On the Design of AI-powered Code Assistants for Notebooks

Andrew McNutt
mcnutt@uchicago.edu
University of Chicago
Chicago, IL, USA

Chenglong Wang
chenwang@microsoft.com
Microsoft research
Redmond, WA, USA

Rob DeLine
rob.DeLine@microsoft.com
Microsoft research
Redmond, WA, USA

Steven M. Drucker
sdrucker@microsoft.com
Microsoft research
Redmond, WA, USA

ABSTRACT

AI-powered code assistants, such as Copilot, are quickly becoming a ubiquitous component of contemporary coding contexts. Among these environments, computational notebooks, such as Jupyter, are of particular interest as they provide rich interface affordances that interleave code and output in a manner that allows for both exploratory and presentational work. Despite their popularity, little is known about the appropriate design of code assistants in notebooks. We investigate the potential of code assistants in computational notebooks by creating a design space (reified from a survey of extant tools) and through an interview-design study (with 15 practicing data scientists). Through this work, we identify challenges and opportunities for future systems in this space, such as the value of disambiguation for tasks like data visualization, the potential of tightly scoped domain-specific tools (like linters), and the importance of polite assistants.

CCS CONCEPTS

• **Human-centered computing** → **Human computer interaction (HCI)**; • **Computing methodologies** → **Natural language processing**; • **Software and its engineering** → **Integrated and visual development environments**.

KEYWORDS

Computational Notebooks, Artificial Intelligence, Code Assistant, Copilot, Design Probe

ACM Reference Format:
Andrew McNutt, Chenglong Wang, Rob DeLine, and Steven M. Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544548.3580940

CHI '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9421-5/23/04…$15.00
https://doi.org/10.1145/3544548.3580940

1 INTRODUCTION

AI-powered code assistants like GitHub *Copilot* [25] are designed to improve programmers' productivity. Powered by large language models (LLMs), these tools can automatically generate high-quality code suggestions from a programming context—consisting of both code and natural language, such as comments or docstrings. Interaction with code assistants typically occurs as part of normal program authoring: the programmer indicates a context in which they would like assistance, which the assistant uses to provide a list of suggestions. The programmer selects a desired code recommendation, adapts it to fit their context, and continues programming. Through this cycle, users can reduce the burden of boilerplate [81], increase perceived programming productivity [84, 98], and receive guidance on how to address unfamiliar tasks [4].

Given this facility to improve developer experience, these AI-powered code assistants have the potential for enormous impact in *computational notebooks*, such as Jupyter [71]. However, the simple text-based interactions suited to more traditional code editors do not naturally fit into notebook workflows. For instance, in notebooks, programming is not limited to textual inputs but can involve a wide variety of multi-modal data—such as code, markdown, data tables, and plots. Notebook users often write small, loosely structured code snippets, which they execute interactively to understand the code and data [41]. To support this often exploratory style, these code snippets are often written and executed in a nonlinear order, with parallel solutions being examined in an iterated and interleaved manner [87].

Without accounting for these differences, a purely code-based interface to code assistants would make it challenging for notebook users to specify the desired context, understand assistant suggestions, and adapt them into their work. In this work, we seek to enable future system designers by considering

(1) *What choices are available in the design of AI-powered code assistants in notebooks?*
(2) *What do users expect from such assistants in this context?*

To answer these questions, we conducted two studies, a design space analysis, and a semi-structured design study.

In the first of these, we sought to characterize the design space by surveying interaction designs in notebooks whose goal is to improve the end-user programming experience through code generation. Despite their often ad hoc or domain- and algorithm-specific design, these systems—which range from integrating graphically

specified elements [42] to live spreadsheet manipulations [19]—provide valuable insights for the design of AI-powered assistants more generally. We extrapolate these design choices into a collection of design concerns (Sec. 3), which we display in Fig. 2 and Fig. 3. We classify the design space based on interface components (user gestures, model artifacts, disambiguation, and refinement interfaces) as well as the relationship between these components (code context, provenance management, model specialization, and customizability). We phrase our concerns in a generalizable manner that can be used to generatively explore the space.

Building on the structure of this space, we conducted a second study that sought to understand data scientists' expectations of AI-powered code assistants in notebooks. In this study, we interviewed 15 professional data scientists about their preferences for tools in this space. We presented participants with various designs that probed their perceptions of context specification, suggestion disambiguation, result adaptation, and code provenance situated within several data analysis and visualization tasks (Sec. 4).

While different participants preferred different options in the design space, they were unanimously enthusiastic about the potential that our probes suggested. Our study revealed a rich set of predilections about this style of system (Sec. 5), which we summarize in Table 1. Most notable among these: the importance of polite interfaces that respect users' agency and flow, but are sufficiently prominent to promote usage; the potential that linter-like assistants that highlight inappropriate usage in an approachable manner might have when scoped to a specific medium or task (such as data science or notebooks); that assistants might usefully take on a variety of interface forms throughout the notebook to aid in a corresponding array of tasks.

Integration of code assistants backed by AIs offers rich potential to radically improve notebook users' programming experience and efficacy. Through our characterization of the design space and elicitation of realistic-user expectations and opinions, we hope to empower future designers to build more helpful code assistants.

2 RELATED WORK

Our work is informed by prior work on code generation models, interfaces for interacting with those models, design interventions in notebooks, and code suggestions more generally.

Code Generation. Program synthesizers (which we refer to more generally as code generation) seek to reduce programming effort by automating challenging or repetitive programming tasks. These techniques automatically generate code based on high-level specifications—such as through demonstrations, input-output examples, natural language, and partial implementations [29].

Program synthesizers come in a variety of styles, but of particular interest to our work are code generation models, especially those powered by LLMs. These models automatically learn program concepts from large-scale code corpora and are typically not limited to a specific language or a task domain. For example, Codex [12] (the model used in *Copilot*), InCoder [24], and CodeGen [64] are GPT-style code generation models [68] trained on GitHub repositories (among other sources) that support code generation from texts and partial implementations in most mainstream programming languages. A number of code assistants have been developed based on these models [2, 20, 25, 83].

Despite their support for general languages and tasks, these models are no panacea: in addition to requiring significant natural resources to train [9], they often handle specific application scenarios poorly. These issues can sometimes be addressed through prompt engineering [36, 76] (rephrasing the specification in languages that the model understands) or fine-tuning [12, 14] (additional training on datasets that represent the application domain).

Our work focuses on *Copilot*, not because we seek to develop a better understanding of it in particular, but instead using it as a representative of a wider class of LLM-driven code assistants.

Interacting with Code Assistants. A major open question is how to employ code generation models in a way that is understandable and maintains users' sense of control. This has long been a concern of mixed-initiative systems [33], but only recently have code assistants become powerful enough to enable study of their interfaces.

A variety of interface strategies have been developed to adapt code assistants to particular domains. Jiang et al. [38, 39] describe GenLine: an IDE-based LLM-powered code assistant for HTML/JS applications. Users guide synthesis through natural language prompts and control context through "code fences", which explicitly indicate what is and is not relevant to code generation. Ferdowsifard et al. [23] provide a live programming interface that involves an always-on display of values in small python programs that use local test cases to guide synthesis. To aid users in providing better specifications (or prompts) to the code generation models, AI Chains [92] and PromptIDE [82] help users experiment and refine their prompts. Our work draws on these design patterns to understand the space of possible interface elements that might be used in our domain. In addition to these tools, some systems focus specifically on code generation in notebooks (Sec. 3).

GitHub *Copilot* [25] is the first LLM-based code assistant powered to reach widespread usage. It provides a text-based interface where users provide tacit descriptions of context, either through description or partial programs. Recent studies of user experiences have sought to characterize the user experience and perceptions of interacting with AI-powered assistants. Barke et al. [4] find that users tend to use *Copilot* either in an acceleration mode (in which it is used to enhance their efficiency) or an exploration mode (in which users are aided in understanding the available program space). Sarkar et al. [81] find that *Copilot* usage has similarities to search, compilation, and pair programming. Jayagopal et al. [37] analyze how novices learn to use code generation tools and highlight the tensions between triggered and triggerless initiation and communication of code generation. Other studies suggest that code generation quality depends on user experience level [98], that less experienced users had difficulty understanding, editing, and debugging generated code [84], and that users may not trust generated code compared to code produced by a human pair programmer [35]. Our work extends these efforts by considering AI-powered code assistants in a particular medium.

Intervening in Computational Notebooks. Computational notebooks, like Jupyter [71], have become a popular environment for data scientists to conduct some aspects of their work [46, 80]. While

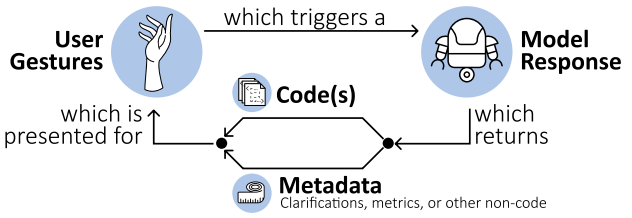


Figure 1: The *specification-refinement loop* interaction underlying our design space.

AI-powered code assistants like *Copilot* enhance software development environments, researchers meanwhile have explored other types of code assistance in computational notebooks [58, 87]. For instance, Kery et al. [40] explore design interventions for adapting the notion of history to the specifics of notebooks, while Head et al. [30] apply program slicing as means to automatically organize notebooks. To help with end-user programming in notebooks, tools such as *Mage* [42] let users specify code using graphical elements (e.g. through a visualization widget). *Mito* [19] and *B2* [93] allow users to manipulate data directly through spreadsheet interfaces and dashboards, respectively, where the interactions are recorded as code. Additional notebooks have been developed to address notebook usability issues [45]. For instance, *Glinda* [18] is a notebook with live programming and declarative language support, while Lodestar [73] enables automation of data science workflows through analysis templates provided via automated recommendation. Our efforts are closely related to these, as we seek to enhance the end-user experience of notebooks by better understanding the design of code assistants in notebooks.

Code Suggestions. A common strategy for making code easier to produce is through in-editor code suggestions, of which generative assistants (such as *Copilot*) are only the latest incarnation. This approach is ubiquitous (arising as early as 1985 [1]), appearing in IDEs, purpose-specific editors like Jupyter [58] or Excel [60], as well as specific-domains such as exploratory data analysis [50], data visualization [78], and computational notebooks [49]. This strategy might take the visual form of an autocomplete (as in IntelliSense [59] or Calcite [61]) or through UIs with search-like signifiers like a query bar (as in Blueprint [11] or Bing Developer Assistant[97]). The interface for asking for a suggestion (and therein specifying intent and context) can involve caret position [11, 59, 61, 66], code comments [95], partial implementations [74], integrated examples [67], or explicit querying. Liu et al. [52] find that code search can use a variety of input modalities, including free text, source code, API descriptions, input-output examples, test cases, and UI sketches—typically without direct control from the user [27]. Some works [11, 78] highlight the value of integrating documentation, provenance, or justification as part of the generated results. In modeling interactions with generative code assistants (Sec. 3), we did not analyze purely search-based interfaces, although generation and search are typically seen as closely-related interactions [81] (Sec. 5.2.2). For instance, Xu et al. [94] compare search and generation through an assistant that mixes both approaches, finding that each modality is beneficial for different tasks. Our studies draw on the patterns found in these interaction forms, however, our exclusion of search systems is a limitation of our approach.

