Extending Jupyter with Multi-Paradigm Editors

THOMAS WEBER, LMU Munich, Germany
JANINA EHE, LMU Munich, Germany
SVEN MAYER, LMU Munich, Germany

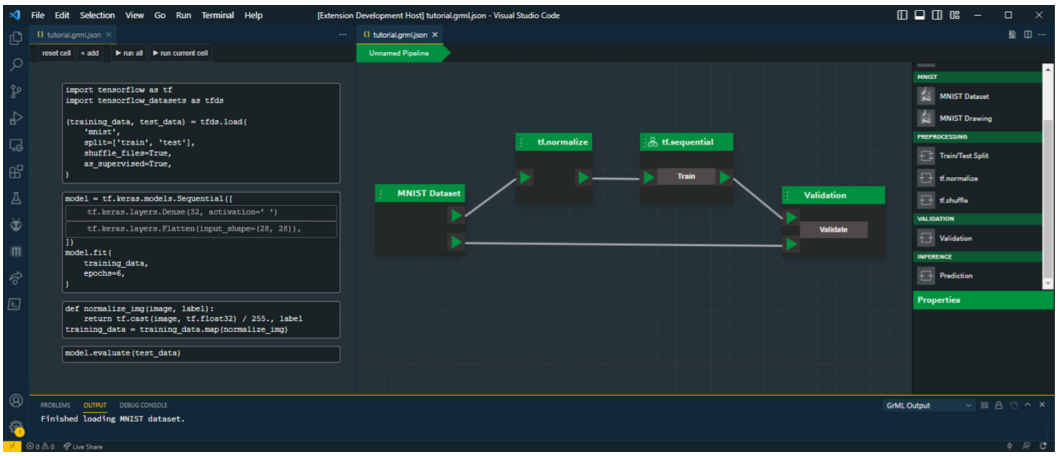

Fig. 1. The multi-paradigm editor is implemented as a Visual Studio Code extension and allows users to open the same file in multiple different editors side by side. On the left, the file is displayed as textual code in a notebook-like interface, while the right side uses the visual programming paradigm with connected boxes. Changes in either editor, graphical or textual, are automatically reflected in the other editor.

Computational notebooks like the Jupyter programming environment have been popular, particularly for developing data-driven applications. One of its main benefits is that it easily supports different programming languages with exchangeable kernels. Thus, it makes the user interface of computational notebooks broadly accessible. While their literate programming paradigm has advantages, we can use this infrastructure to make other paradigms similarly easily and broadly accessible to developers. In our work, we demonstrate how the Jupyter infrastructure can be utilized with different interfaces for different programming paradigms, enabling even greater flexibility for programmers and making it easier for them to adopt different paradigms when they are most suitable. We present a prototype that adds graphical programming and a multi-paradigm editor on top of the Jupyter system. The multi-paradigm editor seamlessly combines the added graphical programming with the familiar notebook interface side-by-side, which can further help developers switch between programming paradigms when desired. A subsequent user evaluation demonstrates the benefits not only of alternate interfaces and paradigms but also of the flexibility of seamlessly switching between them. Finally, we discuss some of the challenges in implementing these systems and how these can enhance the software development process in the future.

Authors' addresses: Thomas Weber, LMU Munich, Munich, Germany, thomas.weber@ifi.lmu.de; Janina Ehe, LMU Munich, Munich, Germany, j.ehe@campus.lmu.de; Sven Mayer, LMU Munich, Munich, Germany, info@sven-mayer.com.

2573-0142/2024/6-ART245 $15.00
https://doi.org/10.1145/3660247

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; *Software creation and management*; • **Computing methodologies** → Machine learning; • **Human-centered computing** → *Empirical studies in HCI*.

