# Notebooks On Demand

Eric Rawn
erawn@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

S.E. Chasins
chasins@cs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

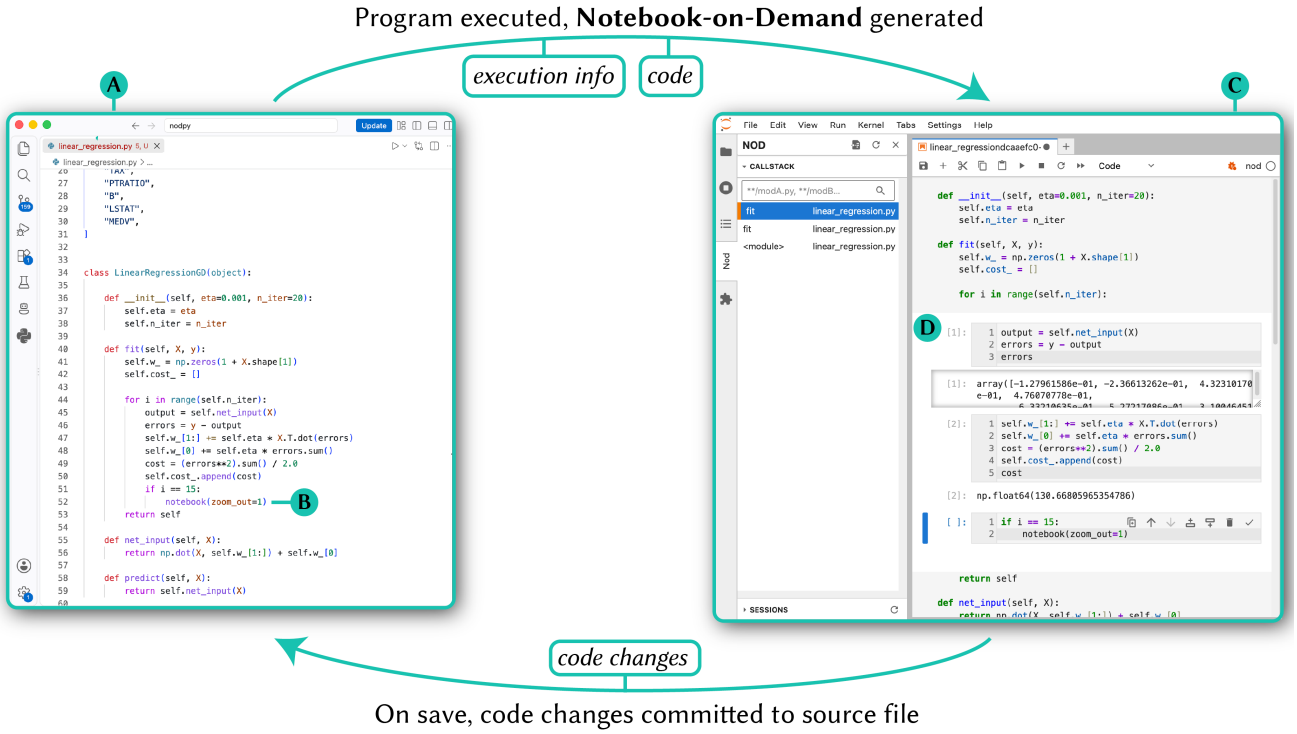



**Figure 1: Notebooks-on-Demand is a programming system that enables a programmer working on a Python codebase to flexibly transition between an IDE (A) and an augmented, embedded Jupyter notebook (C). When the `notebook()` function is executed (B), the system pauses program execution, then generates a notebook-on-demand (D). The new notebook includes all code in the function or block that contained the `notebook()` call, and a programmer can edit and rerun the code as in a standard notebook. Crucially, all variables in scope when the `notebook()` call was executed are also available inside the notebook; this execution information is necessary for making the notebook-on-demand useful for editing a small portion of a larger codebase. Users can commit code changes back to the original source file, where cell divisions and markdown cells are converted to empty lines and code comments, which will be reflected in future notebooks-on-demand invoked on the same code. Example adapted from [60]. We use Notebooks-on-Demand to explore design interventions into the *relationship between* programming environments, rather than design interventions into a single, isolated environment.**

## Abstract

IDEs support programmers in authoring large codebases but offer limited interaction with execution-time information. Computational notebooks support programmers in inspecting execution-time information during program authoring, but they can struggle to support users in authoring large programs that use functions and are divided across multiple files. This paper explores how to give programmers the benefits of both environments. We present Notebooks-on-Demand (Nod), a programming system that enables programmers to flexibly move between an IDE and a Jupyter notebook. In a four-week longitudinal user study with five participants,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*Preprint,*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

we found that Nod addressed challenges with notebooks and IDEs, helped users focus on small parts of a codebase to build understanding, and served as an experimental staging ground for developing code edits. We discuss design implications for ephemeral, on-demand, and multi-environment programming systems design.