3 DESIGN SPACE

To explore design options for AI-powered code assistants, we first analyzed the broader design space of how code assistants are deployed in computational notebooks. We consider systems that emit textual code based on some user gesture[1] to assist user programming, with or without AI backends. Through this work, we develop a characterization of this design space that describes both *interfaces* and *interactions* available in such systems. Our approach draws on observational and reflective approaches [62].

From the observational side, we sought to assemble a maximum variation sample [69] of this interface form to understand as many different approaches as available. We did so by searching Google Scholar for systems involving notebooks and code generation from which we iteratively snowball sampled. Through this process, we identified 14 systems: *B2* [93], *bamboolib* [44], *Copilot* [25], *Gauss* [7], *Glinda* [18], *Hex* [54], *Jigsaw* [36], *Lux* [47], *Mage* [42], *Mito* [19], *Observable data table* [65], *Tabnine* [83], *VizSmith* [6], and *Wrex* [21]. These systems range from domain-specific code assistants (e.g. assistants for spreadsheet data manipulation or statistical analysis) to live programming environments and frameworks for building GUI-embedded widgets. While there are other systems of interest, we believe this selection is sufficient for our analysis. We provide example images of each system in the appendix.

Despite the diversity of these systems, some corners of this relatively new space remain under-explored, and some dimensions may simply reflect the design choices that Jupyter or its Extension API impose. To address these biases, we augmented our observations with a series of identified properties. For instance, in our analysis of where code assistants are located within the notebook, we observed systems using only Ambient, Inline, and Application style assistants but missing cell-level designs, so we extended the design space to include this option. Finally, while closely connected, we exclude search-based systems to limit scope and because generative assistants seem [81] to be used differently than search systems.

3.1 The Design Space

We describe the interaction between the user and the code assistant as a *specification-refinement loop* (Fig. 1). In this loop, the user specifies the programming goal using a sequence of gestures, then the system returns code or metadata to the user, which forms the basis for the next cycle of interaction. We capture how different systems embody this loop along two principal categories, namely interactions with the assistant (Sec. 3.1.1) and relationships between the code assistants and other notebook components (Sec. 3.1.2), which is based on Beaudouin-Lafon's dichotomization of interactions versus interfaces [8]. Within these categories, the reviewed systems vary along several dimensions. These categories and their dimensions constitute our design space (Fig. 2, Fig. 3) wherein systems can be described by choosing a value (or values) from each dimension. The remainder of this section explains each dimension.

3.1.1 Interactions. We begin by looking at the dimensions that describe the user interactions with the system.

[1]We use the term *gesture* to cover a variety of input modalities, such as direct manipulation, form usage, coding, and natural language.

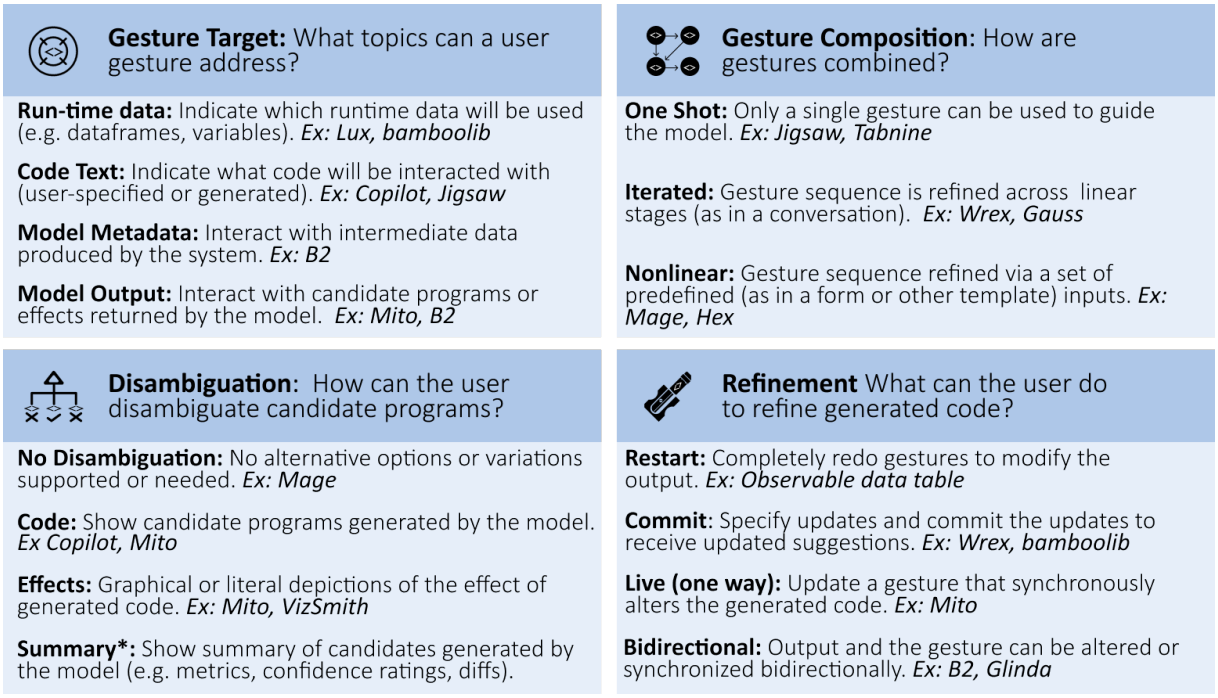


Figure 2: The components of our design space considering interactions with the code assistant. Among these topics, our interview study investigates *Disambiguation* interaction most directly.

Gesture Target. Each user gesture targets an entity. In our context, these include components from the notebook environment and artifacts produced by interaction with the assistant. Notebook targets include Runtime Data, such as variables and data frames from the execution context (as in *bamboolib* [44]), and Code Text, such as text-represented program snippets in code cells (as in *Jigsaw* [36]). The model-generated targets include Model Metadata, such as confidence in textual predictions, model parameters or partial programs (as in *B2* [93]), and Model Output such as the code generated by the model or visualizations of its effects (as in *Mito* [19]). This rich space of gesture targets defines the basic means through which the user can communicate with the code assistant.

Gesture Composition. It is often necessary to sequence or compose different user gestures to express more nuanced intent. The simplest way is One Shot interaction, wherein each user gesture triggers an independent interaction with the assistant, as is the case with *Tabnine* [83]. For multi-step interactions between user and system, systems can employ either Iterated, in which gestures are used to iteratively or conversationally refine intent (as in *Wrex* [21]), or Nonlinear, in which gestures are inputted in an unprescribed order like in a template—as in the *Hex* [54] query builder. The means of composition informs how state is handled and how the UI should present interactions with that state.

Disambiguation. Due to incompleteness of the user specification and uncertainty of the underlying model, many systems produce a set of candidate suggestions rather than a single result. We highlight the presentation of these possible results as a potentially distinct step from the usage of the final results—although they may be colocated (as in *Copilot*). We observed several approaches to this task. Many systems have No Disambiguation mechanism and only present a single suggestion to the user (as in *Mage* [42]). Others show multiple candidate Code options, as in *Copilot's* alternate selection tab. Some preview the Effects of running the code, as in *VizSmith's* [6] multiple chart options. Our model also allows Summary information, which includes designs like surfacing token confidence [88] and textual [4] or semantic [85] diffs.

Refinement. The user may need to refine or adjust their specification after the initial gesture. For instance, a particular recommendation may necessitate that the user adjust their mental model of what can be expected from the assistant (as in the user-synthesizer gap [23]) and then update their prompt accordingly. Support for refinement defines to what degree the user can modify their initial specification. The most basic support is Restart, wherein the user needs to redo gestures to update their specification (as in *Observable Data Table* [65]). Some systems support a Commit action, in which the user can specify updates to the specification while the system holds previous gestures in state, and then the suggestion is updated after commit, perhaps via an "update" button—such as updating form fields in *Wrex* [21]. Other systems employ synchronous or Live (One Way) updates, in which the suggestion is updated immediately after the gesture, such as how *Mito* [19] updates the generated code immediately after interactions with its spreadsheet. Some systems, such as *Glinda* [18] and *B2* [93], extend this idea through Bidirectional updates [31, 32, 90], in which modifications to the output update the gesture input as well. While bidirectional

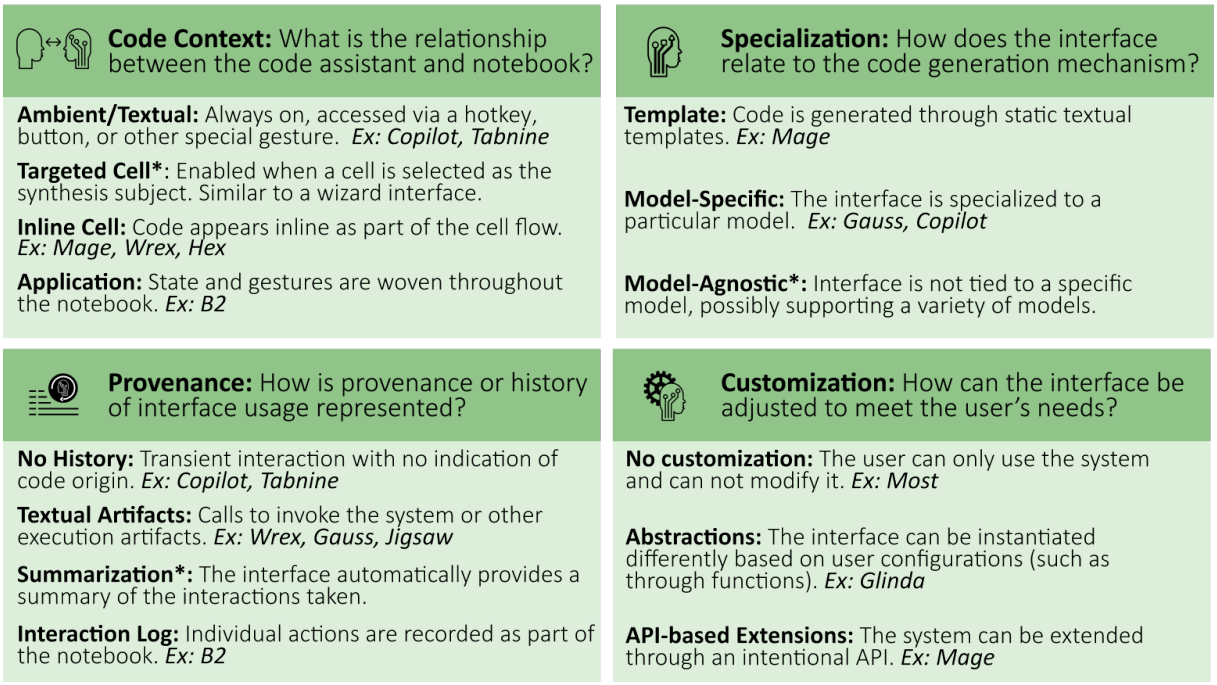


Figure 3: The parts of our design space considering the relationship between code assistant and other system components mediated by the interface. Our interview study considers the *Code Context* and *Provenance* interface relationships.

updates are ideal—as they maintain consistency between specification and model state—this approach is rare in practice due to the high complexity of synchronizing model, text, and output updates.