Additional Key Words and Phrases: software development, jupyter, development tools

ACM Reference Format:
Thomas Weber, Janina Ehe, and Sven Mayer. 2024. Extending Jupyter with Multi-Paradigm Editors. *Proc. ACM Hum.-Comput. Interact.* 8, EICS, Article 245 (June 2024), 22 pages. https://doi.org/10.1145/3660247

1 INTRODUCTION

Computational notebooks, probably most prominently represented by Jupyter Notebooks[1] or Google Colab[2], are a fairly recent addition to the tool landscape for software development. They offer a user interface for the literate programming paradigm and have gained increasing popularity in general but particularly in the data science domain. In comparison to other established forms of code presentation, they promise several changes that aim to improve the lives of developers [19, 21, 24, 27, 34, 39, 45]. Most notably, they combine code and its output into cells that can be executed individually. The code execution, language permitting, happens just in time and on-demand and directly in the same interface where the code is entered, which allows for fairly rapid exploration. In a way, they offer a modern interface for the well-established REPL (Read-Eval-Print-Loop) interface. Furthermore, the cell structure allows and encourages a combination of documentation, e.g., in Markdown cells, interleaved between codes. A computational notebook thus fills the role of code, documentation, and possibly visualization of results [27, 39].

The underlying infrastructure of notebook environments like Project Jupyter furthermore allows great flexibility by supporting different programming languages with so-called kernels. These kernels are used for executing the code in the cells and can be swapped fairly easily [10, 22, 46]. Additionally, the user interface also offers the option to replace individual cells or their output with widgets, which can either replace existing cells or extend their functionality, e.g., with more elaborate presentations or rich interaction [22, 52]. However, the cell structure of typical computational notebooks can also lead to issues, like confusion about execution order, since the cells can be executed in arbitrary order. A different presentation, that emphasizes the relationship between cells, like the visual, graph-based presentation we use, may provide more clarity. In our work, we investigate how we can best leverage different interaction and programming paradigms on top of the Jupyter infrastructure. Unlike previous works, we do so not just by adding individual widgets or modifying the existing UI but by replacing the full Jupyter front end while preserving the underlying kernel infrastructure with all its benefits.

We present a prototypical implementation that adds a graphical programming environment to the existing cell-based presentation. While a first user study (N = 20) already 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. Such a multi-paradigm presentation increases the flexibility of the users and may assist in various stages of software development. While the complexity of models at the scale used by the global players in this field may exceed what graphical programming can handle, users of small to medium models may benefit in particular. This can be, for example, small to medium enterprises for whom Machine Learning models are only an occasional tool. Since they likely do not have a large team of experts, a multi-paradigm system may allow them to select different, suitable presentations for different stakeholders. Additionally, graphical programming

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

is also suggested for education. The option to seamlessly switch between the different views can also help those with basic knowledge, e.g., students and young professionals, to expand their knowledge step by step. Thus, we collected user feedback during the development (N = 12) from young professionals starting in this domain and professional developers who started to increasingly work with ML, 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.

2 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 [31]. 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.

2.1 Literate Programming and Computational Notebooks

Literate programming as a concept describes the idea that programs should be considered as pieces of literature [24] 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 [24]. 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 [42]. Thus, computational notebooks try to facilitate literate programming by making it easy to keep code and documentation closely intertwined. Lau et al. [25] 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 [36], 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 [27] and for programming education [33, 44].

Project Jupyter is not just the interface but also acts as a platform for different usage scenarios and different target groups [30, 35, 36]. 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 [22], 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. [23] 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 [52]. The code cell structure, which is popular in many

computational notebook systems, also motivated Watson et al. [46] 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. [35] 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. [8] 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. [21] and Lau et al. [25] 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.

2.2 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 [7, 31] 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 [40], 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 [12, 26, 31] 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. [28]). 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 [9, 48, 51].

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 [11], data-relations [17], 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 [2]. 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 [3, 37] or high-performance computing [18, 41].

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 [16, 32, 38]. 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[3] or the Unreal[4]) 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 [50].

2.3 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 [36], there are other tools, some of which also explore alternate programming paradigms.

One example, using the graphical programming paradigm, is RapidMiner [5, 29], 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[29]. In contrast to textual programming, this visual presentation provides a better overview of workflows and accelerates redundant steps, e.g., data preprocessing [5]. KNIME [4] and Orange [15] 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.

Weber and Hußmann [47] 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*

[3] https://unity.com/features/unity-visual-scripting
[4] https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/

established Jupyter infrastructure. Additionally, we also test whether combining different paradigms side-by-side can further assist with this endeavor.

3 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[5].

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 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.

3.1 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 [5, 29] but also other domains, e.g. programming of embedded systems [43], education [49], robotics [53], 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), 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 2). 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.