## CCS Concepts

• **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → *Software notations and tools.*

**ACM Reference Format:**
Eric Rawn and S.E. Chasins. 2018. Notebooks On Demand. In *Proceedings of (Preprint).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Sometimes a single programming environment cannot give us everything we need. Consider this block of code marked as a "workaround" for a specific observation on the James Webb Space Telescope [3]:

```
1  slitnum = {'S200A1': 1, 'S200A2': 2, 'S400A1': 3,
   ↪  'S1600A1': 4, 'S200B1': 5}
2  for name in names:
3      with fits.open(name, 'update') as hdulist:
4          fxd_slit = hdulist[0].header['fxd_slit']
5          grating = hdulist[0].header['grating']
6          for extnum in range(len(hdulist)):
7              hdu = hdulist[extnum]
8              if hdu.name == 'SCI':
9                  sltname = hdu.header['sltname']
10                 sourceid = hdu.header['sourceid']
11                 if sltname == fxd_slit:
12                     newid = 1
13                 else:
14                     newid = 10 * slitnum[fxd_slit] +
                       ↪  slitnum[sltname]
15                 hdu.header['sourceid'] = newid
```

This is a complex piece of code! As part of a larger piece of software (~600 lines), this block of code references many other existing variables that load complex data objects. As part of the larger script, all ~600 lines are executed at once, including expensive data loading and processing operations, which slows the programmer's process of iteratively making changes in response to seeing intermediate outputs (e.g. the value of hdu after this block of code is run). While a breakpoint debugger within an Integrated Development Environment (IDE) would enable *inspection* of a variable like hdu, the programmer has to re-run the entire loading and pre-processing workload as soon as they have made their next tweak. A computational notebook [19] would solve this problem while also enabling the programmer to debug by inspecting the James Webb Telescope code's own custom visualizations of choice, not via a single standardized debugger interface.

But moving lines 9 to 15 into a notebook from a Python script is not as easy as it looks. Without the many complex data objects (e.g., names, hdulist, hdu) that are needed to execute this block of code, a computational notebook would not provide any of the benefits listed above.

Building complex, large-scale software in notebooks instead of IDEs can lead to other problems. This James Webb Telescope script was built in a notebook (~600 lines of Python over 12 cells), and it defines only one function. Because each cell must be parseable and a partial function definition isn't parseable, function definitions can't be split across multiple cells. Since notebook programmers often like composing programs by tweaking one cell at a time and composing cells to produce the broader script, notebook environments discourage functions more than IDEs. As a result, almost all of the variables defined in this script are global variables, which can be hard for programmers to reason about. Organization into functions is also one of the primary characteristics that IDEs leverage to support navigating between different portions of a large codebase. Navigating a notebook that uses few functions can become burdensome—the telescope script, with its ~600 lines of Python, would span 8 A4 pages if printed (including outputs), so scrolling is time-consuming even for this relatively small-scale program.

One way to overcome these obstacles to organizing and navigating complex or large codebases is by using an IDE instead of a notebook. In an IDE, using functions and organizing code across multiple files don't come with the same drawbacks that they carry in notebooks. Unfortunately, IDEs offer only very limited access to *dynamic* (execution-time) information. Suddenly the custom visualizations that were so helpful for authoring, understanding, and debugging lines 9 to 15 within the notebook setting—those are only available on a complete program reexecution, not sitting side-by-side with the relevant code cell and getting rapidly updated every time we tweak the cell.

Both IDEs and notebook environments support valuable interactions, but programmers have to structure their programs quite differently to get the benefits of each. Many IDE features only become available if a programmer organizes their project into many files and functions. Many notebook features only become available if programmer compresses their project into a single file with no functions.

How might we give users the benefits of *both* IDE and notebook environments? Rather than try to integrate the features of one environment into the other, in this paper we explore how to support programmers moving flexibly between two environments to author a single program, benefiting from the interactions provided by both notebooks and IDEs without having to restructure their programs to do so.

We built *Nod*, a programming system that generates a *Notebook-On-Demand*: an augmented Jupyter notebook [19] interface constructed during the execution of a Python script. In a Nod notebook, users interact with execution-time information, edit and rerun their code, and send code edits back to the source file.

This work contributes:

- **Nod**, a Python package that enables programmers to generate an augmented Jupyter notebook (a notebook-on-demand) at any point in a Python script.
- An in-situ [66] user study with five programmers, who used Nod in their regular work for ~4 weeks.

- A discussion of design implications for building ephemeral, on-demand programming systems.

