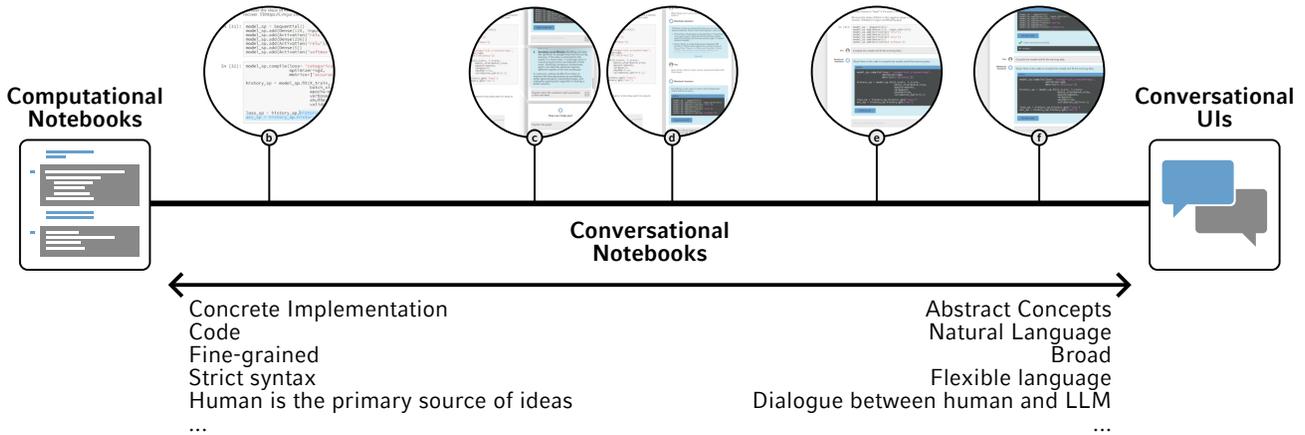


# From Computational to Conversational Notebooks

Thomas Weber  
LMU Munich  
Munich, Germany  
thomas.weber@ifi.lmu.de

Sven Mayer  
LMU Munich  
Munich, Germany  
info@sven-mayer.com



**Figure 1: Conversational Notebooks, including the examples described above, fall on a spectrum, depending on how strongly each of the two interface types is represented in the final design. This affects many aspects of the interface and the interaction with it.**

## ABSTRACT

Today, we see a drastic increase in LLM-based user interfaces to support users in various tasks. Also, in programming, we witness a productivity boost with features like LLM-supported code completion and conversational agents to generate code. In this work, we look at the future of computational notebooks by enriching them with LLM support. We propose a spectrum of support, from simple inline code completion to executable code that was the output of a conversation. We showcase five concrete examples for potential user interface designs and discuss their benefits and drawbacks. With this, we hope to inspire the future development of LLM-supported computational notebooks.

## CCS CONCEPTS

• **Human-centered computing** → **Human computer interaction (HCI)**.

## KEYWORDS

Computational Notebooks, Conversational Interfaces

## 1 INTRODUCTION

Large Language Models (LLMs) have proven to be a technology that greatly impacts many areas of our lives by enhancing, simplifying, or replacing existing tasks, workflows, or activities. Consequently, many companies now integrate LLMs into their products (e.g., Microsoft Copilot<sup>1</sup>, Salesforce<sup>2</sup>, BloombergGPT<sup>3</sup>) to make them and their benefits accessible to businesses and end users.

Naturally, LLMs also greatly affect software development by offering rich capabilities to generate code from natural language descriptions and specifications. While general-purpose LLMs are already quite capable at this, there are also dedicated models [3] for supporting software developers in their work. Products like GitHub Copilot<sup>4</sup>, Amazon CodeWhisperer<sup>5</sup>, and many more are now being used by developers worldwide to enhance their productivity and improve the code and with that the software they write [17].