3.1.2 Interface Relationships. We next consider the parts of our space that describe the relationship between the code assistant interface and other system components.

Code Context. A key design choice for the code assistant is its relative location within the notebook UI. This location, in turn, suggests the scope of information available to the code assistant. We list these from the most localized context to the most global. In Ambient/Textual interfaces, the code assistant follows the text cursor, and the assistant is invoked via a special action such as a hotkey, as in *Tabnine* [83]. In Targeted Cell interfaces, the code assistant apparently resides within notebook cells to support editing or code generation directly within a given cell, as in Fig. 4. Code assistants presented as Inline Cells appear as a contiguous part of the notebook flow, sometimes as a distinct cell type of its own, as in *Mage* [42] or Fig. 5. Application-style interfaces appear integrated into the notebook at a level above the cells, such as through a side panel or other higher-level control, as in *B2* [93] or Fig. 6. The location of the assistant interface informs what information the user can provide to the system for code generation.

Specialization. Behind any code assistant is a mechanism for transforming inputs into code—such as an ML model, a search engine, an algorithm, or other heuristics—however, there are trade-offs between mechanism exploitation and the interface flexibility to accommodate different input modalities. For instance, in Model-Specific interfaces, the assistant is customized to assist the model, such as in the drag-and-drop interaction used to construct computation traces in *Gauss* [7]. While Template-based interfaces are constructed from simple static or textual templates (as in *Mage* [42]), which can enable a much wider array of input styles. Finally, Model-Agnostic assistants support a variety of different model types (perhaps even exposing that functionality to the end-user), however, this can come at the cost of more restricted input modalities, such as the text or tabular data input common to many models.

Provenance. The history of code generated by the assistant—such as what interactions led to the final generated code—can serve as the documentation for auditing, analysis, and sharing. Many systems provide No History of interactions with the assistant, as *Copilot* [25]. Some systems leave Textual Artifacts of their usage, such as the library calls used to initiate *Jigsaw*. Others included an explicit Interaction Log, such as *B2*'s [93] replayable history. Automatic Summarization of interactions can be used to describe the usage from a high level. The interface for displaying provenance may be contained in cells or may be baked into other parts of the interface—such that interactions are privately held in local state.

Customization. Enabling end-users to alter the code assistant interface allows the system to adapt to different users and tasks. Yet, most code assistants we examined had No Customization support. Customization can be achieved through Abstractions where the system provides modifiable abstractions (akin to functions) for end-users to customize the system, as in *Glinda* [18]. Similarly, API-based Extensions allow the user to configure properties of the

interface through provided APIs, as in *Mage* [42]. Creating tools with extensibility in mind may help the end-user avoid limitations but may complicate interface design.

Besides the above design considerations, there are other elements that vary among existing systems but do not necessarily alter the user experience. These include elements such as whether there is a secondary notation [10] integrated into a code assistant or the sorts of input modalities available to the user. Because notebooks and assistants are an active area of research and design [45], we forgo classification of such features rather than engaging in the quixotic task of enumerating all possible designs of such elements, instead seeking to provide a high-level description of the types of patterns that can occur in their intersection.

While the social roles that code plays (such as being proxies for trusted colleagues) do affect the types of features users perceive as being useful, we forgo modeling the social context surrounding these assistants. Such social interaction with and through code is its own rich topic of study—even just within data science [16, 22, 43]—and extends beyond the scope of this work. However, understanding the role that code assistants can play in those relationships is valuable future work.

3.2 Model validation

We validated our design space, by reflecting on its relationship to other systems and consulting experts on *Copilot*-style interfaces.

Code assistants are a relatively new form of interface, and innovation will continue. We have intentionally tried to construct our space in such a way that future designs may be located within it, but may not be predicted by it. That is, following Beaudouin-Lafon's [8] typification of design spaces, ours seeks to be descriptive (characterizing what has been done) and generative (prompting subsequent designs). We do not seek to be evaluative in our construction both because the space is evolving and so best practices may change with time, as well as because significant prior work has explored the role of evaluation in coordination with AI models (cf. Amershi et al. [3]). Other design spaces might be reasonably formed to describe this space given a different set of priorities. The analysis available through this space is one relating to the identification of similarities between systems and, potentially, the missing elements within a given system. For instance, FlashFill [28] can be described as an Ambient, Model-Specific interface focused on Code Text in a Nonlinear manner. As in most [90] spreadsheets, it has Live (one way) refinement. It provides No History and allows No Customization. Further, this system has No Disambiguation, an absence addressed in follow-up works [53, 63]. Despite this comparison, we only claim that this analysis covers the case of notebooks. Our design space is meant to dovetail with related analyses, such as Jayagopal et al.'s [37] notions of triggered vs triggerless initiation and result communication in code generators.

We presented our design space to several researchers with expertise with *Copilot*, who had run user studies themselves on the topic. These experts received the design space favorably and spoke positively about its value for the design of future systems in this genre. While expert opinion is inherently subjective, their review provides a coarse means of validation. We continue to explore aspects of our design space through our interview study (Sec. 4).

4 INTERVIEW STUDY

To investigate real-world perceptions of elements of our design space we conducted a semi-structured interview study. As the comparison of all design combinations is prohibitively expensive, we focus our study on four under-explored elements that complement prior studies [4, 38, 81, 88]. In particular, we focus on code context, disambiguation techniques, adaptation methods, and designs for provenance exploration. While we did not specifically construct designs relating to other aspects of our design space (e.g. expectations of customization), these topics arose organically in interviews.

4.1 Methodology

We recruited 15 participants (denoted **P1** - **P15**) from a large data-driven software company. Participants were contacted based on job title (including roles such as *data scientist*) through the company address book. We invited those who self-identified on an intake survey as having experience with notebooks and *Copilot* (or another AI-based code assistant) to participate in our interviews. Despite their response to the intake form, **P3**, **P9**, and **P15** did not have experience using an AI-powered code assistant. We include them in the study because their opinions, inclinations, and skepticism about such tools are also informative. Participants were compensated with a $50 USD Amazon gift certificate.

Participants reported a median of 1–3 months of experience with an AI-based assistant, 1–2 years in their current roles, 3–5 years of experience using notebooks, and more than 5 years of doing data science work. All participants were 25–34 years old and had at least master's degrees in a CS-related field. Further demographic data was not collected.

The semi-structured interviews, conducted via video conferencing, alternated between questions regarding participant experiences and presentation of design probes (Figs. 4-8). We used slide-based prototypes instead of developing interactive prototypes to avoid low-level feedback (e.g. implementation bugs or minor elements of visual design), following the methodology used by Kery et al. [42]. We did not collect quantitative scores on the presented designs because the semi-structured format of the interview caused some topics to be considered more deeply than others. Interviews lasted an average of 73 ± 10 minutes.

We recorded and transcribed the interviews. The first author open-coded the transcriptions and built a set of tentative themes, which the research team periodically met to discuss, critique, and iterate on until saturation was reached.

4.2 Study design

We next present the issues considered in our study and the various designs used to explore them. Discussion of these topics was guided by simple data analysis and visualization scenarios, such as merging two datasets. The details of these settings did not substantially affect discussion, so we forgo description here. The full study instrument is in the supplement.

Code Context. Each code generation model has its own notion of context that informs what information the model requires to generate code. For instance, in *Copilot*, context consists of a fixed length of text around the text caret. Complicating this specification

is that a user's understanding of code context may differ from the assistants, potentially yielding a mismatch between the provided and expected suggestions. Through consideration of this topic, we sought to understand: *what are users' perceived needs for context specification*? In our interviews, we considered three design options a Targeted cell-style option (Fig. 4's Wizard), a between-cell or Inline-style option (as in Fig. 5), and a notebook or Application-level option (Fig. 6's Side-panel). These designs form a spectrum of implicit (does not inform the user of the model's context) to explicit (allows the user to specify the code context) context designs.

Disambiguation. As described above, the ambiguities latent to specification incompleteness and model uncertainty can prompt some interfaces to produce multiple candidate solutions where the end-user needs to select a correct solution. Prior studies have indicated that disambiguation can be valuable [4, 48, 53], but there has been little consideration of the types of UI affordances that might support such work. To this end, we considered: *What types of disambiguation representation do users perceive as valuable?* We presented a sequence of design options guided by prior explorations, shown in Fig. 7. These included (A) an Output Only display inspired by *VizSmith* [6] (Effects), (B) a Code Only display inspired by *Copilot*'s multiple suggestions in a new tab feature [26] (Code), (C) a Code with Diff display per Barke et al. [4] (Code and (Summary), (D) a Code and Output display (Code and Effects), (E) a higher-level Summary View (Summary), and a paginated view (No Disambiguation) as in Fig. 5.

Adaptation. While contemporary AIs are powerful, their results are often imperfect [88] due to model limitations or ambiguous user specifications. Users often accept these imperfect solutions and then engage in an accept-validate-repair sequence [4] as a way to read, internalize, and adapt the generated code to the local interface. Here we examined: *How do users expect the interface to assist in the adaptation of generated code?* We highlight that *adaptation* is a different process than *refinement*, involving the use of generated code as opposed to getting the assistant to produce better code. We show design options that sought to elicit the adaptation strategies, as in Fig. 8, which variously surface Model Metadata and Disambiguation Summaries. These include (A) a linter-style UI that highlights potential semantic-level mistakes in generated code, (B) an option where the assistant generates a Snippet skeleton (instead of executable code) where the user is prompted to fill in missing items (per Barke et al. [4]), (C) a Token-confidence design that shows model confidence on different parts of the code, and (D) a Token-alternatives option that lets the user explore alternatives for a given part of the code. (C) and (D) are from Weisz et al. [88].

Provenance. A touted value of computational notebooks is their facility to interleave documentation and code in a literate manner [41, 80], which allows for description of code and design variations and explorations [91]. Barke et al. [4] suggest that interactions with *Copilot*-style interfaces are expected to be transitory, which is aligned with nearly every interface examined in Sec. 3—*B2* excepted. Disentangling these seemingly conflicting positions is an important step toward understanding how to develop tools in this environment. Here, we considered *What, if any, documentation of code Provenance of generated code do users believe is valuable?* Our designs examining this tension included an "autosummary" button that saves the interactions with a code assistant as either a markdown cell or as an inline comment, a cell-level marker to designate which cells contain generated code, and an explorable log that documents the interactions with assistants (per Kery et al. [40]). Figures showing these designs are in the appendix.

5 INTERVIEW STUDY ANALYSIS

We now reflect on the themes, topics, and suggestions that we identified in the interviews. We begin with a summary and then present cross-cutting themes (and return to the particulars of our UI results), ordered from most specific to most general. These include the role the UI design has on the perception and usage of features in this context (Sec. 5.1), the relationship that a code assistant in a notebook has with its surroundings (Sec. 5.2), and finally, the role that code suggestions have on trust (Sec. 5.3).

