machine Learning - Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, Joaquin Vanschoren. The Springer Series on Challenges in Machine Learning. Springer, 2019, pp. 35–61. DOI: 10.1007/978-3-030-05318-5_2.

[301] G. Varoquaux, L. Buitinck, G. Louppe, O. Grisel, F. Pedregosa, A. Mueller. “Scikit-Learn: Machine Learning Without Learning the Machinery.” In: *GetMobile: Mobile Comp. and Comm.* 19.1 (June 2015), 29–33. ISSN: 2375-0529. DOI: 10.1145/2786984.2786995.

[302] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762.

[303] Gregory Vial, Bogdan Negoita. “Teaching Programming to Non-Programmers: The Case of Python and Jupyter Notebooks.” In: *Proceedings of the International Conference on Information Systems - Bridging the Internet of People, Data, and Things*. Ed. by Jan Pries-Heje, Sudha Ram, Michael Rosemann. ICIS. San Francisco, CA, USA: Association for Information Systems, 2018, p. 17.

[304] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Marius Hobbhahn. “Position: Will we run out of data? Limits of LLM scaling based on human-generated data.” In: *Forty-first International Conference on Machine Learning*. 2024.

[305] April Yi Wang, Anant Mittal, Christopher Brooks, Steve Oney. “How Data Scientists Use Computational Notebooks for Real-Time Collaboration.” In: *Proc. ACM Hum.-Comput. Interact.* 3.CSCW (Nov. 2019). DOI: 10.1145/3359141.

[306] Chenyu Wang, Zhou Yang, Ze Shi Li, Daniela Damian, David Lo. *Quality Assurance for Artificial Intelligence: A Study of Industrial Concerns, Challenges and Best Practices*. 2024.

[307] Jiawei Wang, Li Li, Andreas Zeller. “Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks.” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 53–56. ISBN: 9781450371261. DOI: 10.1145/3377816.3381724.

[308] Junfeng Wang, Andrew Hogue. “CVNodes: A Visual Programming Paradigm for Developing Computer Vision Algorithms.” In: *17th Conference on Computer and Robot Vision, CRV 2020, Ottawa, ON, Canada, May 13-15, 2020*. IEEE, 2020, pp. 174–181. DOI: 10.1109/CRV50864.2020.00031.

[309] Pei Wang, Weiling Zheng, Jiannan Wang, Jian Pei. “Automating Entity Matching Model Development.” In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1296–1307. DOI: 10.1109/ICDE51399.2021.00116.

[310] Qianwen Wang, Yao Ming, Zhihua Jin, Qiaomu Shen, Dongyu Liu, Micah J. Smith, Kalyan Veeramachaneni, Huamin Qu. “ATMSeer: Increasing Transparency and Controllability in Automated Machine Learning.” In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*. Ed. by Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos. ACM, 2019, p. 681. DOI: 10.1145/3290605.3300911.

[311] Alex Watson, Scott Bateman, Suprio Ray. “PySnippet: Accelerating Exploratory Data Analysis in Jupyter Notebook through Facilitated Access to Example Code.” In: *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference*. EDBT/ICDT 2019. Lisbon, Portugal: CEUR-WS, 2019, p. 4.

[312] Thomas Weber, Heinrich Hußmann, Malin Eiband. “Quantifying the Demand for Explainability.” In: *Human-Computer Interaction - INTERACT 2021 - 18th IFIP TC 13 International Conference, Bari, Italy, August 30 - September 3, 2021, Proceedings, Part II*. Ed. by Carmelo Ardito, Rosa Lanzilotti, Alessio Malizia, Helen Petrie, Antonio Piccinno, Giuseppe Desolda, Kori Inkpen. Vol. 12933. Lecture Notes in Computer Science. Springer, 2021, pp. 652–661. DOI: 10.1007/978-3-030-85616-8_38.

[313] Thomas Weber, Heinrich Hußmann, Zhiwei Han, Stefan Matthes, Yuanting Liu. “Draw with me: human-in-the-loop for image restoration.” In: *Proceedings of the 25th International Conference on Intelligent User Interfaces*. IUI ’20. Cagliari, Italy: Association for Computing Machinery, 2020, 243–253. ISBN: 9781450371186. DOI: 10.1145/3377325.3377509.

[314] Thomas Weber, Sven Mayer. “From Computational to Conversational Notebooks.” In: *Proceedings of the 1st CHI Workshop on Human-Notebook Interactions*. Honolulu, Hawaii, USA: Online, 2024.