One common way to integrate these models into the workflow is to prompt them in line in the editor as a richer, more context-sensitive auto-completion feature. While this has proven to increase developer productivity [1, 20], it is tailored towards quickly producing more code [1]. This interaction paradigm has not well supported other uses of LLMs for programming. An alternative is using a conversational user interface where the interaction with



This work is licensed under a Creative Commons Attribution 4.0 International License.  
1st CHI Workshop on Human-Notebook Interactions, May 11, 2024, Honolulu, HI, USA

<sup>1</sup><https://copilot.microsoft.com>

<sup>2</sup><https://www.salesforce.com/news/press-releases/2023/06/12/ai-cloud-news/>

<sup>3</sup><https://www.bloomberg.com/company/press/bloomberggpt-50-billion-parameter-llm-tuned-finance/>

<sup>4</sup><https://github.com/features/copilot>

<sup>5</sup><https://aws.amazon.com/codewhisperer/>

the LLM happens via text chat. Considering how LLM assistants are often considered an alternative to pair programming [2, 8, 14, 15], it makes sense to choose an interaction modality that one would also use with a coworker for interacting with the LLM assistant. Additionally, we see computational notebooks being extended with a wide range of extensions, for instance, graphical programming [27]. Thus, while this allows developers to interact more broadly with the LLM, the question of how best to integrate these interfaces into development tools like computational notebooks is still open.

In this work, we explore how conversational interfaces can extend and improve computational notebooks as one of the popular software development tools based on some existing approaches and feedback from practitioners and experts. We present a series of different design approaches that integrate conversational LLM support into computational notebooks to different degrees. We provide examples across a spectrum of interaction patterns and will discuss these interface designs' potential benefits and drawbacks.

## 2 EXTENDING COMPUTATIONAL NOTEBOOKS WITH LLMs

In this section, we will briefly outline the far ends of the spectrum with standard computational notebooks without LLM support on the one side and pure conversational systems on the other. We will then explore various steps between these two poles and discuss their benefits and drawbacks. These concepts are based on both personal experience and unstructured discussions with practitioners and experts over the span of the last two years.

Computational notebooks like Jupyter notebooks<sup>6</sup> or Google Colab<sup>7</sup> follow the principles of literate programming and are built around the idea of mixing code with contextual and documentary [6, 7, 10, 12, 18, 21, 25]. To this end, a software project is presented as a sequence of individual cells that can be either executable source code, the output of this code, arbitrary text, or even figures and visualizations [12, 21]. While the interface is relatively simple, there is support for some mechanisms commonly found in other software developer tools, such as syntactical suggestions and auto-completion. As mentioned before, the code in the cells can be executed on demand. In this case, the code is typically sent to a backend of so-called "kernels" which execute the code [4, 9, 26], maintain the state of the program, etc. For the popular notebook-style environments, there are kernels for many programming languages. This makes the architecture of computational notebooks quite adaptable to many use cases. The interface itself can also be extended by adding new types of cells.

Still, at its core, computational notebooks closely resemble the old paradigm of the REPL (Read-Eval-Print-Loop), where the developer writes code, executes it, and then iterates upon their code. This requires intimate knowledge of the programming language in question and, in case of errors, a good understanding of the working principles of software code.

On the other hand, recent advancements in LLMs have made it possible for less experienced developers and complete novices to also quickly create software code and iterate on it when it does not perform as desired [16, 19, 22–24]. This is possible because large

language models can generate this output in response to simple language descriptions of the desired functionality. One prominent interface paradigm for this is the use of conversational systems. The user converses with the LLM in a chat style in these interfaces. To create a program, the user can request source code from the LLM via a prompt and, in subsequent prompts, refine it or request changes due to errors in other observations.

At first glance, these conversational systems and the chat histories they produce can look quite similar to computational notebooks: both feature an interleaving of natural language descriptions and code. They also both allow for rapid iteration of small code fragments. However, there are also some obvious differences between these two interfaces: since the computational notebook is built for software development, it comes with the whole infrastructure to quickly and easily execute the code via the kernels as described above. In conversational systems, this is commonly not the case, and the user has to choose an execution environment to see whether it works and does what it is meant to do.

The goal of the typical conversational UI is not to execute code but to give humans a means to interact with the machine in a natural format. While code cells in computational notebooks require a programming language with a rigid syntax, LLM chatbots can produce responses and potentially desired behavior even from vague, misspelled, or ambiguous descriptions in natural language.

So, both these interfaces strive to make two different aspects of software development easier and more accessible. Since notebook UIs offer an easy way to try out code and conversational UIs are an accessible way to create it, the obvious question is how to combine these two UI paradigms to maximize their benefits and find synergies.

*Executing LLM Responses.* Utilizing the infrastructure behind computational notebooks clearly is an easy way to implement this capability of executing code directly from within a chat. Like cells, the LLM code responses could be sent to a kernel backend, and the result becomes part of the chat history. It may be noted that some vendors of conversational LLM interfaces offer the capability to execute the code directly from within the conversation [13], but this is usually still quite limited. However, this is only one way to combine computational notebooks with the capabilities of LLMs into conversational notebooks (Figure 2f).