All participants believed it is valuable to adapt code assistants to notebooks. For instance, **P11** noted *"the use case for [code assistants] in notebooks is, I would say, pretty strong"*. **P4** saw *"a lot of potential where you can go with [code assistants] for the data scientists."*

Yet, for most topics we considered, there was no consensus about a single best feature over other design choices for code assistants. No one of the interfaces we described for code context was universally preferred, with each being liked or disliked according to participants' preferences for clutter, politeness, and locality. Participants nearly unanimously viewed Code and Output (Fig. 7D) as the best way to explore collections of suggestions, although some participants (e.g. **P4** and **P11**) liked the Summary View—likely for its novelty. Each of the presented refinement strategies—excluding Token-confidence—was seen as valuable for different reasons ranging from familiarity (**P13**) to transparency (**P4**). Artifacts of interactions with code assistants were seen as unnecessary, as code generation was not viewed as a part of their deliverable. In concert, this suggests a wealth of opportunities in this design space to serve differing desires and interests, as well as the wickedness [77] of the problem—given the multi-faceted and contradicting ways in which data scientists expect to use code assistants and notebooks.

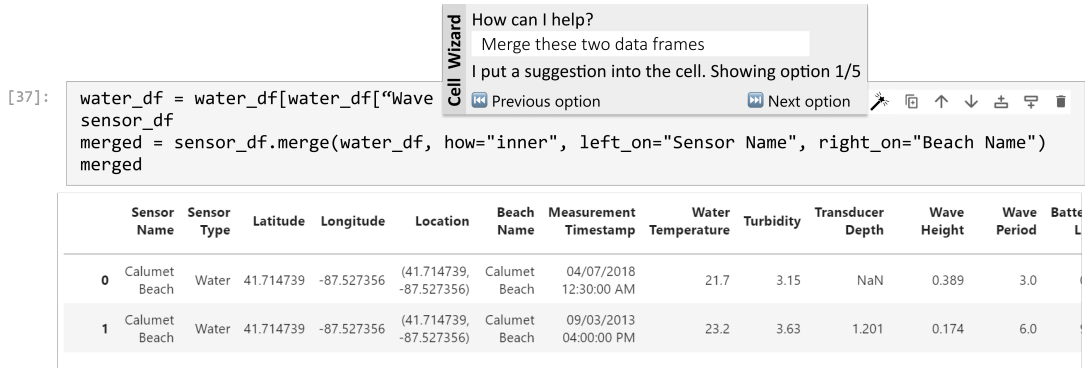
As notebooks are a subset of interfaces more generally, many of our findings align with similar expectations users might have of other contexts. However, our results are specific only to the context of code assistants in notebooks as understood by data scientists, as we only sought opinions from this type of user in this context. We reference these topics as code assistants and users, respectively, as a notational convenience, but they should be understood as these specific cases rather than general ones.

5.1 Role of UI Design

The way that users can interact with code assistants shapes what they attempt to do with those assistants. Here we consider how the design of the interface itself might influence different behaviors.

5.1.1 Context and Interfaces. In presenting several different modalities for interacting with code assistants, we sought to elicit perceptions about what and how much context participants believed would be useful for code generation tasks.

Most participants did not care about manual specification of context, typically expressing *"I want [the code assistant] to look at*



	Sensor Name	Sensor Type	Latitude	Longitude	Location	Beach Name	Measurement Timestamp	Water Temperature	Turbidity	Transducer Depth	Wave Height	Wave Period	Batte L
0	Calumet Beach	Water	41.714739	-87.527356	(41.714739, -87.527356)	Calumet Beach	04/07/2018 12:30:00 AM	21.7	3.15	NaN	0.389	3.0	
1	Calumet Beach	Water	41.714739	-87.527356	(41.714739, -87.527356)	Calumet Beach	09/03/2013 04:00:00 PM	23.2	3.63	1.201	0.174	6.0	

Figure 4: The Wizard interface (Targeted cell) included in our interviews surfaces control of a code assistant via a popover specific to each cell. The context used to inform the code generation is specified implicitly and is local to the targeted cell.

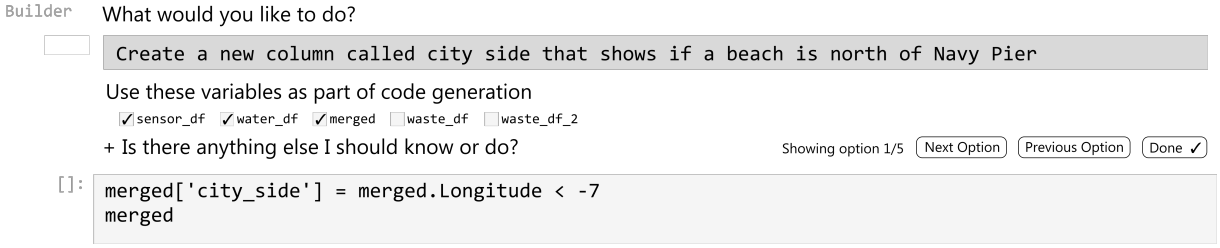


Figure 5: The Inline interface (Inline) places an assistant between cells, akin to any other notebook cell. Code generation context is formed by selecting variables that should be used to form the scope, however other strategies might also be used.

*everything," **(P8)*** rather than requiring manual inclusion or exclusion of parts of the code. Some participants valued having control over what goes into code generation (e.g. **P1**, **P6**, and **P7**), but those participants also had non-trivial knowledge of the underlying code-generating model. For instance, **P1** *"liked the idea of selecting variables for the code generation."* This group suggested explicit opt-in inclusion of particular cells (**P1** and **P6**) and giving priorities to certain gestures—such as library usage (**P7**)—might be valuable. **P6** and **P13** said context-free suggestions (akin to a search functionality) would be useful—echoing Barke et al. [4].

Some participants liked the Inline Cell design, as in Fig. 5, because it was familiar (**P1**, **P4**, **P7**, and **P15**) and immediate (**P2**) and believed it would allow them to keep a consistent mental model of the execution flow. However, **P10** and **P13** disliked Inline because they believed that it cluttered the notebook and would make it hard to keep track of interactions with the assistant. Others preferred the Side-panel (Application), as in Fig. 6, because of the ability to keep things separated. **P5** noted that *"I don't like to mess with my code...unless I'm sure"* about the changes to it. **P4**, **P5**, and **P13** noted that they tended to have large screens and so having a separate section of the notebook dedicated to interactions with code assistants was a good use of that space. **P4** analogized it to using an integrated Google-like search functionality. However, some (e.g. **P2**, **P8**, and **P15**) disliked the perceived ergonomics of needing to split their attention between areas of the notebook. **P14** and **P15** pointed to the Wizard design (Targeted Cell), as in Fig. 4, as being valuable because of its locality. Others were averse to the Wizard because of a dislike of popups: **P3**, **P4**, **P7**, and **P12** felt that it would *"break my train of thoughts every time" **(P4)*** or bring them out of *"the zone" **(P7)***. Only **P10**, **P11**, and **P14** brought up the autocomplete style found in *Copilot* as being a desirable alternative to the designs we presented—although **P11** specifically noted that he would prefer them to be used in conjunction, as they served different purposes. For instance, **P11** and **P15** seemed to think of creating visualizations as a different process than other forms of coding, **P15** noting that *"it doesn't add value to your final product, like data wrangling [does]."* This suggests that there are numerous design opportunities to address tasks that users do not view as central to their contribution to the work.

Many participants (**P2**, **P4**, **P6**, **P8**, **P11**, **P13**, and **P15**) noted that their usage of a code assistant was not motivated by a desire to learn or explore (although **P11** and **P14** espoused this view), but simply increase their efficiency. This focus on efficiency is in tension with some tasks at the heart of notebooks' literate programming style. One such example is data analysis, which may require alternating modalities to investigate correctness, such as interacting with a chart or spreadsheet to determine if a data manipulation has been executed correctly. Features intended to support exploration may be able to take greater liberties in their designs (which **P3**, **P7**, and **P11** noted about Fig. 7E's summary view), while those focused on acceleration require a shorter and less-noticeable interaction loop.

Evidently, not all designs will work for all users. It may be beneficial to surface multiple places in the interface where the user can interact with code assistants—akin to how spelling/grammar checkers are often surfaced through both a triggerless [37] ambient mode as well as a mode triggered from a menu.

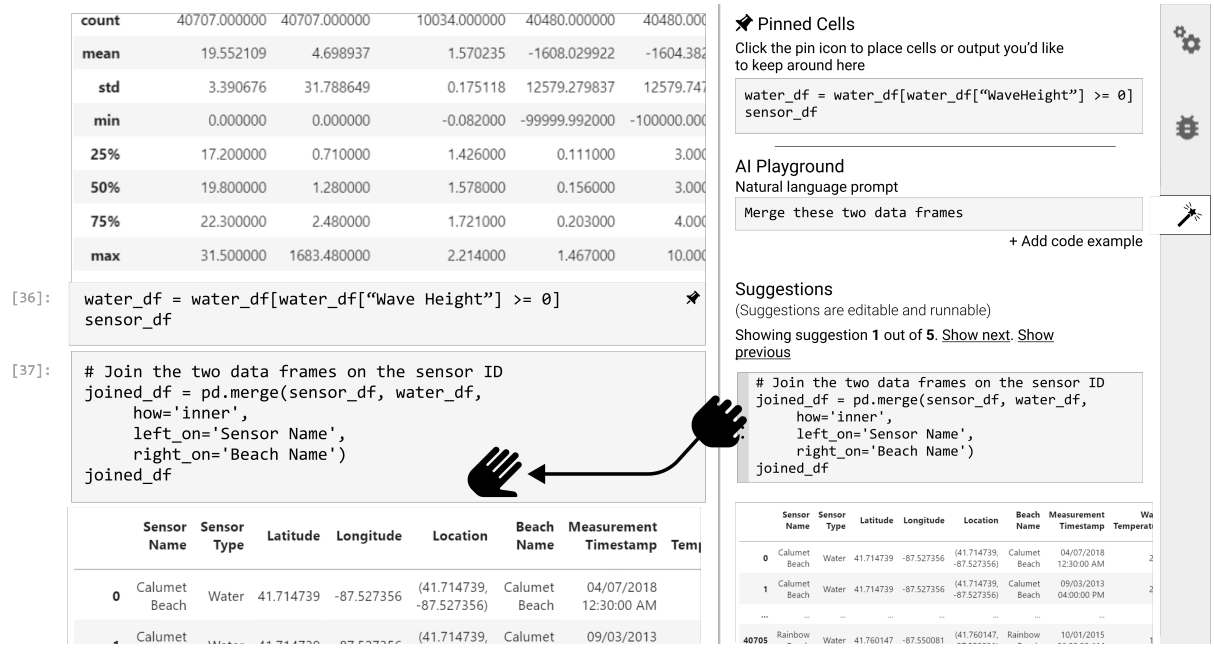


Figure 6: The Side-panel interface (Application) paradigm sits outside of the notebook flow. It provides a sandbox to explore suggestions before dragging them in. Context is described explicitly by pinning cells or through code or natural language.