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

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

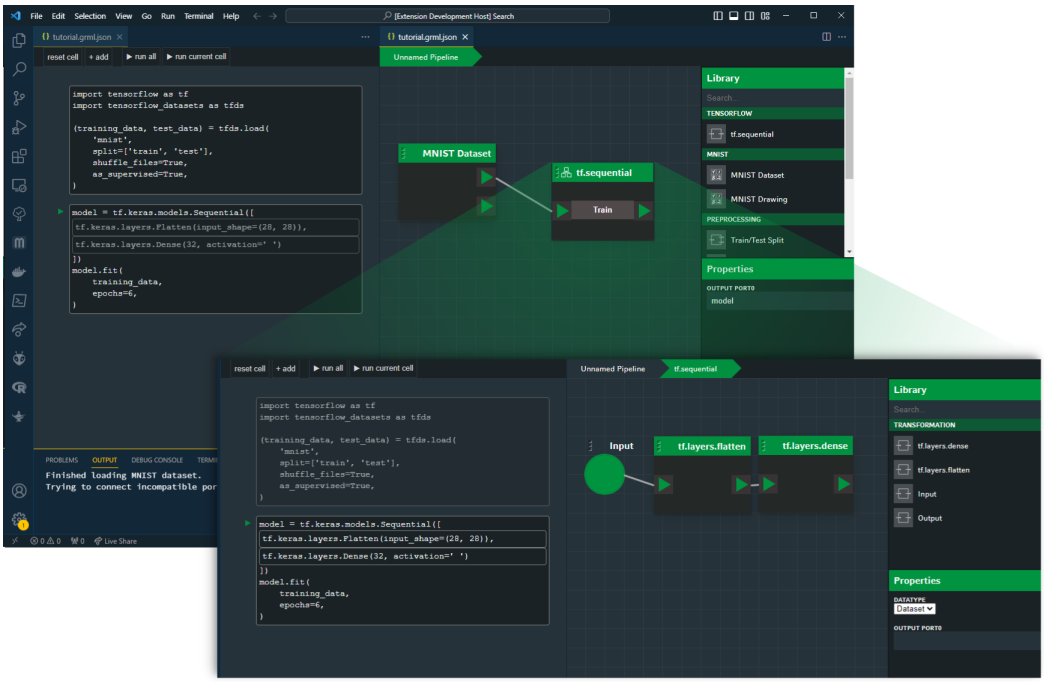

Fig. 2. Certain blocks can be composed of additional sub-blocks, e.g., a neural network can be composed of multiple blocks representing its layers. Double-clicking a block allows the user to navigate through the hierarchy of composite blocks. In the textual view, the sub-blocks are equivalent to groups of lines of code within a cell. The code cell corresponding to the currently active block is highlighted.

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. See Figure 1 for an overview of the layout and the different panels.

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

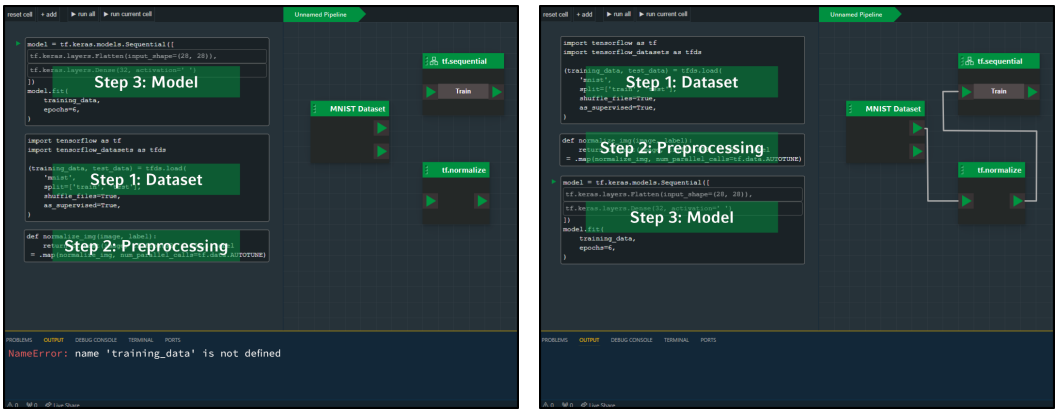

Fig. 3. When adding new cells/blocks, they will appear in the order in which they were created (left) and can be executed in arbitrary order. Particularly in larger projects without rigorous structuring, this can lead to errors during code execution, for example, because variables are set in later cells but used before. Connecting them explicitly (right) defines the correct execution order, preventing these errors.