*In-Cell Coding Assistance.* A second, fairly simple way, and one that is already being practiced, is to use the LLM-powered inline auto-completion of systems like GitHub Copilot to generate code and documentation in individual cells (Figure 2b). Anecdotal evidence suggests, however, that current implementations still struggle with considering the correct and complete context across cells, so this remains an area for future improvements.

Beyond this, a more comprehensive integration of these two paradigms is also possible. One next step towards a more conversational interaction style could be integrating the chat interface directly into the development environment, i.e., the computational notebook. This can be done side by side with the sequence of notebook cells (Figure 2c). Particularly when the chat interface is only used occasionally, requesting it on demand for specific cells seems a way to minimize distractions and visual clutter. This way, users

<sup>6</sup><https://jupyter.org>

<sup>7</sup><https://colab.research.google.com>

A ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0.

Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. <https://i.imgur.com/MieyFyy.png>

```
In [31]: model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))

In [32]: model_sp.compile(loss='categorical_crossentropy',
metrics=['accuracy'])
history_sp = model_sp.fit(X_train, Y_train,
batch_size=batch_size,
epochs=epochs,
verbose=0,
shuffle=True,
validation_split=0.1)
loss_sp = history_sp.history.get('loss')
acc_sp = history_sp.history.get('acc')
```

**Sigmoid**

Not recommended because convergence is too slow and it is easy to occur vanishing gradient problem.

A ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0.

Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. <https://i.imgur.com/MieyFyy.png>

```
In [31]: model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))

In [32]: model_sp.compile(loss='categorical_crossentropy',
metrics=['accuracy'])
history_sp = model_sp.fit(X_train, Y_train,
batch_size=batch_size,
epochs=epochs,
verbose=0,
shuffle=True,
validation_split=0.1)
loss_sp = history_sp.history.get('loss')
acc_sp = history_sp.history.get('acc')
```

(a) Without LLM support.

(b) With inline LLM powered code completion.

outputs 0.

Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. <https://i.imgur.com/MieyFyy.png>

```
In [31]: model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))

In [32]: model_sp.compile(loss='categorical_crossentropy',
optimizer=sgd,
metrics=['accuracy'])
history_sp = model_sp.fit(X_train, Y_train,
batch_size=batch_size,
epochs=epochs,
verbose=0,
shuffle=True,
validation_split=0.1)
loss_sp = history_sp.history.get('loss')
acc_sp = history_sp.history.get('acc')
```

**Sigmoid**

Not recommended because convergence is too slow and it is easy to occur vanishing gradient problem.

**4. Avoiding Local Minima:** Shuffling can help the optimizer to escape local minima during training. If the data is presented to the model in a fixed order, it might get stuck in a local minimum that corresponds to that order. Shuffling introduces randomness, which can help the optimizer explore different regions of the loss landscape. In summary, setting shuffle=True helps to improve the training process by promoting better generalization, preventing biases, and aiding the optimization algorithm in finding a better solution.

Explain what the validation split parameter in this cell does

How can I help you?

Explain the graph

outputs 0.

Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. <https://i.imgur.com/MieyFyy.png>

```
In [31]: model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))

In [32]: model_sp.compile(loss='categorical_crossentropy',
optimizer=sgd,
metrics=['accuracy'])
history_sp = model_sp.fit(X_train, Y_train,
batch_size=batch_size,
epochs=epochs,
verbose=0,
shuffle=True,
validation_split=0.1)
loss_sp = history_sp.history.get('loss')
acc_sp = history_sp.history.get('acc')
```

**Sigmoid**

Not recommended because convergence is too slow and it is easy to occur vanishing gradient problem.

You: What library can I use to build a neural network in python?

**Notebook Assistant**

In Python, there are several libraries you can use to build neural networks. Some of the most popular ones include:

- TensorFlow: Developed by Google Brain, TensorFlow is an open-source machine learning library widely used for building and training various types of neural network models.
- Keras: Keras is a high-level neural networks API written in Python and capable of running on top of TensorFlow, Theano, or Microsoft Cognitive Toolkit (CNTK). It provides a user-friendly interface for building and training neural networks.

Expand

You: Give me the code to create a keras sequential model with three layers

**Notebook Assistant**

```
python
model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))
```

Copy to new cell

What can I do to improve the accuracy of the model in cell 32?