5.1.2 Feature Prominence. A paradox of interface design is how to make something discoverable—and easy to remember—without making it annoying. Demo or novelty-driven design can be helpful for buy-in but can make interactions with those systems frustrating in practice—as was the infamous case of Microsoft Clippy [15].

A recurring issue in the interviews related to how prominent the controls for a coding assistant were. A feature that is too prominent might be *"irritating" (**P15**)*, while one that is too subtle will be *"undiscoverable" (**P9**)* or unlearnable. A feature being prominent means that it is more likely to be discovered but also potentially that it will be disabled (**P15**). **P3** valued prominence, noting it prompted him *"to pay attention to this new feature,"* whereas features that do not announce themselves may not get found. For instance, only **P1**, **P7**, and **P10** were aware of *Copilot*'s alternatives menu [26]. **P9** noted that he preferred the Inline design because it was easier to discover compared to the Wizard or Side-panel (an advantage shared by triggerless initiations [37]), whereas **P4** noted that he preferred the Side-panel because it could more easily be ignored when it was not relevant. Yet, triggerless systems are not appropriate solutions for all cases: **P11** commented about *Copilot* that it was *"frustrating for me to kind of say, like, No, I don't want this right, because it kept on trying to auto guess"*—a position also shared by **P13**. UIs that abuse triggerless or eager presentation may poison the well for users. For instance, **P15** noted that every time *"I see an attempt from an interface to help me, I just disable it,"* having had negative previous experiences. **P7** and **P15** suggested that interfaces that may sometimes be annoying (referring to Fig. 7C's diff view and Fig. 8C's token-confidence view) might be more usefully shown on demand rather than all of the time, that is, as lenses—echoing Kery et al.'s [41] recommendations for debugging or history tasks in notebooks. This style of lightweight opt-in feature may be useful, however, this may make that feature all but invisible. We suggest then that navigating the tension between being polite [89] (by respecting user agency) and promoting usage (such as finding opportunities for usage that could be missed otherwise) is central to the design of effective code assistants.

Familiarity with and novelty of particular features were powerful factors in participants' expected usage patterns. **P9** discovered the snippet search functionality in his editor by using a search engine to investigate if *"there is anything like [snippet search] by PyCharm?"* Jayagopal et al. [37] note that this is a factor for novices, but the presence of such concerns among our participants suggests it may be more universal among notebook users. For instance, **P6** noted that his use of Token-confidence displays (Fig. 8C) in a past project biased him toward them. **P5** liked the output-only (Fig. 7A) displays because they reminded her of related features in Excel. Such expectations are closely informed by the vocabulary of features found in other notebooks (cf. Sec. 5.2.3). The summary view feature (Fig. 7E) was sometimes viewed as intriguing (e.g. **P2** and **P3**), however, that may be due to novelty. Novelty and breaking expectations in surprising ways can be beneficial and lead to magical feeling experiences—although, as prior studies on *Copilot* have shown, the magnitude of this improvement may be overestimated by end-users [35, 84]. Participants observed that novelty can have value (in that it can provide an incentive to explore or interact with a feature), but the *"value of [the feature] has to outweigh the novelty" (**P11**)*. We suggest that, as with humor [86], expectation can be subverted to useful and surprising effect, however, such manipulations can be seen as annoying and lead to feature disuse.

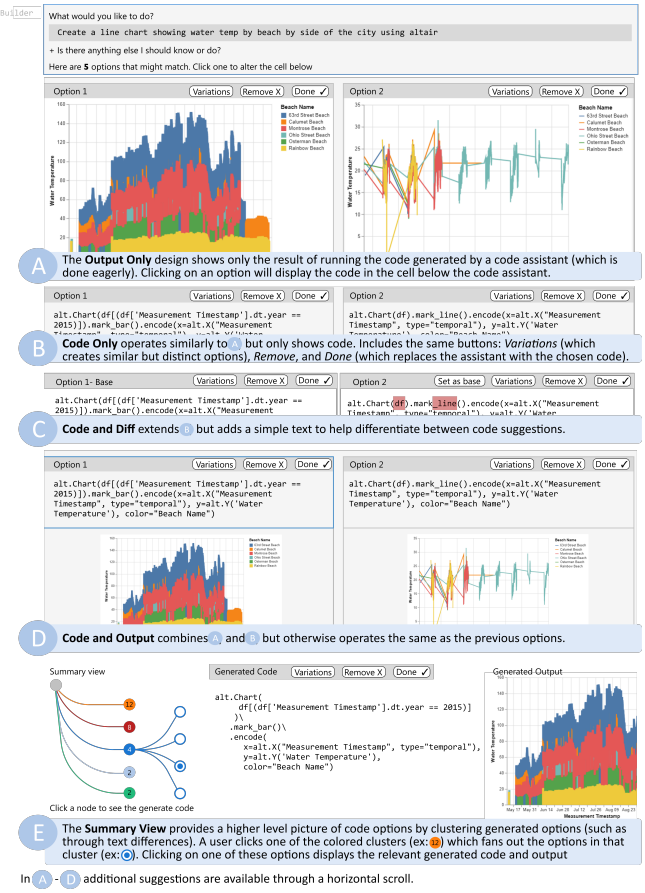


Figure 7: We presented a variety of options for disambiguation covering combinations of Code, Effect, and Summary strategies. These options could be applied to other styles besides Inline (as here), such as Fig. 4 or Fig. 6.

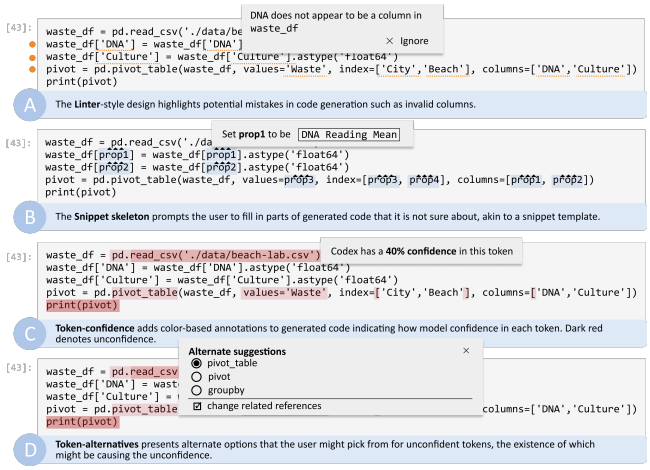


Figure 8: The four designs presented to participants in our study for adapting code generated by an assistant to the task or domain.

5.2 Relationship with surroundings

Notebooks do not exist as solitary objects. They serve a broad range of purposes, including exploration and experimentation (**P4**, **P7**, and **P9**), an environment for development before formalization into a script or pipeline (**P4**), presentation or report (**P3** and **P13**), as well as both shared (**P5** and **P7**) and solitary objects (**P2**, **P6**, and **P8**). We found that being situated in such usages informs the desired UI affordances, such as for contextualization in general and domain-specific enhancements particularly.

5.2.1 Provenance. Participants did not view documentation of interaction with code assistants (Provenance) as important. For instance, **P1**, **P2**, **P4**, **P5**, **P8**, **P9**, **P10**, **P11**, **P12**, and **P15** noted that they only cared about the quality of the output and not how it was achieved, with some participants (**P3**, **P5**, **P6**, **P8**, **P9**, **P11**, and **P13**) adding that data science is an output-driven discipline. **P13** noted that data scientists *"spend more time thinking about insights, thinking about strategies, rather than thinking about how to code,"* highlighting that tasks like documenting code provenance are not seen as essential to their role.

Some participants viewed our documentation designs as only useful in untrusted situations (**P2**, **P5**, **P8**, and **P13**), which were uncommon in their work. For instance, **P2**, **P5**, **P7**, **P11**, and **P13** noted that they trusted any notebooks presented to them by a colleague and thus would not need to audit the code's origin.

P1, **P4**, **P5**, and **P13** compared documentation of code generation as being similar to including attribution to Stack Overflow posts from which they may have copy-pasted code. **P1** observed that *"Stack Overflow is like a really slow coding assistant"*. Most participants did not attribute code gathered from other sources because they did not view it as something worth documenting—unless it was particularly unusual code or they might not have been expected to know it (**P14**). **P8** and **P15** expressed skepticism for the utility of any attribution at all, with **P8** questioning *"What is human's memory and creativity? How did you know whatever you write is even your original thing or if it exists somewhere else?"* These reservations agree with prior observations [22] that data scientists tend not to see code as a deliverable.

While hesitancy about documentation may be surprising—in light of the well-known [41, 80, 91] and well-discussed (**P1**, **P2**, **P4**, **P7**, and **P8**) literate programming paradigm in which notebooks operate— this disinterest seems to suggest that there are different categories of work in notebooks: that which is meaningful to the deliverable and that which is only necessary for its construction. For instance, **P9** suggested that documentation of interactions with code assistants would be something akin to recording or publishing the undo/redo stacks associated with typing. Prior work [4, 81] suggests that documentation of interactions with code assistants is not seen as valuable, and we extend this to suggest that it may be beneficial to specifically align the design of code assistants so that they fall into the category of work that is not viewed as significant to document because, by being ignorable, they can more easily become commonplace—as is the case with spell checkers. That is, they should be treated like a ghostwriter.

5.2.2 Search. It has been argued that integration of search with code suggestions improves the facilities of code assistants. For instance, Xu et al. [94] found that code search enabled different categories of tasks, such that users employed search for larger or more complex pieces of code and synthesis for small simple modifications. Similarly, prior work has explored integrating search into IDEs [11], as well as more specifically in notebooks [49, 50].

Our participants (e.g. **P4**, **P6**, **P8**, **P13**, and **P14**) espoused a similar set of desires, describing that code search may be a valuable addition to code assistants. This is in line with how some users already expect to use code assistants [81], for instance, **P4** noted that he already tended to think of *Copilot* as a search system. **P9** spoke at length about the value of being able to search against a fixed library of snippets—although this library also might be usefully made malleable. For instance, **P13** wanted to be able to save snippets (such as her preferred manner of cleaning categorical data) and have them be suggested later. **P8** and **P14** wanted to be able to access elements that were outside of the model's training data, such as code written by their teammates or others within their organization, but have it still be adapted to context rather than shown in a decontextualized way. As **P3** noted that *"if I know my code is going to be shared across the team, I usually tend to write a lot of comments"* (echoing prior work [22]), code search with automated integration may reduce the need for manual documentation.

Yet search alone was not sufficient: it seems that it is necessary to also provide context or reasoning for code suggestions. For instance, **P5** desired that context should be integrated into each suggestion, such as through library documentation. **P9** suggested a similar desire for *"documentation for each [of the suggestions], then you can look into it like, what are the pros and cons of it."* **P2** and **P5** desired a way to see what parts of the training data influenced each code suggestion to get a better understanding of how that code was used—a feature commercialized in CodeWhisperer [2]. Prior work [78, 84] also has recommended library documentation be included as part of code recommendation, which is an instance of the Human-AI guideline that systems should "make clear why the system did what it did" [3].