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.

3.1.1 Serializing Computational Notebooks. When executing code in a computational notebook environment, one known issue [21, 25] 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) 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 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.

3.1.2 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.

3.2 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 1).

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 [14] 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.

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 4 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

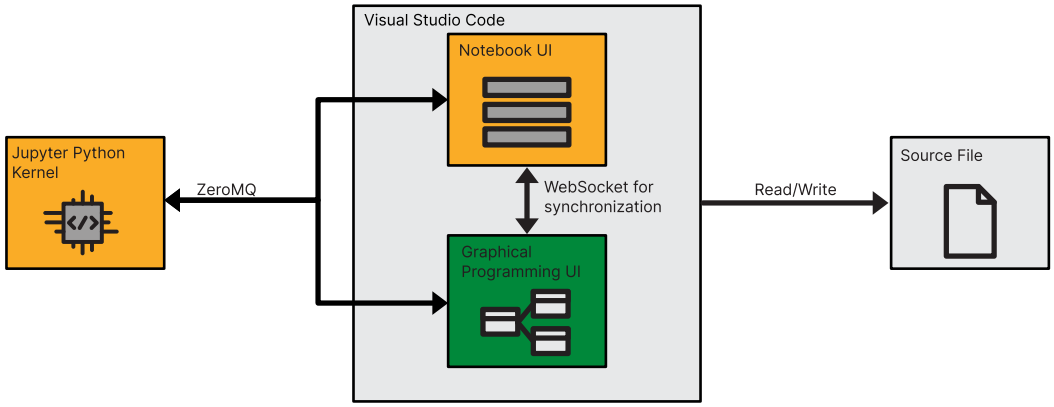

Fig. 4. Our multi-paradigm setup in VSC uses two different user interfaces to display the same underlying source file. These two interfaces with different programming paradigms used WebSockets to synchronize the user interface. For code execution, the code was sent to a shared Jupyter kernel running in the background.

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.

4 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.

4.1 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[6] 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.

4.2 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.

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.

4.3 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 [20], System Usability Scale (SUS) [6], and Technology Acceptance Model (TAM) [13] 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.

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

4.4 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.

4.5 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.

5 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.

5.1 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 [6], 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 [20], 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 1 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.

5.2 Evaluation of the Multi-Paradigm Editors

5.2.1 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 workload but not significantly higher than the baseline from the first study, at an average of 35.1% workload (see Figure 5). 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.