## 2 Motivation and Background

Throughout this paper we will frequently distinguish between information that is *static* versus *dynamic*, using the standard definitions from programming languages. *Static* information is available at program editing time or program compile time—e.g., the particular file that includes an expression, the line number at which the expression can be found, the fact that a given function references a variable named `x`. *Dynamic* information is not available statically but becomes available at program run time or program execution time—e.g., the sequence in which functions are called during a given program execution, the fact that the name `x` is now associated with the value 5 based on the arguments to those function calls.

### 2.1 Programmers Value Notebooks for their *Dynamic* Interactions

A defining feature of notebook environments is that they support iterative program modification and execution against a persistent interpreter, enabling users to compose, inspect, and debug their textual programs and the intermediate variables those programs generate. Previous work illustrates how notebook programmers find these interactions valuable for exploratory [5] work where the purpose of writing the program is to "discover what it should do"[70]. Notebook programmers find iteratively executing small parts of their program (as little as a single line of code) while displaying the results of that computation to be a valuable strategy for composing [36], debugging [10, 28, 65], and verifying [28, 70] their programs. Notebook programmers also report using cells as impromptu alternatives [22, 34] which could be selectively executed. Because these interactions all center on users working with execution-time information about their programs, we will refer to them collectively as *dynamic interactions* for convenience.

Notebook programmers from multiple studies reported *reorganizing* code into different cells depending on their current needs: Kery identified an "expand then reduce" strategy [36] in which notebook programmers composed new code in small cells and later incorporating it into larger cells which corresponded to different "logical units" of their work [36]. But when notebook programmers needed to debug their programs, they often rewrite their programs back into an "expand[ed]" form to inspect the results of each line individually[10, 65], then "slowly stitch [their program] back together until [they] get something that works right" [10]. In other words, because the organization of a notebook program into cells determines what can be executed together, what code is seen together, and where outputs are displayed, notebook programmers rewrite their notebook programs to enable different interactions with their programs. In this paper, Nod offers an alternative story: by generating temporary notebook environments on small parts of a program which can be iteratively composed and debugged, we might see Nod as providing a tool to automatically "expand" different parts of a large Python codebase into a temporary notebook as needed.

### 2.2 IDE Programmers Value IDE *Static* Interactions, Especially with Large Codebases

Integrated Development Environments (IDEs) give programmers valuable tools for working with large codebases, which often involves serious time spent reading, interpreting, and navigating the program text which is distributed across many files and functions. In a study of Java programmers performing maintenance tasks, Ko et al. [38] found that participants spent just 20% of their time modifying the program text itself, while the remainder of the time was spent reading, searching and navigating, tracing relationships between the source code via the Eclipse IDE reference navigator (e.g. to trace implementations of function calls), file navigator, dependencies view, and text search. Kramer et al. found that call graph-based program text navigation tools [33, 41] incorporated into an IDE increased correctness rates and shortened completion times of maintenance tasks [40]. Robillard [64] hypothesized that searching for relevant parts of a codebase through dependency- and reference-tracing tools (rather than visually scanning or keyword-searching) increased success rates on their study task. Other studies have further emphasized the importance of code reading and navigation in software engineering [72] and debugging [43]. We refer to these interactions as *static* because they operate on program source text with compile-time information.

While much work remains on improving static IDE interactions for searching and navigating large codebases, compared to notebook environments they represent a core focus and strength of IDEs. With notebooks-on-demand, our design challenge was not to improve on this, but rather to design a programming system which allowed users to take advantage of these static interactions *while also* being able to interact *dynamically* with their same programs through notebook environments.

### 2.3 Large Programs Aggravate Existing Challenges with Notebook Environments

Especially once notebooks become sufficiently large [22, 28, 36], the benefits discussed above can often become serious problems for notebook programmers.

The persistence and sharing of variables between all notebook cells (which enables the "expand-then-reduce" strategy above) can lead to confusion over how a variable was instantiated if it is defined in multiple cells [75] or accidentally referencing the wrong variable with a shared name [22, 78].

Because a notebook program can only be split into cells between two independently-parseable subprograms [28, 62], notebook programmers give up the ability to flexibly restructure their code into different cells (as in the "expand-reduce" strategy) when that code is included in a function definition, for loop, or conditional. Using functions in notebooks was described as "counter-productive" [59], often too onerous [34, 31], and only done at the conclusion of the work [16, 28]. Avoiding function use in notebooks leads to large numbers of global variables in a single notebook state (since variables are not scoped locally), a program that can difficult to parse [28], and an inability to use code as a "modular building block" [28], leading to duplicating code across a single notebook.

The ability to execute individual cells in a user-directed order enables impromptu versioning but can also result in a different notebook state than if that notebook had been restarted and run top-to-bottom [13, 67, 71, 75, 78].Cells can also make non-idempotent changes to persisted notebook variables which can lead to unexpected results [71].