We suggest that integration of both search *and* documentation into code assistants is valuable as it can help users find code and validate it. We note that these contextualizations (or similar Model Metadata) may increase automation biases [84] and move users to be even less questioning [56] of suggestions.

5.2.3 Relationship to domain. Critical to any usage is the context or domain in which that work is done. We identify several ways in which knowledge of the domain might aid tool design.

We found that participants' expectations were guided by the makeup of the data science ecosystem. For instance, analysts often [22] conduct analyses in environments in which they do not have full control, like Databricks, in order to use protected resources, like Spark clusters. Such systems often provide their own variant of notebooks with their own collection of UI affordances drawn from a relatively limited vocabulary of allowed interface forms in notebooks [45]. While **P14** noted it can be easy enough to port from one system to another, **P13** commented that she uninstalled *Copilot* after months of usage noting that *"I don't want to be too reliant on it,"* as it was not available in all of the environments in which she needed to do work. A motivated user might adopt new interaction forms specific to a particular environment, but, as **P15** argued, users whose key concern is efficiency may quickly dismiss things they perceive as hampering their process. Thus code assistants may be more likely to be adopted if they follow or are in dialogue with familiar patterns. We suggest that it may be useful to build code assistants that operate on a browser level rather than a notebook level to take advantage of the necessity of moving between browser-based notebook environments.

As code assistants generate code, it is natural to consider how they relate to best practices and code styles. Some participants (**P6**, **P10**, **P13**, and **P15**) thought that it would be better if assistants adapted to their style and learned their preferred way of doing things over time. In contrast, **P14** noted that he would not want a system to adapt to his style because *"I don't want to stay in the bubble."* Similarly, **P12** noted a preference to exert control over *"naming convention and those kinds of stuff."* Others (**P5**, **P7**, **P8**, **P9**, and **P11**) thought that generated code should strive to follow best practices and match the conventions of their team. **P4** believed that code assistants helped enforce and teach best practices more effectively than traditional assistants such as linters or auto-formatters, observing *"for me, Copilot is just an evolution of these tools."* This is related to Sarkar et al.'s [81] observation that some users believe that *Copilot* helps with best practices. It may be useful to capitalize on this expectation of offering best practices as a way to integrate opinionated advice, although this should be done cautiously as automation bias [84] may reinforce negative behaviors.

A feature of frequent interest (**P1**, **P3**, **P5**, **P7**, **P9**, and **P10**) was a *linter for notebooks*. Such a system could assist with common data science errors such as out-of-order execution (**P1**), side effects (**P10**), and identify opportunities to convert frequently rerun cells to functions (**P5**) in a manner analogous to how spell checkers offer suggestions on grammar or usage errors. Notably, this typically came up prior to discussion of Fig. 8A's linter-style design for adaptation. A common observation was that code in notebooks tends to be lower quality—*"I guess I'm lazier when I'm using a notebook" (**P5**)*—suggesting that an ambient design intervention, like a linter, agrees with typical usage patterns in notebooks. However, this UI style can be seen as annoying if it does not provide consistent utility: *"sometimes linter just complains, it just says, too many lines or too many variables within that function" (**P7**)*. In addition to providing basic notebook usage hints (such as those highlighted by Rule et al. [79]), it may be useful to support domain-specific practices such as schema awareness (**P1**, **P5**, **P12**, and **P14**)—like highlighting when a column does not exist on a dataframe or when a particular method is slow compared to a vectorized alternative—or best practices local to data science—like ensuring that data fitting uses a train-test split pattern. Domain-specific linters have been created in other domains—including visualization [13, 57, 92], spreadsheets [5], and ML data [34]—suggesting the value of lightweight domain or medium-specific ambient assistants for data science in notebooks. While not addressing every domain concern, such a lightweight assistant may reduce the need to specifically adapt more-powerful coding assistants to notebooks.

5.3 Role of trust and control

We found that the relationships that participants have or expect to have with code assistants are closely mediated by their sense of control—how they understand what it does and their ability to accept or reject suggestions. These aspects are mediated by perceived quality of output (including correctness and readability) and knowledge (or perception) of the underlying model.

5.3.1 Points of control. The primary point where users can exert agency in our *specification-refinement* loop is through gesture and interpretation of output. **P1** noted that *"being in control of the code"* felt essential to trust in the tool.

We found that a critical point of control is the way that participants expected the assistant to integrate into their workflow. Barke et al. [4] observed acceleration and exploration modes in *Copilot* usage in which users either used it as a way to simply type more quickly or to try to more generally understand how to do something. Participants (e.g. **P1**, **P2**, **P7**, **P8**, **P10**, **P14**) believed they only used assistants like *Copilot* to increase their efficiency (akin to advanced autocomplete). Participants did not directly self-describe that they used *Copilot* to explore alternatives, for instance, **P8** noted that he did not *"because I don't really trust it"* to know uncommon functions or libraries. **P5** observed that type or syntax-based autocomplete engines found in many editors are sufficient for exploration of libraries and function parameter values. Yet some participants did seem to value the results of using code assistants for exploration. For instance, **P2**, **P4**, and **P9** described it as a way to help them learn to do new things, while **P6** and **P14** observed that it helped them learn new coding patterns. Our findings thus comport with Barke et al. [4] regarding bimodal usage, however, our view of this behavior is more limited as we only have participant self-reports rather than direct observations.

Some participants wished to be able to control certain parameters (Model Metadata) about the assistant, such as *"suggestion length" (**P8**)* or *"adventurousness" (**P1**)* (i.e. more exploratory or less straightforward suggestions, akin to increasing the model temperature). **P12** expressed an interest in being able to swap between models (Model-Agnostic) if he could see an automated report of his use of each model, while **P11** desired a toggle to indicate whether or not the system should try to help learn rather than merely complete the task. While this sort of configuration can be valuable, many participants noted that it (and other presentations of choices) can be overwhelming (**P2**, **P7**, **P8**, and **P13**). Hiding these choices in an expert menu may reduce this burden, but it also may cause those elements to remain undiscovered (Sec. 5.1.2), even by those who might wish to use them.

Despite the seemingly static nature of code, its role as a component of an editor is inherently dynamic and, in this context, involves both reading and adaptation as points of control. For instance, **P1** noted that he felt out of his control when he could not understand generated code (e.g. when *Copilot* suggests hard-to-understand one-liners). **P3** expressed some hesitancy about generated code more generally: *"I'm not like somebody who can only read code. I do write code and I do want to have the control."* Drosos et al. [21] and Kery et al. [42] highlight the importance of synthesizing readable code in contexts like data wrangling, while Sarkar et al. [81] make similar observations specifically related to *Copilot*. Following Weisz et al. [88] some participants thought that code annotations would help them better read and understand suggestions. For instance, **P5** noted that *"reading someone else's code is always harder"* and that design interventions such as those presented in the study would *"definitely be helpful to read [generated] code."* However, **P3** and **P9** thought annotations, such as diffs or Weisz et al.'s [88] designs, would be distracting. Participants described a variety of adaptation strategies, including rewriting the prompt (or just continuing to type in the case of triggerless systems like *Copilot*), fixing the output (following Barke et al.'s [4] accept-validate-repair sequence), or simply completing the task manually—although the use of these strategies was mediated by experience with the language (**P5**) and estimated time to fix the errors (**P4** and **P12**). Our interviews explored design interventions that might augment these strategies, although participants were divided on which particular flavor would be valuable. For instance, **P1**, **P4**, and **P10** liked the Snippet skeleton approach because it felt like they were able to exert agency over boilerplate, with **P4** noting that *"For me, the code skeleton is the best one because you are being transparent and honest."* **P5** and **P13** liked the linter-style approach because it was familiar from other programming contexts. **P9** found numeric representations of confidence to be confusing (as in Fig. 8C's token-confidence view), noting that *"that's not useful to me. For me, it's either 100% or zero."* This potentially disagrees with prior work, which found that numeric heuristics as being associated with algorithmic intelligence [51]—although this may be due to our participant population, data scientists, who may have higher computational literacy than other groups. We suggest that effective code annotation in this context should either be ambient (ignorable unless needed) or provide value for active interaction (as a dedicated input mechanism).

5.3.2 The effects of knowledge. We observed that the process of understanding the capabilities of a code assistant is one of forming a relationship. Participants with prior relationships with such models bring with them expectations that in turn inform their usage.

Participants with experience with LLMs claimed that they would be tolerant of bad answers. **P8** and **P14** noted *Copilot* was only useful for problems that were well represented in the training data. **P8** highlighted them as being limited to popular APIs (such as NumPy or pandas) and that more esoteric topics or APIs that change rapidly tend to induce incorrect results when using *Copilot*—a point also noted by Sarkar et al. [81]. Similarly, **P12** believed that *"I can tolerate its wrong answer. But I may generalize the type of mistake that it makes, like, for example, if it's bad at generating graphs."* Participants with a less clear understanding of the systems underlying *Copilot* believed their reactions to poor performance would be more severe, noting that after two or three bad suggestions that would probably turn it off (**P3**, **P9**, **P11**, and **P15**) or *"if it's not too overbearing, I might just ignore it" (**P5**)*—highlighting the potential advantages of polite triggerless designs. It seems that these first impressions are an important part of developing the working relationship with the assistant: if value or trust is not clearly established either through preexisting assumptions or early behavior, then users may be unlikely to continue to try to use it.

Yildirim et al. [96] note that if designers have at least a limited understanding of what AI can do, they are better able to use it as a design material. Lau [46] highlights a similar guideline in the

design of effective synthesis tools. Ferdowsifard et al. [23] observe the "user-synth gap" in which it is not clear what the synthesizer can produce, which can cause an impedance mismatch between user behavior and expectation, which is a case of the Human-AI guideline [3] to clearly communicate system abilities. We suggest that similar principles may apply to users of notebook-based code assistants, as giving users a model of what is going on under the hood will likely shape their approaches to understanding and using generated code and affordances therein.

5.3.3 Verification. Even if a provided solution looks right, it may need to be examined to ensure its correctness. Participants described how the need for verification is tempered by how high stakes the code is, how long it will take to run, and the perception of its quality.

In addition to code reading, participants also described employing strategies like consulting documentation as a way to verify suggestion correctness. For instance, **P6** noted that *"I still Google to verify its correctness"* in cases when it would take multiple minutes to establish whether or not a system worked correctly. **P8** noted *"I mean, I always double-check"* when using an uncommon API, such as for creating ML pipelines. **P2** and **P14** desired validity checks akin to popularity markers in other settings. Some noted that having a low friction way to explore suggestions (as in Fig. 6's sandbox) before accepting them would enhance their trust in the model: *"I think it will make me feel more comfortable with...trying out longer suggestions and accepting it" (**P8**)*. **P1** added that such features *"make me feel safer."* Others (**P7**, **P9**, and **P10**) felt that this was unnecessary and could be achieved through normal cell usage.