[315] Thomas Weber, Alois Zoitl, Heinrich Hußmann. "Usability of Development Tools: A CASE-Study." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 228–235. DOI: 10.1109/MODELS-C.2019.00037.

[316] Nathaniel Weinman, Steven M. Drucker, Titus Barik, Robert DeLine. "Fork It: Supporting Stateful Alternatives in Computational Notebooks." In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445527.

[317] David Weintrop. "Block-Based Programming in Computer Science Education." In: *Commun. ACM* 62.8 (July 2019), pp. 22–25. ISSN: 0001-0782. DOI: 10.1145/3341221.

[318] David Weintrop, Uri Wilensky. "Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments." In: *Proceedings of the 2017 Conference on Interaction Design and Children*. IDC '17. Association for Computing Machinery, 2017, pp. 183–192. ISBN: 9781450349215. DOI: 10.1145/3078072.3079715.

[319] Daniel S. Weld, Corin Anderson, Pedro Domingos, Oren Etzioni, Krzysztof Gajos, Tessa Lau, Steve Wolfman. "Automatically Personalizing User Interfaces." In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. IJCAI'03. Acapulco, Mexico: Morgan Kaufmann Publishers Inc., 2003, pp. 1613–1619.

[320] Kurt D Welker. "The software maintainability index revisited." In: *CrossTalk* 14 (2001), pp. 18–21.

[321] James Wexler, Mahima Pushkarna, Tolga Bolukbasi, Martin Wattenberg, Fernanda B. Viégas, Jimbo Wilson. "The What-If Tool: Interactive Probing of Machine Learning Models." In: *IEEE Trans. Vis. Comput. Graph.* 26.1 (2020), pp. 56–65. DOI: 10.1109/TVCG.2019.2934619.

[322] Gesa Wiegand, Matthias Schmidmaier, Thomas Weber, Yuanting Liu, Heinrich Hussmann. "I Drive - You Trust: Explaining Driving Behavior Of Autonomous Cars." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in*

Computing Systems. CHI EA '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, 1–6. ISBN: 9781450359719. DOI: 10.1145/3290607.3312817.

[323] Jeannette M. Wing. "Computational thinking." In: *Commun. ACM* 49.3 (2006), pp. 33–35. DOI: 10.1145/1118178.1118215.

[324] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, Katsuro Inoue. "How do developers utilize source code from stack overflow?" In: *Empirical Software Engineering* 24.2 (Apr. 2019), pp. 637–673. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9634-5.

[325] Doris Xin, Eva Yiwei Wu, Doris Jung Lin Lee, Niloufar Salehi, Aditya G. Parameswaran. "Whither AutoML? Understanding the Role of Automation in Machine Learning Workflows." In: *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*. Ed. by Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn, Steven Mark Drucker. ACM, 2021, 83:1–83:16. DOI: 10.1145/3411764.3445306.

[326] Frank F. Xu, Uri Alon, Graham Neubig, Vincent Josua Hellendoorn. "A Systematic Evaluation of Large Language Models of Code." In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 1–10. ISBN: 9781450392730. DOI: 10.1145/3520312.3534862.

[327] Frank F. Xu, Bogdan Vasilescu, Graham Neubig. "In-IDE Code Generation from Natural Language: Promise and Challenges." In: *ACM Transactions on Software Engineering and Methodology* 31.2 (Mar. 2022), pp. 1–47. DOI: 10.1145/3487569.

[328] Weiwei Xu, Kai Gao, Hao He, Minghui Zhou. *LiCoEval: Evaluating LLMs on License Compliance in Code Generation*. 2024.

[329] Zhou Yang, David Lo. *Hotfixing Large Language Models for Code*. 2024.

[330] Burak Yetistiren, Isik Ozsoy, Eray Tuzun. "Assessing the quality of GitHub copilot's code generation." In: *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2022. Singapore, Singapore: ACM, Nov. 2022, pp. 62–71. ISBN: 9781450398602. DOI: 10.1145/3558489.3559072.

[331] Enes Yigitbas, Stefan Sauer, Gregor Engels. “Adapt-UI: An IDE Supporting Model-Driven Development of Self-Adaptive UIs.” In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. Lisbon, Portugal: Association for Computing Machinery, 2017.

[332] Enes Yigitbas, Hagen Stahl, Stefan Sauer, Gregor Engels. “Self-adaptive UIs: Integrated Model-Driven Development of UIs and Their Adaptations.” In: *Modelling Foundations and Applications*. Ed. by Anthony Anjorin, Huáscar Espinoza. Cham: Springer International Publishing, 2017, pp. 126–141. ISBN: 978-3-319-61482-3.