The challenges users face with notebooks are exacerbated by notebook length such that many users choose to start new notebooks once existing ones get too large, copying relevant sections [22, 36] or using notebooks only as a temporary "scratch pad" [28, 36] alleviate struggles with navigating, reasoning about, and reorganizing large notebooks. Others moved function definitions to Python scripts [28], at the cost of increasing the manual program restructuring required to "expand" the body of that function in a notebook later on.

To summarize, the dynamic interactions notebooks provide to selectively edit, run, and inspect the results of different sections of a program via cells depends on that program being structured without functions (and other constructs like loops and conditionals). In order to make use of these interactions, notebook programmers are often hesitant to use functions until the conclusion of their work (if at all). Notebook programmers accordingly face challenges that would be expected given a dearth of function use: unable to otherwise locally scope variables, they face confusion managing many global variables. Unable to organize code into different files, notebooks grow into large, single-scoped programs which can be difficult to navigate, edit, and debug. Only able to use cells and a top-to-bottom notebook execution functionality to direct the control flow of their program, they face confusions when variable names are unexpectedly overwritten or variables are modified non-idempotently.

### 2.4 IDEs Provide Limited Dynamic Interactions

Compared to notebook environments, IDEs often provide limited *dynamic* interactions with execution-time information about a program. Integrated breakpoint debuggers are a common IDE feature, but can be often avoided by programmers [4] in favor of printing the value of intermediate variables and evaluated expressions to the console (i.e. `print()` debugging), with some participants citing difficulty with and lack of knowledge of using debuggers as reasons to prefer other methods. Although integrated debuggers can encourage programmers to transition more quickly between editing program text and inspecting execution results [1], breakpoint debuggers are not responsive to changes in the source program text once the debugging session has begun (this is also true for REPL enviornments like IPython [55]). By contrast, notebook environments support programmers in composing and modifying their source programs while re-executing them in small sections against intermediate variables to see the results.

### 2.5 Notebooks-on-Demand Aim To Bridge Notebooks and IDEs Without Tradeoffs

In both their strengths and challenges, notebook environments and IDEs are *complementary*: IDEs support static interactions for navigating and managing large programs spread out among multiple files and functions, which programmers struggle to do with notebooks. Notebooks support dynamic interactions which IDEs often do not provide, and integrated debuggers are often eschewed. But notebook environments and IDEs are *incompatible* in a key way: their interactions require different program structures. In notebooks, iteratively "expand[ing]" [36] a program into different cells requires that program not be within a function definition and be located in a single notebook file. IDEs support managing large codebases in exactly the opposite way: providing interactions which are unhampered by and even rely on code being structured across files and functions. The goal of Nod is to build a programming system which does not force programmers to choose between the dynamic interactions they value about notebooks and structuring their programs into files, functions, and classes—especially when dealing with large programs. By dynamically generating a notebook-on-demand on just a small region of a user's codebase, Nod aims to side-step many of the challenges users face with notebooks by containing their scale and disposing of them when users are finished making changes, while at the same time providing additional ways for IDE users to interact dynamically with their program.

## 3 Related Work

### 3.1 Addressing Challenges with IDEs

A wide range of systems have explored how to enable users to interact with execution-time information inside their IDEs. Systems like projection boxes [42], Python Tutor [20], Omnicode [32], Birdseye [2], and the system presented in Cheng et al. [11] display associated execution-time values of intermediate variables as annotations onto program text. REPL (read-eval-print loop) environments such as LISP [15], OPL-I [30], IPython [55] provide ways for users to interactively execute programs and see results. Other systems display execution-time information in separate interfaces or embedded windows, such as Smalltalk's debugger [74], which enables editing a source program within the debugger interface (but not re-executing that code) and Pynt [17], which provides an embedded debugger in Emacs with an IPython notebook.

These systems support programmers in inspecting the execution-time information of their programs in various ways. In this work, we explore how to give users a fully-featured *program authoring environment*, rather than only a program inspection environment, alongside an IDE. With Nod, programmers have the option, if they want, to author their entire source file within a JupyterLab interface after a single initial `notebook()` call. The tools above are focused on extending an IDE (a program authoring environment) with program inspection or program exploration tools (not program authoring environments), so none of the interfaces above support this workflow. In contrast to the authoring+exploring combinations above, we explore an authoring+authoring combination.

### 3.2 Addressing Challenges With Notebooks

We have seen a vibrant exploration of the notebook design space among researchers and beyond. Some interventions aim to address global variable and unintentional cell execution order confusions in notebook environments by controlling how variables can be referenced in some way: dividing the notebook into contiguous groups which share a scope [62], requiring variable references outside of

defining cells to include an unambiguous cell label [39], creating a scoped "side-board" notebook region [27], and duplicating the underlying kernel to operate a notebook "fork" in parallel [78]. Other systems aim to avoid just cell execution order confusions by automatically enforcing certain execution orders of cells [47, 48, 52, 56, 63]. Other strategies to address global variable confusions include removing or identifying problematic code and visually organizing notebooks: filtering out unnecessary code [22, 44, 69], analyzing notebooks for potential issues [6, 57, 76], visually organizing and annotating sections [9, 21, 68, 73, 77, 79], and helping users manage exploration histories and alternatives [35].