A common means of verification explored in our interviews was through the exploration of alternative suggestions (i.e. Code or Effects disambiguation). This allowed them to explore variations on different approaches to different tasks, such as visualization or particular activities like identifying *"different ways to connect to the blob," (**P9**)* and in doing so verify the output of the assistant. Valuing browsing agrees with prior observations of *Copilot* users [4] and notebook users more generally [50], in that foraging can lead to the discovery of new functionality or ways of doing things. Among the designs we presented to facilitate this type of task, participants mostly preferred the Code and Output (Fig. 7D) for both visualization tasks as well as other situations, noting that it allowed them to *"be your own audit" (**P1**)* by checking code and output as task required. **P5** noted that it was valuable because it reminded her of Excel's chart chooser feature. **P10** and **P14** worried about the computational resources required to generate alternatives and suggested a hybrid "à la carte" execution mode in which only selected options were run. While trust in a system was noted (**P2** and **P6**) as being mediated by how fast it could supply an answer, **P6** and **P10** noted that if the system would produce high-quality charts, they would wait multiple minutes. The Token-alternatives design from Weisz et al. [88] was seen as a valuable mechanism for disambiguation. **P12** noted that it allowed them to apply *"minimum mental effort at learning to make adjustments"* to the generated code, and **P5** added that they are *"super useful if I wanted to explore things."* **P11** suggested that it would be helpful to have documentation integrated with the alternatives. This suggests that multiple modalities of disambiguation may be valuable for both acceleration tasks (as in the relatively lightweight Token-alternatives) as well as exploration tasks (as in the heavier weight Code and Output). Providing means of verification integrated into the process of code generation (such as Effect previews) may shorten the accept-validate-repair sequence [4], particularly when tuned to different usage modes.

6 DISCUSSION

In this paper, we explored the design space of interfaces for AI-powered code assistants in notebooks. To do so we investigated the design space of this style of tool through an analysis of preexisting systems that support code generation in notebooks. We sought to understand perceptions of several key elements in this space (context, disambiguation, adaptation, and documentation) for a realistic user population of notebooks. This led us to conduct a semi-structured interview study with 15 professional data scientists.

Through this work, we delineated guidance for designers of future systems in this space, which we summarize in Table 1. Participants were unanimously enthusiastic about adapting *Copilot*-style code assistants to the notebook domain. While not every design was to every participant's taste, it seems that there is ample space to introduce new valuable designs *in addition to* the Ambient style of *Copilot*. We suggest, to this end, that the most fruitful ground in this regard lies in creating systems that are specific to a domain task or those tasks things that users do not view as core to their work. For instance, writing code that is well formatted or follows best practices, or making visualizations. Within such tasks, disambiguation seems to be an especially powerful means by which to exert control over notebook-based assistants. This may be because the execution loop in which notebooks operate closely mirrors that of the specification-refinement loop. We suggest that navigating these elements in a polite [89] and non-irritating manner is essential for assistant adoption. Ambient interfaces offer a low-friction way to provide recommendations; however, their generality can lead to opaque usage and a lack of domain specificity. If such issues are key, then a more explicit design is preferable.

6.1 Limitations and Generalizability

As with any study, our studies have limitations that affect their generalizability.

Our design space and interview studies are limited by their predication on design paradigms used in Jupyter and similar ecosystems. While these patterns are quite common [45] (and have been stable for at least a decade), we do not utilize their ubiquity and commonalities with other systems to generalize beyond code assistants in notebooks. While some of our observations may be aligned with other domains (such as assistants in IDEs), we emphasize that our findings are local to notebooks, as investigation of another interface may have led to different findings.

Our design space was limited by our exclusion of search-based systems. Despite their relevance and long history [52], they fell outside of our focus on systems that generate code. While Sarkar et al. [81] found generative models are sometimes viewed as being similar to search, they also found that diverge in key ways, such as how search yields mixed-media, while generation gives fixed-media. Future work should investigate the similarities and

Table 1: Takeaways from our interviews. While some elements are always valuable (e.g. polite interfaces [89]), these findings are only grounded in code assistants for computational notebook-based for data science tasks.

Politeness	1. Code assistants should be designed so that they are treated as ghostwriters	(Sec. 5.2.1)
	2. Assumption of best practices enforcement can be used to provide opinionated guidance	(Sec. 5.2.3)
	3. Code annotation should be ambient or provide value for active interaction	(Sec. 5.3.1)
Notebook Patterns	4. Surfacing multiple ways to control context is useful—Ambient alone is likely insufficient	(Sec. 5.1.1)
	5. Expectation can be subverted to useful and surprising effects	(Sec. 5.1.2)
	6. Following familiar notebook UI patterns is important for adoption	(Sec. 5.2.3)
Code Assistant Patterns	7. Integration of search and docs is valuable, but code provenance is not—No History	(Sec. 5.2.2)
	8. Task (or medium)-specific assistants for data science in notebooks are valuable	(Sec. 5.2.3)
	9. Knowledge of the underlying model changes expectations and required affordances	(Sec. 5.3.2)
	10. Tools like disambiguation (e.g. via Code *and* Effects) can aid the *specification-refinement* loop	(Sec. 5.3.3)

differences between expected interface affordances between search and generation, so as to better understand how those interaction forms might be more effectively blended [94].

The slide-based prototypes advantageously allowed for exploration of many different designs but lacked the concreteness of interactive prototypes, which may have elicited different responses had we used them instead. For instance, some participants (Sec. 5.1.2) discussed how annoying a given feature might be and speculated on how that would affect their usage. While user perception of utility can be elucidative of their actual behavior (per the Technology Acceptance Model [17]), it is not necessarily reflective of their real-world usage. We sought to address this issue by requiring that our study population have previously used a code assistant so that they could draw on their real-world experiences to shape their expectations of code assistants. However this measure can only approximate real-world usage, and future work should study the experience of interacting with similar designs in live implementations. Similarly, the simplicity of the tasks presented may have limited the types of responses and a more complex task type may have elicited other responses. While a number of respondents (e.g. **P1**, **P4**, **P6**, **P8**, **P9**, **P11**, and **P14**) reflected on their experiences using AI assistance for complex tasks and how the presented design might interact with those tasks (covering topics such as modeling, data pipeline creation, and other NLP tasks), future work might usefully explore the perceptions of our design space in the context of more nuanced tasks. Our slides may have caused participants to be overly optimistic about our designs, such as the common out-of-order execution issues found in computational notebooks—although this was not always the case, for instance **P14** worried about the bugs that might be introduced by the sandbox in Fig. 6. Further, the consistent order in which we presented designs may have biased participant responses, however, we believe this effect is limited, given the minimal agreement on preferred features.

While work in notebooks is sometimes seen as synonymous with data science (**P13**), this is not always the case. Our interviews focused only on a single type of user which may impede the transferability of our findings to other tasks or categories of work. Users who mostly complete engineering (a divergence noted by **P8**) or creative tasks likely have different usage patterns. Future work should investigate the beliefs and expectations of other user types.

6.2 Future work

Notebooks are but a single medium among a vast array of other areas of application. For instance, how code generation should be adapted to end-user tools, like spreadsheets, is an important open question. Ragavan et al. [72] explored weaving natural language prompts into spreadsheets, finding that they can be successfully used by spreadsheet users, suggesting the value of studying such intersections. Other important application areas that balance graphical and textual specifications (to different proportions) include creative coding systems [70], low-code analytics environments that interweave graphical components with analytics queries (such as visual analytics systems like PowerBI or Ivy [55]), as well as systems that preference graphical representations, such as block-based systems (e.g. Scratch [75]).

Our interview study considered design probes that investigated a limited cross-section of the large number of possibilities described by our design space. Future work might consider the design permutations implied therein. This might usefully involve explorations about the role of Nonlinear input systems, such as the code-generating spreadsheets of *Mito* [19], or bidirectionally synchronized projectional-editors to handle thorny interface tasks, such as visualization annotation. Further, it may be advantageous to consider code assistants who only output metadata, such as code explanations.

The space of possible functionality available to assistants in notebooks is vast. The mixture of code, data, and graphics presented interactively offers a rich space of opportunities in which an assistant might be usefully interwoven. As code assistants continue to become more powerful, we highlight that careful adaptation to computing environments is critical, as is paying attention to the space of available design alternatives—particularly concerning context control and disambiguation. Models powering these assistants might usefully be designed with the end-user interface in mind, and facilitate features like search, context-free suggestions, recommendation explanations, and lightweight configuration so that they might be tuned to task on the fly.

ACKNOWLEDGMENTS

We thank our anonymous reviewers and our study participants. In addition we thank Christian Bird, Thomas Zimmermann, Elsie Lee-Robbins, as well as the Microsoft Research VIDA and EPIC teams as a whole for their useful advice and support.

REFERENCES

[1] 2001. History of code completion. https://web.archive.org/web/20220302082225/https://groups.google.com/g/comp.compilers/c/fJHahKDCNGg?pli=1. Accessed 11/11/22.

[2] Amazon. 2022. Amazon CodeWhisperer Features. https://aws.amazon.com/codewhisperer/features/. Accessed 8/16/22.

[3] Saleema Amershi, Daniel S. Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi T. Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Conference on Human Factors in Computing Systems (CHI)*. ACM, 3.

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. *arXiv preprint arXiv:2206.15000* (2022).

[5] Daniel W Barowy, Emery D Berger, and Benjamin Zorn. 2018. ExceLint: automatically finding spreadsheet formula errors. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–26.

[6] Rohan Bavishi, Shadaj Laddad, Hiroaki Yoshida, Mukul R Prasad, and Koushik Sen. 2021. VizSmith: Automated Visualization Synthesis by Mining Data-Science Notebooks. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 129–141.

[7] Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: program synthesis by reasoning over graphs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[8] Michel Beaudouin-Lafon. 2004. Designing interaction, not interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*. 15–22.

[9] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the ACM Conference on Fairness, Accountability, and Transparency*. 610–623.

[10] Alan Blackwell and Thomas Green. 2003. Notational Systems–the Cognitive Dimensions of Notations Framework. *HCI Models, Theories, And Frameworks: Toward An Interdisciplinary Science. Morgan Kaufmann* (2003).

[11] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Conference on Human Factors in Computing Systems (CHI)*. 513–522.

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

[13] Qing Chen, Fuling Sun, Xinyue Xu, Zui Chen, Jiazhe Wang, and Nan Cao. 2021. Vizlinter: A linter and fixer framework for data visualization. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2021), 206–216.

[14] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. (2021). https://arxiv.org/abs/2110.14168

[15] Michael Correll. 2021. The Clippy-ization of Human-Computer Design. https://mcorrell.medium.com/the-clippy-ization-of-human-computer-design-c66e8042de88

[16] Anamaria Crisan, Brittany Fiore-Gartland, and Melanie Tory. 2020. Passing the data baton: A retrospective analysis on data science work and workers. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2020), 1860–1870.

[17] Fred D Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* (1989), 319–340.

[18] Robert A DeLine. 2021. Glinda: Supporting data science with live programming, GUIs and a Domain-specific Language. In *Conference on Human Factors in Computing Systems (CHI)*. 1–11.

[19] Jacob Diamond-Reivich. 2020. Mito: Edit a Spreadsheet. Generate Production Ready Python.. In *LIVE: Workshop on Live Programming*.