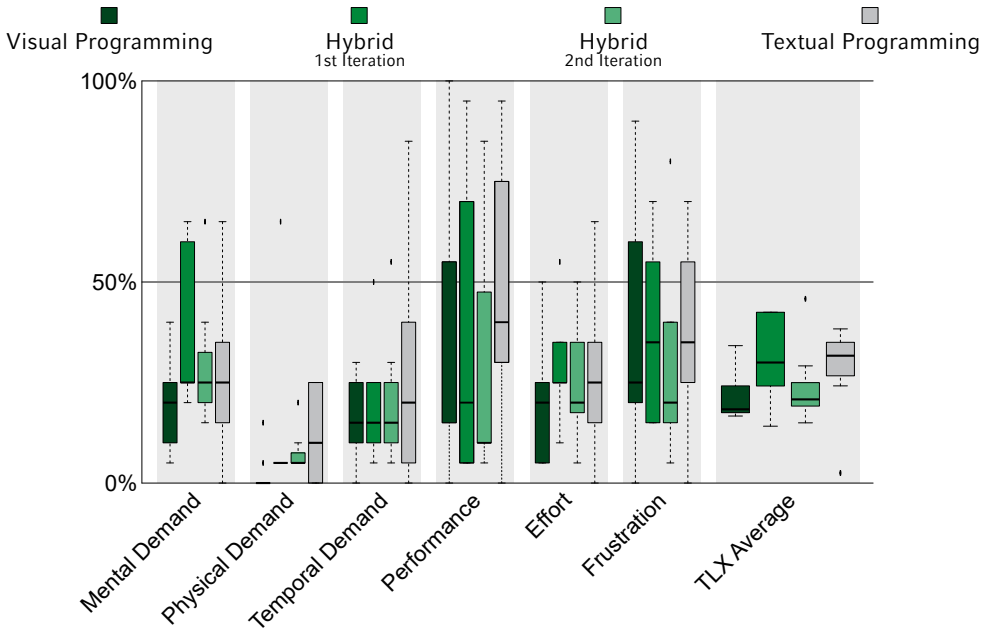

Fig. 5. The raw NASA-TLX results for visual and textual editor in the first study as well as both iterations of the hybrid development environment.

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.

5.2.2 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 5. 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.

6 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.

6.1 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.

6.2 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. 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. 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.

6.3 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.

7 CONCLUSION

We present an exploration of the concept of *Multi-Paradigm Editors*, software development tools that seamlessly combine multiple development paradigms. We demonstrate the technical feasibility of building this kind of system on top of the existing and popular infrastructure of Jupyter notebooks. This work was motivated by a user study that evaluated the option of adding alternative interfaces to the Jupyter infrastructure. The feedback showed that alternative paradigms can be beneficial but developers desire flexibility. Thus, we further extended the Jupyter interface to give developers more choices, which can encourage them to use the programming paradigm most suited for any situation during development. We further evaluated this multi-paradigm editor in the context of data-driven development. Our user studies indicate that adding different development paradigms to a software development tool does not result in a usability or performance penalty. Instead, it increases flexibility. Overall, the multi-paradigm editors can allow developers to easily use the right tool for the job, taking advantage of the specific benefits of each, such as the variability of textual input with the accessible presentation of graphical programming. At the same time, it does not take away familiar tools, but adds to them. Participants in the study highlighted specific scenarios in which they would be willing to use different paradigms, such as visual programming for early project phases when the high-level organization and reduced boilerplate are useful. They also considered the option to switch based on one's own expertise as valuable for the education context.

Beyond improving this specific tool, it will be interesting to see how modern multi-paradigm tools affect different use cases and scenarios within and beyond data-driven development. Given the diverse teams involved in the creation of modern software systems, an investigation of group interaction and how different paradigms help to support and communicate with different audiences, e.g., domain experts, programming novices, or non-programmers, may also provide further insights into how and when to apply different paradigms and how they can be adapted to be more helpful. Finally, we have only investigated the combination of textual and visual presentation in parallel, but many more paradigms can be mixed.

OPEN SCIENCE

We encourage readers to reproduce and extend our results and analysis methods. Therefore, our VSC extension with both types of editors, as well as the task descriptions and tutorials used in the study, are available on GitHub via https://github.com/mimuc/hyde.

REFERENCES

[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, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] William B. Ackerman. 1982. Data Flow Languages. *Computer* 15, 2 (1982), 15–25. https://doi.org/10.1109/MC.1982.1653938

[3] Muhammad Waseem Anwar, Muhammad Rashid, Farooque Azam, Muhammad Kashif, and Wasi Haider Butt. 2019. A model-driven framework for design and verification of embedded systems through SystemVerilog. *Des. Autom. Embed. Syst.* 23, 3-4 (2019), 179–223. https://doi.org/10.1007/s10617-019-09229-y

[4] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. 2009. KNIME - the Konstanz information miner. *ACM SIGKDD Explorations Newsletter* 11, 1 (2009), 26–31. https://doi.org/10.1145/1656274.1656280

[5] Mohamed Bjaoui, Houneida Sakly, Mourad Said, Naoufel Kraiem, and Mohamed Salim Bouhlel. 2020. 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* (Virtual Event, Tunisia) *(DTUC '20)*. Association for Computing Machinery, New York, NY, USA, Article 21, 6 pages. https://doi.org/10.1145/3423603.3424059

[6] John Brooke. 1996. SUS - A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[7] Margaret M. Burnett and David W. McIntyre. 1995. Visual Programming - Guest Editors' Introduction. *Computer* 28, 3 (1995), 14–16. https://doi.org/10.1109/MC.1995.10027