In this paper, we explore the idea of programming systems that dynamically generate live programming environments for a user's program given its execution behavior. We picked one particular live programming environment—Jupyter Notebooks—to understand how making a Jupyter notebook environment that complements an IDE rather than replacing the IDE might address some of the challenges users report. For systems that aim to improve user experience within Jupyter notebooks, our contribution is orthogonal but in fact complementary. Nod can benefit from this line of research because improvements to notebook environments can also improve notebooks-on-demand, extending the reach of these improvements to IDE users.

### 3.3 Bridging Multiple Programming Environments

Livelits [53], Mage [37], Sketch-n-Sketch [12, 23–25, 45, 50], B2 [80], a system by Rauch et al. [61], and Glinda [14] all enable users to modify programs via standard textual edits *or* by editing a visual representation of some information about the program—e.g., the visual output produced by execution, or a color picker representing an RGBA literal. Overall in the literature, this textual+visual combination is the most common approach to bringing together multiple program *authoring* environments. In this work, we explore how to support users in authoring programs flexibly between two text-based editing environments, IDEs and notebooks.

Engraft [26] provides an API for composing live programming GUI components. Where Engraft explores how to support users in authoring a bespoke live interface by composing programming environments together, Nod explores how to support users in authoring one shared textual program flexibly between two programming environments.

## 4 Designing Notebooks-on-Demand

Notebook environments enable users to interact with the intermediate values produced during program execution, editing and re-running individual statements which can utilize and define persisted intermediate variables. But notebook users can face challenges navigating, organizing, and reasoning about notebook programs, especially large notebook programs2.3. While best practices often recommend that notebook programmers make greater use of functions to address these challenges [67], changing the program's structure in this way prevents many of the notebook interactions users value: variables defined inside a function are only available within the single cell that defines the function, and code within function definitions cannot be executed interactively by splitting lines into individual cells. By offering users interactions that are only available for certain kinds of program structure, notebook interfaces force users to choose between writing their programs in potentially more organizable, navigable, and debuggable ways (e.g., using functions) and fully utilizing their notebook environments.

By contrast, many IDE interactions are only available if programs are organized into functions and files. Many IDEs show hierarchical codebase views, showing the layout of files in directories. Table-of-contents navigators enable users to jump to function and class definitions within a given file. Context menu interactions enable navigating across functions and files via reference tracing and provide function definition previews when function calls are selected. Aside from interface support for multiple files and multiple functions, the fact that the IDE does not interfere with function or file divisions puts the many other benefits of codebase organization and modularity in reach [18, 49, 54]. We might additionally notice that IDEs seem to support users in exactly the ways programmers struggle with notebooks: navigating, organizing, and reasoning about large programs. On the other hand, IDEs often rely on external dynamic environments like REPLs and breakpoint debuggers to enable users to inspect and debug intermediate variables, which do not support interactive program authoring as notebooks do.

The key problem, on this framing, is that notebook environments and IDEs require users to structure their programs in different ways to make full use of each environment's functionality. Therefore, the first design problem Nod explores is: how might we design a programming system which enables both interactive and non-interactive programming environments to use the *same* program (with the same program structure) for *different* interactions?

The design of Nod was informed by four key design principles:

*4.0.1 Notebooks-on-Demand Should be **Limited**.* How might we design a programming system to give users the strengths of both notebooks and IDEs without their drawbacks? In this paper we explore one possible direction: seeing both environments as providing complementary ways of interacting with a single program. In other words, could we address some of the challenges with notebooks by asking them to do *less*?

This was initially counterintuitive for us as designers. We had been thinking about ways to address the challenges that programmers have with notebooks by trying to build more guardrails into notebooks themselves: restricting how users could reference variables, automatically running cells in certain orders, or creating cells that can reference and call each other as functions. But each seemed to be trying to force notebooks to work more like an IDE, and would accordingly limit the ways users can interact dynamically with their programs as they addressed the notebook challenges users faced (see Section 2.3).

We wanted to design a system that would enable a notebook environment to focus on its strengths: enabling users to interact with the dynamic aspects of their programs, iteratively composing and executing small programs on their intermediate variables (for brevity we label the cluster of interactions involved in this usage of notebooks as *dynamic interactions*). Additionally, we hypothesized that as long as the notebooks stayed small, they would avoid many of the problems notebook programmers report (see Section 2.3). For both of these reasons, notebooks-on-demand should be ***limited***:

they should focus on enabling the *dynamic* interactions that notebook users value on small programs, and leave the challenges of organizing and navigating large programs to an IDE.