(c) With per-cell LLM conversational support

(d) With global LLM conversational support

A ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0.

Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. <https://i.imgur.com/MieyFyy.png>

```
In [31]: model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))
```

You: Compile the model and fit the training data.

**Notebook Assistant**

Okay! Here is the code to compile the model and fit the training data:

```
python
model_sp.compile(loss='categorical_crossentropy',
optimizer=sgd,
metrics=['accuracy'])
history_sp = model_sp.fit(X_train, Y_train,
batch_size=batch_size,
epochs=epochs,
verbose=0,
shuffle=True,
validation_split=0.1)
loss_sp = history_sp.history.get('loss')
acc_sp = history_sp.history.get('acc')
```

Insert as cell

Ask your Notebook Assistant

python

```
model_sp = Sequential()
model_sp.add(Dense(128, input_dim=200))
model_sp.add(Activation('relu'))
model_sp.add(Dense(256))
model_sp.add(Activation('relu'))
model_sp.add(Dense(5))
model_sp.add(Activation('softmax'))
```

Run the code

Code executed successfully

No output

You: Compile the model and fit the training data.

**Notebook Assistant**

Okay! Here is the code to compile the model and fit the training data:

```
python
model_sp.compile(loss='categorical_crossentropy',
optimizer=sgd,
metrics=['accuracy'])
history_sp = model_sp.fit(X_train, Y_train,
batch_size=batch_size,
epochs=epochs,
verbose=0,
shuffle=True,
validation_split=0.1)
loss_sp = history_sp.history.get('loss')
acc_sp = history_sp.history.get('acc')
```

Run the code

Ask your Notebook Assistant

(e) With inline LLM conversational support

(f) With inline LLM support and executable code responses.

Figure 2: The interfaces with increasing support of LLMs. a) shows the state of the art without LLM support, while b)-f) show the LLM support, which is highlighted in blue for illustration purposes.

can also have the chat history about a specific piece of code directly available without having to scroll through many unrelated messages to a specific cell.

*Side-by-Side Interfaces.* However, suppose the chat considers the whole notebook as context. In that case, it is equally plausible to add a global chat next to the whole notebook and have these two interaction paradigms side by side (Figure 2d). The Visual Studio Code extension for GitHub Copilot, for example, allows a layout that is very similar to that. However, so far, the chat is mostly a separate interface with only some functionality to simplify providing context to the LLM and copying back its results. The integration with the structure of notebooks is also still lacking at times, leading to limited options for interaction between the conversational agent and the notebook. Consequently, copying and pasting between the notebook and the chat is often necessary.

*Conversational Notebooks.* To minimize the overhead and integrate conversational interaction more tightly into the notebook structure, we could add another cell type as a conversational cell (Figure 2e). Similar to how a developer would type code and the output is displayed below, in this cell type, the user could enter a prompt into the cell in the notebook, and the output below would be the response of the LLM.

*Executable Conversations.* Since such a cell is then part of the program, we may treat it not just as a means to produce the code, but an ML-powered cell could directly alter the program state. In this case, we would skip the intermediate step of translating the natural language description to code and executing it. Instead, we would execute the natural language description directly. Effects and outputs of the execution could then be presented in natural language as well, leading to a seamless natural language conversation with computation in the background. In case of errors, there is still some value to debugging the code, but for the most part, it could be kept hidden, making notebooks more human-readable and accessible.

These different user interfaces all support the two key aspects, simplifying code creation and execution, and additional development tasks like documentation, explainability, collaboration, etc., to varying degrees. This emphasizes how there is no way to integrate LLM assistance into computational notebooks correctly. Instead, there appears to be a spectrum of how strongly the conversational aspects are integrated into computational notebooks or how the existing infrastructure of computational notebooks is utilized for the LLM (see Figure 1). Depending on how this integration is executed in any novel conversational notebook interface, i.e., where it is placed on this spectrum, it will not just affect how to interact with it. Many aspects, like the ratio of natural language to programming languages, the granularity of the descriptions, etc., will give each interface distinct use cases and advantages and drawbacks.

## 2.1 Advantages and Drawbacks

While all these variants levels of conversational and LLM interaction in notebooks are viable, they will all be more suitable in some circumstances than others.