[8] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. 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, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376729

[9] 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, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021), 35. arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[10] Daniel J.B. Clarke, Minji Jeon, Daniel J. Stein, Nicole Moiseyev, Eryk Kropiwnicki, Charles Dai, Zhuorui Xie, Megan L. Wojciechowicz, Skylar Litz, Jason Hom, John Erol Evangelista, Lucas Goldman, Serena Zhang, Christine Yoon, Tahmid Ahamed, Samantha Bhuiyan, Minxuan Cheng, Julie Karam, Kathleen M. Jagodnik, Ingrid Shu, Alexander Lachmann, Sam Ayling, Sherry L. Jenkins, and Avi Ma'ayan. 2021. Appyters: Turning Jupyter Notebooks into data-driven web apps. *Patterns* 2, 3 (2021), 100213. https://doi.org/10.1016/j.patter.2021.100213

[11] Alain Colmerauer and Philippe Roussel. 1996. *The Birth of Prolog*. Association for Computing Machinery, New York, NY, USA, 331–367. https://doi.org/10.1145/234286.1057820

[12] Allen Cypher (Ed.). 1993. *Watch what I do.* MIT Press, London, England.

[13] Fred D. Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* 13, 3 (1989), 319–340. http://www.jstor.org/stable/249008

[14] Robert DeLine and Kael Rowan. 2010. 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*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 207–210. https://doi.org/10.1145/1810295.1810331

[15] Janez Demsar, Blaz Zupan, Gregor Leban, and Tomaz Curk. 2004. 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 (Lecture Notes in Computer Science, Vol. 3202)*, Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi (Eds.). Springer, Berlin, Heidelberg, 537–539. https://doi.org/10.1007/978-3-540-30116-5_58

[16] Stefano Federici. 2011. A Minimal, Extensible, Drag-and-Drop Implementation of the C Programming Language. In *Proceedings of the 2011 Conference on Information Technology Education* (West Point, New York, USA) *(SIGITE '11)*. Association for Computing Machinery, New York, NY, USA, 191–196. https://doi.org/10.1145/2047594.2047646

[17] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. 1984. Logic and Databases: A Deductive Approach. *ACM Comput. Surv.* 16, 2 (jun 1984), 153–185. https://doi.org/10.1145/356924.356929

[18] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. 2011. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 10, 4 (2011), 39:1–39:36. https://doi.org/10.1145/2043662.2043663

[19] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

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

[21] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[22] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. 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* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[23] Mary Beth Kery, Donghao Ren, Kanit Wongsuphasawat, Fred Hohman, and Kayur Patel. 2020. The Future of Notebook Programming Is Fluid. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3334480.3383085

[24] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (01 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[25] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. 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, New York, NY, USA, 1–11. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[26] Henry Lieberman (Ed.). 2001. *Your Wish is My Command*. Morgan Kaufmann / Elsevier, Amsterdam, Netherlands. https://doi.org/10.1016/b978-1-55860-688-3.x5000-3

[27] Eric S. Liu, Dylan A. Lukes, and William G. Griswold. 2023. Refactoring in Computational Notebooks. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 77 (apr 2023), 24 pages. https://doi.org/10.1145/3576036

[28] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. https://doi.org/10.1145/3544548.3580817

[29] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. 2006. YALE: Rapid Prototyping for Complex Data Mining Tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06)*. Association for Computing Machinery, New York, NY, USA, 935–940. https://doi.org/10.1145/1150402.1150531

[30] Michael Milligan. 2017. Interactive HPC Gateways with Jupyter and Jupyterhub. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* (New Orleans, LA, USA) *(PEARC '17)*. Association for Computing Machinery, New York, NY, USA, Article 63, 4 pages. https://doi.org/10.1145/3093338.3104159

[31] Brad A. Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, Massachusetts, USA) *(CHI '86)*. Association for Computing Machinery, New York, NY, USA, 59–66. https://doi.org/10.1145/22627.22349

[32] Mark Noone and Aidan Mooney. 2018. Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education* 5 (2018), 149–174.