*4.0.2 Notebooks-on-Demand Should be* ***Dynamically-Generated and Disposable*** *Environments.* Notebooks-on-Demand should be generated at execution-time and should be persisted only for the life of that execution.

The primary design goal of Nod is to support programmers in using both a notebook environment and an IDE to author the same program, with the *same program structure*, for *different interactions*. But standard notebook environments and IDEs represent programs differently (in the most literal sense, a difference between an `.ipynb` and `.py` file). Notebook files contain much more than program text: a JSON representation that includes a list of cells, each of which has a snippet of associated program text, sometimes markdown that will display near the cell, or even images that will be rendered alongside the cell. More importantly, a notebook environment contains not just static text files (as an IDE does), but the state of a currently-running program execution. This is a persisted REPL session that includes the values generated by running notebook cells over the course of the notebook's history. What should be the relationship between these environments and their respective artifacts?

Returning to our initial question of how to enable both environments to provide interactions on *the same program*, we wanted a notebook-on-demand to be a representation of the user's program that gave them new interactions with their program, not *a different program*. Because a notebook environment contains a REPL session in addition to program text, a notebook-on-demand should correspond not just to a region of the program *text* but also a specific place in the *execution* of that program.

With these considerations in mind, notebooks-on-demand should be **dynamically-generated and disposable**: because notebooks-on-demand manipulate a specific execution of the user's program, they must be generated at execution time and should be persisted only for the life of that execution. Users should only have one persistent representation of their program: the Python source file. If the user wants to revisit the same notebook-on-demand, they should be able to regenerate their notebook-on-demand by re-executing the program, without needing to save any additional information outside of their source program.

*4.0.3 Notebooks-on-Demand* ***Should Be Controlled by The Execution of the Codebase.*** At the beginning of the design process for Nod, a major question was how users should control when a notebook-on-demand is generated and what it displays. Consider an interaction for a user to specify a notebook-on-demand on a certain region of their text (i.e. `#a,#b,#c, or #d` and at a specific time in execution (e.g. a place where `i` is a certain value) in the following program:

```
def f(): #a
    def g(x): #b
        print(x)
    for i in range(10):
        g(i) #c
    #d
f()
```

We imagined an IDE interaction where users would indicate a region of their program (e.g. the code under `#b`) with a mouse selection or special code comments (analogous to `#a,#b,#c,#d` above) to open in a notebook-on-demand. This would provide useful static information—specifically, the span of the source code that should be included—but would not express key dynamic information. For example, how would the user express that they want the notebook to include information about the third iteration through the for loop? We might also imagine an interaction analogous to a breakpoint debugger, where users mark a line in a program which, upon execution, pauses the program. But this fails to specify how much of the program text to include in the notebook-on-demand. Suppose the line containing `g(i)` above is marked to invoke a notebook-on-demand upon execution—how should a user open just the body of the `for` loop as opposed to the entire function body of `f()`? Later in our ideation we experimented with a collection of library functions, one for generating a notebook-on-demand for the containing function call, one for the containing `for` loop, another for the containing `while` loop, and another for the `if` statement. But for every scenario we supported with a new library function or parameter, we could imagine yet another that hadn't yet been supported. These approaches seemed to add needless complexity and clutter to a user's program and IDE, while still lacking the flexibility to handle many plausible notebook-on-demand targets.

So we returned to the initial insight for Nod: that how a program is structured can change how programming environments enable users to interact with it. This is a problem for notebook environments and IDEs because they each offer valuable interactions, but for *different* program structures. For the question of how to control the behavior of a notebook-on-demand, however, there is no apparent tension with other environments. This early insight now points to our solution, to **use the structure of the program**! A notebook-on-demand operates on a given program execution, and programmers already have a familiar and extremely flexible tool for manipulating program executions, the programming language itself. A user should specify when a notebook-on-demand is run and what it displays by doing the same task that an IDE already supports: writing their program.

*4.0.4 Notebooks-on-Demand should Enable* ***Further Dynamic Interactions.*** Notebooks-on-demand are *limited* in that they focus on giving users ways to interact with the dynamic aspects of just one small part of their codebase at a time. Because a notebook-on-demand is designed to offer complementary interactions *alongside* an IDE on the same program, a notebook-on-demand does not need to enable interactions that are already well-supported by an IDE. Could limiting notebooks to their dynamic strengths also present opportunities for *even more dynamic interactions* in a notebook-on-demand that might otherwise be confusing in a normal Jupyter notebook? A notebook-on-demand should explore new ways for users to interact dynamically with their program.