*Inline Auto-Completion:* Adding inline auto-completion promises a considerable productivity benefit for developers. Existing research [1] shows how particularly experienced developers can utilize this mechanism to build a piece of software using many small-scale completions quickly. Since the code completions can be small, it is easier for the developer to maintain an overview of the actual generated code. Meanwhile, the time spent typing out large swaths of boilerplate code is greatly minimized. However, the speed increase may tempt some developers to breeze through the coding part and neglect documentation. While they may write comments to trigger auto-completion, these comments are intended for the LLM, so they are not necessarily phrased in a way that is most helpful for a human reader. Additionally, this interface is clearly tailored towards generating code. This interface is less ideal for “discussing” the model with the LLM, for example, asking for clarification or justification. However, prior studies demonstrated how developers can benefit from LLMs in this regard [1, 20]. Conversational interfaces can also act as a partial replacement for extensively searching through API documentation and generally replace online search. Performing these tasks from within the code using code comments is sub-optimal at best.

*Integrated Chat:* These higher-level activities are where a separate chat interface alongside the computational notebook can shine, allowing for this kind of discussion in a dedicated space. As previously mentioned, both multiple smaller and larger global chats will add to the visual clutter and likely also to the mental workload of such an interface. Furthermore, the interaction between the notebook cells and the conversational interface is also an area where the interaction design will have a considerable impact. This interaction goes bi-directionally for sending contextual information from the notebook to the chat and integrating the responses into the notebook cells. However, the nature of a notebook can provide some valuable additional context for the LLM. For example, the model could automatically consider the distinction between code, output, and documentation cells to determine the users’ intention, the approach, and the actual result. When an LLM considers this context, it makes sense to do this implicitly, not by copying redundant information into the prompt. However, this can make it less understandable, so mechanisms need to be in place to explain which context informed certain decisions in the code generation.

*Challenges of the Notebook Paradigm:* An additional open challenge remains the well-known fact that computational notebooks prescribe no strict execution order to the cells [7, 11]. This means that, e.g., through iteration, earlier cells are changed while later cells are outdated. This may lead an LLM to consider misleading context or suggest code changes that are detrimental to other code in the notebook. This further adds to the need for clear explainability of how the context played a role in the generated code.

When conversational cells are integrated into the notebook structure, one policy might be to consider just the previous cells as definitive context and assume that the following cells will be subject to change and are thus unreliable. However, the following cells will contain valuable context even if they become changed. Enforcing strict execution order of cells would theoretically solve this issue, but developers may consider this too restrictive and undesirable. However, this issue is one present for notebooks in general, not

just conversational notebooks, so there should be broader research into adequate mechanisms to inform users and prevent or deal with errors due to execution order.

*Accessibility vs. Accessing All Information.* Finally, treating natural language input as executable cells by using LLMs in the background can turn computational notebooks into mostly natural language documents. This makes it accessible for them to understand what a complete notebook does, but it will obscure the detailed inner workings. Particularly during development, this may be undesirable, and for debugging, it will be necessary to display the generated code. This way of integrating LLMs may be most viable if there is a mechanism in place to switch different views on a notebook, e.g., a development view that displays prompts, generated code, and outputs, including intermediate outputs, and a high-level view that displays the natural language description of the code and the relevant final outputs. This kind of multi-view notebook might also support collaboration: first, a domain expert notes down the specification in natural language cells. Then, an LLM generates code from it, or a developer writes it with LLM support. After a debugging and refinement phase with the code, it is then translated back to natural language descriptions using LLM-based source code summarization, which can again be viewed by non-developers, e.g., domain experts, end users, etc.

### 3 FUTURE WORK

Evidently, combining computational notebooks with conversational interfaces has great potential to improve both interfaces. Both share the goal of making programming more accessible - conversational interfaces with LLMs that make it easier to produce code and computational notebooks by simplifying code execution and iteration. A combination of these interfaces can unlock synergies and create a user interface that offers the advantages of both while minimizing potential shortcomings.

Above, we outlined a number of interface concepts on the intersection of these UI types. We positioned these along a spectrum with varying degrees of conversational interaction and integration into notebooks. These only present a few ways to combine conversational interfaces with computational notebooks into conversational notebooks. We look forward to seeing how computational notebooks will improve in the future beyond some of the trivial ways that are currently being used, as listed above. In this endeavor, the spectrum we described should give authors of future conversational notebooks a point of reference to position their own interface concepts in relation to other design concepts.