[20] Brendan Dolan-Gavitt. 2022. FauxPilot - an open-source GitHub Copilot server. https://github.com/moyix/fauxpilot. Accessed 9/7/22.

[21] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Conference on Human Factors in Computing Systems (CHI)*. 1–12.

[22] Will Epperson, April Yi Wang, Robert DeLine, and Steven M Drucker. 2022. Strategies for Reuse and Sharing among Data Scientists in Software Teams. *International Conference on Software Engineering (ICSE)* (2022).

[23] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-step live programming by example. In *ACM Symposium on User Interface Software and Technology (UIST)*. 614–626.

[24] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv:2204.05999* (2022).

[25] GitHub. 2022. GitHub Copilot. https://github.com/features/copilot. Accessed 6/30/22.

[26] GitHub. 2022. Seeing multiple suggestions in a new tab. https://docs.github.com/en/copilot/getting-started-with-github-copilot/getting-started-with-github-copilot-in-visual-studio-code#seeing-multiple-suggestions-in-a-new-tab. Accessed 9/7/22.

[27] Luca Di Grazia and Michael Pradel. 2022. Code search: A survey of techniques for finding code. *ACM Computing Surveys (CSUR)* (2022).

[28] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices* 46, 1 (2011), 317–330.

[29] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[30] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Conference on Human Factors in Computing Systems (CHI)*. 1–12.

[31] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[32] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-sketch: Output-directed programming for SVG. In *ACM Symposium on User Interface Software and Technology (UIST)*. 281–292.

[33] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Conference on Human Factors in Computing Systems (CHI)*. 159–166.

[34] Nick Hynes, D Sculley, and Michael Terry. 2017. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS MLSys Workshop*, Vol. 1.

[35] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *International Conference on Software Engineering (ICSE)*. IEEE, 319–321.

[36] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *International Conference on Software Engineering (ICSE)*. IEEE, 1219–1231.

[37] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *ACM Symposium on User Interface Software and Technology (UIST)*.

[38] Ellen Jiang, Edwin Toh, Alejandra Molina, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2021. Genline and genform: Two tools for interacting with generative language models in a code editor. In *ACM Symposium on User Interface Software and Technology (UIST)*. 145–147.

[39] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *Conference on Human Factors in Computing Systems (CHI)*. 1–19.

[40] Mary Beth Kery, Amber Horvath, and Brad A. Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *CHI Conference on Human Factors in Computing Systems*. ACM, 1265–1276.

[41] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Conference on Human Factors in Computing Systems (CHI)*. 1–11.

[42] 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 *ACM Symposium on User Interface Software and Technology (UIST)*. 140–151.

[43] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2017. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering* 44, 11 (2017), 1024–1038.

[44] 8080 Labs. 2022. bamboolib. https://bamboolib.8080labs.com/. Accessed 7/20/22.

[45] 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 *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–11.

[46] Tessa Lau. 2008. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*. 65–67.

[47] Doris Jung Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. 2021. Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows. *Proc. VLDB Endow.* 15, 3 (2021), 727–738.

[48] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M Mitchell, and Brad A Myers. 2020. Multi-modal repairs of conversational breakdowns in task-oriented dialogs.

In *ACM Symposium on User Interface Software and Technology (UIST)*. 1094–1107.
[49] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. 2021. NB-Search: Semantic Search and Visual Exploration of Computational Notebooks. In *Conference on Human Factors in Computing Systems (CHI)*. 1–14.
[50] Xingjun Li, Yizhi Zhang, Justin Leung, Chengnian Sun, and Jian Zhao. 2021. EDAssistant: Supporting Exploratory Data Analysis in Computational Notebooks with In-Situ Code Search and Recommendation. *arXiv:2112.07858* (2021).
[51] Q. Vera Liao and S. Shyam Sundar. 2022. Designing for Responsible Trust in AI Systems: A Communication Perspective. In *Conference on Fairness, Accountability, and Transparency (FAccT)*. ACM, 1257–1268.
[52] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–40.
[53] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the ACM Symposium on User Interface Software & Technology*. 291–301.
[54] Barry McCardel. 2022. Hex Blog: Introducing: "No-Code" Cells. https://hex.tech/blog/introducing-no-code-cells. Accessed 8/16/22.
[55] Andrew McNutt and Ravi Chugh. 2021. Integrated Visualization Editing Via Parameterized Declarative Templates. In *Conference on Human Factors in Computing Systems (CHI)*. 1–14.
[56] Andrew McNutt, Anamaria Crisan, and Michael Correll. 2020. Divining Insights: Visual Analytics Through Cartomancy. In *Extended Abstracts of the Conference on Human Factors in Computing Systems (CHI)*. 1–16.
[57] Andrew McNutt, Gordon Kindlmann, and Michael Correll. 2020. Surfacing Visualization Mirages. *Conference on Human Factors in Computing Systems* (2020).
[58] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. In *International Conference on Art, Science, And Engineering*. ACM.
[59] Microsoft. 2022. IntelliSense. https://code.visualstudio.com/docs/editor/intellisense. Accessed 11/9/22.
[60] Microsoft. 2022. Use AutoComplete when entering formulas. https://web.archive.org/web/20221005003956/https://support.microsoft.com/en-us/office/use-autocomplete-when-entering-formulas-d51ef125-60ff-438f-ba26-d9bd6b363bbe. Accessed 11/11/22.
[61] Mathew Mooty, Andrew Faulring, Jeffrey Stylos, and Brad A. Myers. 2010. Calcite: Completing Code Completion for Constructors Using Crowds. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*. IEEE Computer Society, 15–22. https:///10.1109/VLHCC.2010.12
[62] Tamara Munzner. 2022. Developing Design Spaces for Visualization. (2022). https://hci.stanford.edu/courses/cs547/speaker.php?date=2022-03-04 Stanford Human-Computer Interaction Seminar.
[63] Minori Narita, Nolwenn Maudet, Yi Lu, and Takeo Igarashi. 2021. Data-centric disambiguation for data transformation with programming-by-example. In *International Conference on Intelligent User Interfaces (IUI)*. 454–463.
[64] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[65] Observable. 2022. Data table cell. https://observablehq.com/@observablehq/data-table-cell. Accessed 7/20/22.
[66] Cyrus Omar, Young Seok Yoon, Thomas D LaToza, and Brad A Myers. 2012. Active Code Completion. In *International Conference on Software Engineering (ICSE)*. IEEE, 859–869.
[67] Stephen Oney and Joel Brandt. 2012. Codelets: linking interactive documentation and example code in the editor. In *Conference on Human Factors in Computing Systems (CHI)*. 2697–2706.
[68] OpenAI. 2022. GPT-3. https://openai.com/api/. Accessed 6/30/22.
[69] Michael Quinn Patton. 1990. *Qualitative evaluation and research methods*. SAGE Publications, inc.
[70] Kylie Peppler and Yasmin Kafai. 2005. Creative coding: Programming for personal expression. *Retrieved August* 30, 2008 (2005), 314.
[71] Fernando Pérez and Brian E Granger. 2007. IPython: a system for interactive scientific computing. *Computing in science & engineering* 9, 3 (2007), 21–29.
[72] Sruti Srinivasa Ragavan, Zhitao Hou, Yun Wang, Andrew D. Gordon, Haidong Zhang, and Dongmei Zhang. 2022. GridBook: Natural Language Formulas for the Spreadsheet Grid. In *International Conference on Intelligent User Interfaces (IUI)*. ACM, 345–368.
[73] Deepthi Raghunandan, Zhe Cui, Kartik Krishnan, Segen Tirfe, Shenzhi Shi, Tejaswi Darshan Shrestha, Leilani Battle, and Niklas Elmqvist. 2021. Lodestar: Supporting Independent Learning and Rapid Experimentation Through Data-Driven Analysis Recommendations. *Visualization in Data Science Workshop* (2021).
[74] Steven P Reiss. 2014. Seeking the User Interface. In *ACM/IEEE International Conference on Automated Software Engineering*. 103–114.
[75] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
[76] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the Conference on Human Factors in Computing Systems (CHI)*. 1–7.
[77] Horst W Rittel and Melvin M Webber. 1974. Wicked problems. *Man-made Futures* 26, 1 (1974), 272–280.
[78] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. Codemend: Assisting interactive programming with bimodal embedding. In *ACM Symposium on User Interface Software and Technology (UIST)*. 247–258.
[79] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, and Peter W. Rose. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. *PLoS Computational Biology* 15, 7 (2019).
[80] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Conference on Human Factors in Computing Systems (CHI)*. 1–12.
[81] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence?. In *Psychology of Programming Interest Group (PPIG 2022)*.
[82] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. 2023. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2023), 1146–1156.
[83] Tabnine. 2022. Tabnine. https://www.tabnine.com/. Accessed 6/30/22.
[84] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *ACM SIGCHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–7.
[85] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the Loop: Supporting Data Comparison in Exploratory Data Analysis. In *Conference on Human Factors in Computing Systems (CHI)*. 1–10.
[86] Caleb Warren, Adam Barsky, and A Peter McGraw. 2021. What makes things funny? An integrative review of the antecedents of laughter and amusement. *Personality and Social Psychology Review* 25, 1 (2021), 41–65.
[87] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting stateful alternatives in computational notebooks. In *Conference on Human Factors in Computing Systems (CHI)*. 1–12.
[88] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *International Conference on Intelligent User Interfaces (IUI)*. 402–412.
[89] Brian Whitworth. 2005. Polite computing. *Behaviour & Information Technology* 24, 5 (2005), 353–363.
[90] Jack Williams and Andrew D Gordon. 2021. Where-Provenance for Bidirectional Editing in Spreadsheets. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10.
[91] Jo Wood, Alexander Kachkaev, and Jason Dykes. 2018. Design exposition with literate visualization. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2018), 759–768.
[92] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Conference on Human Factors in Computing Systems (CHI)*. 1–22.
[93] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging code and interactive visualization in computational notebooks. In *ACM Symposium on User Interface Software and Technology (UIST)*. 152–165.
[94] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.
[95] Yunwen Ye and Gerhard Fischer. 2005. Reuse-conducive development environments. *Automated Software Engineering* 12, 2 (2005), 199–235.
[96] Nur Yildirim, Alex Kass, Teresa Tung, Connor Upton, Donnacha Costello, Robert Giusti, Sinem Lacin, Sara Lovic, James M. O'Neill, Rudi O'Reilly Meehan, Eoin Ó Loideáin, Azzurra Pini, Medb Corcoran, Jeremiah Hayes, Diarmuid J. Cahalane, Gaurav Shivhare, Luigi Castoro, Giovanni Caruso, Changhoon Oh, James McCann, Jodi Forlizzi, and John Zimmerman. 2022. How Experienced Designers of Enterprise Applications Engage AI as a Design Material. In *Conference on Human Factors in Computing Systems (CHI)*. ACM, 483:1–483:13.
[97] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. 2016. Bing developer assistant: improving developer productivity by recommending sample code. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 956–961.
[98] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the SIGPLAN International Symposium on Machine Programming*. 21–29.