## 4.1 System Overview: Nod

To describe the Nod system, we start by briefly describing how Nod addresses our design principles.

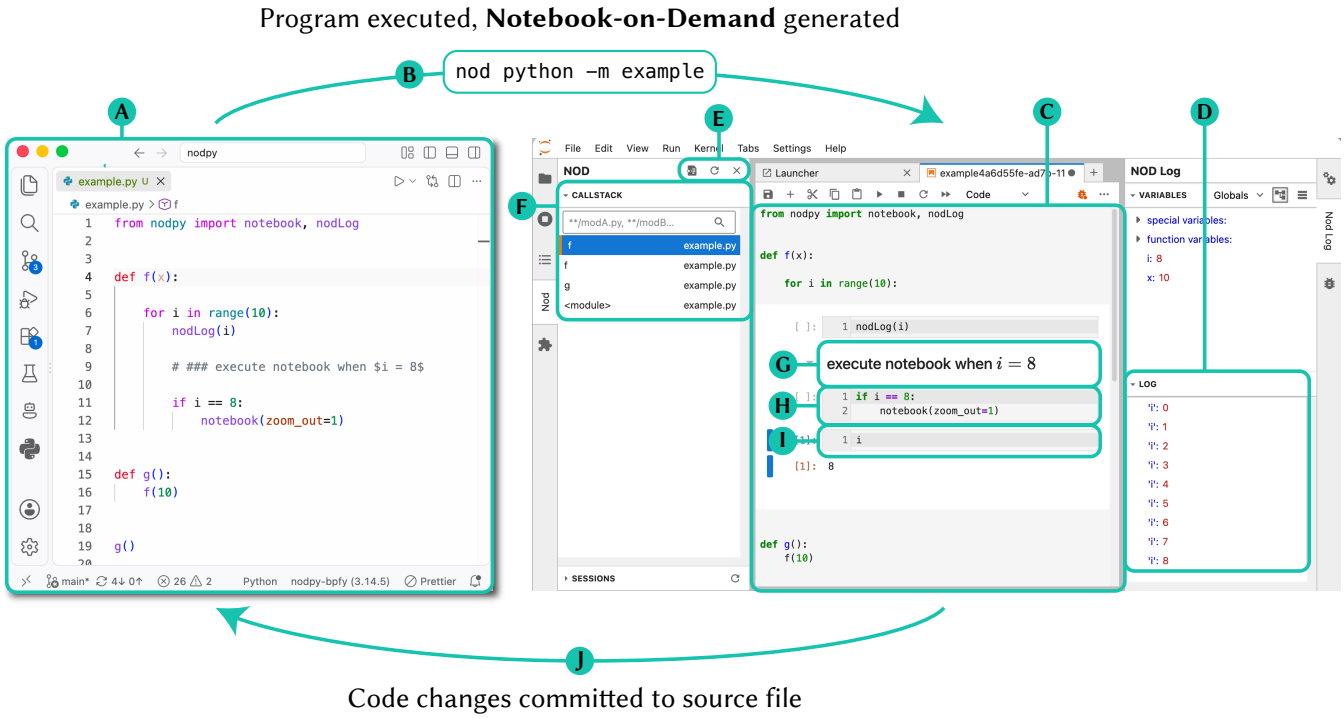



**Figure 2: The Nod Interface. (A) A Python script in an IDE includes the `notebook()` function. When this script is executed from the command line with `nod` preceding the command (B), Nod generates an augmented JupyterLab interface (right image). The editable region of the notebook (C) includes a small region of the overall codebase, selected via placement of the `notebook()` call; the original Python program text has been divided into cells based on the placement of empty lines, and comments have been converted to markdown cells. Nod shows the program execution state at exactly the point of the `notebook()` call (in A), which can be seen by printing the variable `i`; because the programmer triggered Nod in the `i==8` iteration of the loop (H), printing `i` shows 8 (see I). After iterating on their code in the notebook, programmers can use buttons (E) to save edits to their underlying codebase (J) or trigger a fresh run of their codebase (potentially triggering generation of an entirely new notebook-on-demand). Nod enables users to navigate the program execution via the callstack pane (F); in the callstack pane, clicking on any function name will open a new notebook-on-demand representing the state of the execution immediately before control was transferred to the next function in the callstack. Because this program calls `notebook()` with a `zoom_out` parameter, two notebooks-on-demand for function `f` appear in the callstack (F): one displays only the body of the `for` loop (this notebook-on-demand is currently visible in C); a second notebook-on-demand displays the entire body of the function `f`. Using the `nodLog(i)` call in the original program (A), the user has saved the values that appeared in `i` at every instance of a `nodLog(i)` call. These are visible in the rightmost pane (D); clicking on any logged value of `i` in the NOD Log list will assign `i` that value in the current notebook state.**

*4.1.1 Nod is **Limited**.* A notebooks-on-demand shows a small region of a larger Python codebase [1] and loads into the underlying IPython REPL session just the variables available at the start of that small region. That is, a notebook-on-demand offers just a single portion of a potentially much larger codebase, both in terms of the editable text and the variables that can be referenced. The user can inspect, modify, and re-execute that smaller portion within the notebook environment.