Beyond designing and implementing more and different conversational notebooks, the obvious next step is to test whether these potential benefits actually hold true during real-world usage. What metrics can determine whether conversational notebooks are a good interface will depend on a number of factors, though. Most important among those is the intended target group. As mentioned, both conversational interfaces and computational notebooks are means of making programming more accessible. Therefore, one possible target group is novices and domain experts who need to translate their expertise into software. For this target group, aspects of understandability, exportability, and overall positive user experience with little mental load seem desirable. Here, the goal

might be a conversational notebook that lets domain experts focus on their area of expertise but guides them through the steps to produce the desired software. The ability to engage with the LLM in natural language to change existing code and ask for clarification or explanations will also help novice users to debug and iterate their software.

On the other hand, an improved development interface will also benefit professional software engineers. However, for this target group, an increase in productivity may be of greater interest. Of course, productivity can also include aspects that may benefit novices, like the ability to explain existing code. Beyond this, productivity for professional users also includes more quantifiable improvements like increased output, improved software quality, etc. [5].

Consequently, it may very well be that novice and professional users will prefer the conversational aspect in their tools to different degrees. Still, even for professional developers, it has been observed that they use LLM assistance in different modes [1]: when they rapidly request short, immediate suggestions, a well-integrated conversational system may be the top choice. When they are more in an “explorative” mode [1], an open conversation may be the superior option, and a tight integration may be less important.

This points towards the fact that any exploration of the interface along the spectrum of conversational notebooks may have merits and maybe the top choice in certain scenarios and less ideal in others. Thus, it will be the task for researchers in this area to explore how to best utilize the synergies between computational notebooks and conversational systems and provide guidance to professional and amateur users alike when and how best to achieve their goals with conversational notebooks.

### REFERENCES

- [1] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. <https://doi.org/10.48550/ARXIV.2206.15000>
- [2] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (jan 2023), 35–57. <https://doi.org/10.1145/3582083>
- [3] 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>
- [4] 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>
- [5] Nicole Forsgren, Margaret-Anne D. Storey, Chandra Shekhar Maddila, Thomas Zimmermann, Brian Houck, and Jenna L. Butler. 2021. The SPACE of Developer Productivity: There’s more to it than you think. *ACM Queue* 19, 1 (February 2021), 20–48. <https://doi.org/10.1145/3454122.3454124>
- [6] 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>
- [7] 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>
- [8] Saki Imai. 2022. Is GitHub copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania, USA) (ICSE '22). Association for Computing Machinery, New York, NY, USA, 319–321. <https://doi.org/10.1145/3510454.3522684>
- [9] 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>
- [10] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (01 1984), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>
- [11] 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>
- [12] 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>
- [13] Yiwen Lu. 2023. What to Know About ChatGPT's New Code Interpreter Feature. *The New York Times* (11 July 2023). <https://www.nytimes.com/2023/07/11/technology/what-to-know-chatgpt-code-interpreter.html>
- [14] Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. 2023. Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming. arXiv:2306.05153 [cs.HC]
- [15] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023), 111734. <https://doi.org/10.1016/j.jss.2023.111734>
- [16] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2021. Examining Zero-Shot Vulnerability Repair with Large Language Models. <https://doi.org/10.48550/ARXIV.2112.02125>
- [17] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. <https://doi.org/10.48550/ARXIV.2302.06590>
- [18] 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.
- [19] Julian Aron Prenner and Romain Robbes. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. <https://doi.org/10.48550/ARXIV.2111.03922>
- [20] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. 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* (Sydney, NSW, Australia) (IUI '23). Association for Computing Machinery, New York, NY, USA, 491–514. <https://doi.org/10.1145/3581641.3584037>
- [21] 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>
- [22] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. <https://doi.org/10.48550/ARXIV.2301.08653>
- [23] Anna Trudova, Michal Dolezel, and Alena Buchalceková. 2020. 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*. SCITEPRESS, Prague, Czech Republic, 181–192. <https://doi.org/10.5220/0009417801810192>
- [24] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. <https://doi.org/10.48550/ARXIV.2009.05617>
- [25] 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>
- [26] 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](https://ceur-ws.org/Vol-2322/BigVis_8.pdf)
- [27] Thomas Weber, Janina Ehe, and Sven Mayer. 2024. Extending Jupyter with Multi-Paradigm Editors. *Proc. ACM Hum.-Comput. Interact.* 8, EICS (2024), 1–22. <https://doi.org/10.1145/3660247>