[333] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, Charles Sutton. *Natural Language to Code Generation in Interactive Data Science Notebooks*. 2022.

[334] Jun Yuan, Changjian Chen, Weikai Yang, Mengchen Liu, Jiazhi Xia, Shixia Liu. “A survey of visual analytics techniques for machine learning.” In: *Comput. Vis. Media* 7.1 (2021), pp. 3–36. DOI: 10.1007/s41095-020-0191-7.

[335] Guanhua Zhang, Susanne Hindennach, Jan Leusmann, Felix Bühler, Benedict Steuerlein, Sven Mayer, Mihai Bâce, Andreas Bulling. “Predicting Next Actions and Latent Intents during Text Formatting.” In: *Proceedings of the CHI Workshop Computational Approaches for Understanding, Generating, and Adapting User Interfaces*. Stuttgart, Germany: University of Stuttgart, 2022, pp. 1–6.

[336] Qishen Zhang, Tamás Kecskés, Janos L. Mathe, Janos Sztipanovits. “Towards bridging the gap between model- and data- driven tool suites for cyber-physical systems.” In: *Proceedings of the 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS@ICSE 2019, Montreal, QC, Canada, May 28, 2019*. Ed. by Tomás Bures, Bradley R. Schmerl, John S. Fitzgerald, Danny Weyns. IEEE / ACM, 2019, pp. 7–13. DOI: 10.1109/SEsCPS.2019.00009.

[337] Shuai Zhang, Lina Yao, Aixin Sun, Yi Tay. “Deep Learning Based Recommender System: A Survey and New Perspectives.” In: *ACM Comput. Surv.* 52.1 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3285029.

[338] Xufan Zhang, Ziyue Yin, Yang Feng, Qingkai Shi, Jia Liu, Zhenyu Chen. “NeuralVis: Visualizing and Interpreting Deep Learning Models.” In: *34th IEEE/ACM*

International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE, 2019, pp. 1106–1109. DOI: 10.1109/ASE.2019.00113.

[339] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, Takeo Igarashi. "ODEN: Live Programming for Neural Network Architecture Editing." In: *IUI 2022: 27th International Conference on Intelligent User Interfaces, Helsinki, Finland, March 22 - 25, 2022*. Ed. by Giulio Jacucci, Samuel Kaski, Cristina Conati, Simone Stumpf, Tuukka Ruotsalo, Krzysztof Gajos. ACM, 2022, pp. 392–404. DOI: 10.1145/3490099.3511120.

[340] Shuyin Zhao. *GitHub Copilot Chat beta now available for all individuals*. https://github.blog/2023-09-20-github-copilot-chat-beta-now-available-for-all-individuals/. 2023.

[341] Nan Zheng, Aaron Paloski, Haining Wang. "An Efficient User Verification System Using Angle-Based Mouse Movement Biometrics." In: *ACM Trans. Inf. Syst. Secur.* 18.3 (2016), 11:1–11:27. DOI: 10.1145/2893185.

[342] Zhibin Zhou, Qing Gong, Zheting Qi, Lingyun Sun. "ML-Process Canvas: A Design Tool to Support the UX Design of Machine Learning-Empowered Products." In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*. Ed. by Regan L. Mandryk, Stephen A. Brewster, Mark Hancock, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos, Mark Perry. ACM, 2019. DOI: 10.1145/3290607.3312859.

[343] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, Edward Aftandilian. "Productivity Assessment of Neural Code Completion." In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 21–29. ISBN: 9781450392730. DOI: 10.1145/3520312.3534864.

[344] Zhengxia Zou, Keyan Chen, Zhenwei Shi, Yuhong Guo, Jieping Ye. "Object Detection in 20 Years: A Survey." In: *Proceedings of the IEEE* 111.3 (2023), pp. 257–276. DOI: 10.1109/JPROC.2023.3238524.

[345] Igor Zubrycki, Marcin Kolesiński, Grzegorz Granosik. “Graphical Programming Interface for Enabling Non-technical Professionals to Program Robots and Internet-of-Things Devices.” In: *Advances in Computational Intelligence*. Ed. by Ignacio Rojas, Gonzalo Joya, Andreu Catala. Cham: Springer International Publishing, 2017, pp. 620–631. ISBN: 978-3-319-59147-6. DOI: 10.1007/978-3-319-59147-6_53.

All URLs cited were checked in April 2025.

List of Figures

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Dissertation selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich aus der Literatur oder aus anderen Quellen übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

München, den 7. November 2025 Thomas Weber

Ort, Datum, Unterschrift