[33] Keith J. O'Hara, Douglas S. Blank, and James B. Marshall. 2015. Computational Notebooks for AI Education. In *Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, Ingrid Russell and William Eberle (Eds.). AAAI Press, Hollywood, Florida, USA, 263–268. https://doi.org/10.13140/2.1.2434.5928

[34] Vreda Pieterse, Derrick G. Kourie, and Andrew Boake. 2004. A Case for Contemporary Literate Programming. In *Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries* (Stellenbosch, Western Cape, South Africa) *(SAICSIT '04)*. South African Institute for Computer Scientists and Information Technologists, ZAF, 2–9.

[35] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study about Quality and Reproducibility of Jupyter Notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, New York, NY, USA, 507–517. https://doi.org/10.1109/MSR.2019.00077

[36] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. 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* (Toronto, Ontario, Canada) *(JCDL '17)*. IEEE Press, New York, NY, USA, 338–339. https://doi.org/10.1109/JCDL.2017.7991618

[37] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. 2006. A Model-Driven Design Environment for Embedded Systems. In *Proceedings of the 43rd Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '06)*. Association for Computing Machinery, New York, NY, USA, 915–918. https://doi.org/10.1145/1146909.1147141

[38] William Robinson. 2016. 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, New York, NY, USA, 96–99. https://doi.org/10.1145/2978249.2978265

[39] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[40] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, London, UK.

[41] Christian Scheidler, Lorenz Schäfers, and Ottmar Krämer-Fuhrmann. 1993. TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. In *Parallel Architectures and Languages Europe, 5th International PARLE Conference* (Munich, Germany) *(PARLE '93, Vol. 694)*, Arndt Bode, Mike Reeve, and Gottfried Wolf (Eds.). Springer, Berlin Heidelberg, Germany, 403–413. https://doi.org/10.1007/3-540-56891-3_32

[42] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. 2021. What's in a GitHub Repository? – A Software Documentation Perspective. *arXiv e-prints* (Feb. 2021). https://doi.org/10.48550/arXiv.2102.12727 arXiv:2102.12727 [cs.SE]

[43] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, Alois Zoitl, Christoph Sunder, Antonio Valentini, and Allan Martel. 2008. Framework for Distributed Industrial Automation and Control (4DIAC). In *2008 6th IEEE International Conference on Industrial Informatics*. IEEE, New York, NY, USA, 283–288. https://doi.org/10.1109/INDIN.2008.4618110

[44] Gregory Vial and Bogdan Negoita. 2018. 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 (ICIS)*, Jan Pries-Heje, Sudha Ram, and Michael Rosemann (Eds.). Association for Information Systems, San Francisco, CA, USA, 17. https://aisel.aisnet.org/icis2018/education/Presentations/1

[45] Jiawei Wang, Li Li, and Andreas Zeller. 2020. 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* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 53–56. https://doi.org/10.1145/3377816.3381724

[46] Alex Watson, Scott Bateman, and Suprio Ray. 2019. 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)*. CEUR-WS, Lisbon, Portugal, 4. https://ceur-ws.org/Vol-2322/BigVis_8.pdf

[47] Thomas Weber and Heinrich Hußmann. 2022. Tooling for Developing Data-Driven Applications: Overview and Outlook. In *Proceedings of Mensch Und Computer 2022* (Darmstadt, Germany) *(MuC '22)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/3543758.3543779

[48] Thomas Weber and Sven Mayer. 2024. From Computational to Conversational Notebooks. In *Proceedings of the 1st CHI Workshop on Human-Notebook Interactions* (Honolulu, Hawaii, USA, 2024-05-11). Online.

[49] David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (jul 2019), 22–25. https://doi.org/10.1145/3341221

[50] David Weintrop and Uri Wilensky. 2017. 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, New York, NY, USA, 183–192. https://doi.org/10.1145/3078072.3079715

[51] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

[52] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces*. ACM, New York, NY, USA, 392–404. https://doi.org/10.1145/3490099.3511120

[53] Igor Zubrycki, Marcin Kolesiński, and Grzegorz Granosik. 2017. Graphical Programming Interface for Enabling Non-technical Professionals to Program Robots and Internet-of-Things Devices. In *Advances in Computational Intelligence*, Ignacio Rojas, Gonzalo Joya, and Andreu Catala (Eds.). Springer International Publishing, Cham, 620–631. https://doi.org/10.1007/978-3-319-59147-6_53