*4.1.2 Nod is **Dynamically-Generated and Disposable**.* Working with Nod, a user's persistent program is a Python file. When that program is executed with `nod` at the command line and a `notebook()` call is reached, a notebooks-on-demand is generated for each function in the call stack. Any changes made to a notebook-on-demand can be persisted with the button at **H** in Figure 2, at which point the notebook will be converted to Python text (e.g. cell breaks are converted to newlines, markdown cells are converted to comments). The format supports "round-tripping"—if all notebook changes are persisted into the codebase, triggering Nod again will recreate the same sequence of cells and markdown cells.

*4.1.3 Nod is **Controlled by The Execution of the Codebase**.* In Nod, a notebook-on-demand is invoked by using a construct, `notebook()`, which when executed, pauses the program execution and opens a notebook-on-demand with all variables currently in scope. By default, `notebook()` displays the body of the function

[1] Except at the module level, where they show an entire file, akin to a standard notebook

definition within which it is called (all of the program text that is permitted to reference names in the current stack frame). To change the code displayed in a notebook-on-demand, they can change the code in the function definition, call `notebook()` in a different part of their program, or use the `zoom_out` parameter (see **G** in Figure 2) to specify the number of indent levels to include around the `zoom_out` call. `zoom_out` enables users to use their program to specify precise conditions for the invocation of a notebook-on-demand. Users can alter the program execution that the notebook-on-demand manipulates using all of their normal processes they use for altering program execution. For example in Figure 2, the program causes the notebook to display the ninth iteration of the `for` loop by calling `notebook()` inside a conditional that only triggers on the ninth iteration, and then uses `zoom_out` to include only the body of the `for` loop in the editable notebook. Rather than try to build a custom IDE interface, annotation language, or a library of primitives with their own customization options, we instead provide to users a way of specifying the behavior of Nod in the medium they are already familiar with: their programming language.

*4.1.4 Nod **Enables Further Dynamic Interactions**.* A notebook-on-demand corresponds to a larger source program that is navigable in an IDE as a collection of text files. Instead of focusing on this static element of the organization—the position of code inside source code files in the codebase—Nod focuses the user on dynamic organization, the sequence of function calls *during execution* that led up to the `notebook()` call. Nod users navigate their program via the callstack (See **C** in Figure 2). For each function in the callstack, Nod creates a notebook-on-demand, and when the user selects a given notebook-on-demand from the call stack, Nod makes available all (and only) the variables in scope at that point in the program. Nod programmers can make edits and re-execute code at any level in the callstack. Because user edits can change the program's behavior in unpredictable ways (e.g., modifying global variables or changing the control flow of the program such that `notebook()` is no longer called), the default settings enforce that users must restart their program (and therefore generate a new Nod interface) between making edits across levels of the callstack, to ensure their notebooks-on-demand corresponds to the current source program's behavior.

Nod also supports a style of logging that saves logged values in a way that keeps them available for computation. In contrast to standard logging, which prints a stringified version of a logged value, every execution of `nodLog(var)` stores a copy of `var`'s value that the user can later bring into their notebook environment and manipulate. Users can provide any number of variables as arguments, and the value associated with the variable at the time of each `nodLog` call will be depicted in the Nod Jupyter Extension, available for users to load into their current IPython session. (See **D** in Figure 2.)[2] Clicking on the value $v$ collected by a given execution of `nodLog(var)` maps the name `var` to $v$ in the current notebook state.

*4.1.5 Implementation.* When executed, `notebook()` invokes an embedded IPython [55] kernel within a suspended Python source program, which subsequently generates an augmented JupyterLab [19] interface. A kernel connection file is stored in the user's current directory to facilitate the temporary session between the embedded kernel and the Jupyter Client (users with persistent cloud notebook environments can also connect to a Nod embedded kernel from the Jupyter interface this way). To prepare the interface, the system locates the source text associated with each stack frame and separates the function body from its definition via its Concrete Syntax Tree (CST). Variable references for each stack frame are kept by the kernel to load into the user namespace when the user navigates to different stack frame levels in the interface (see **F** in Figure 2. In order to propagate changes in a Nod session back to original source files, Nod changes as little as possible about the program source text it displays, using the CST to preserve formatting. To execute the bodies of functions with `return` statements inside of a Nod session, we transform `return` AST nodes when cells are run to instead print returned variables, preserving the source text displayed to the user while making it compatible for a notebook environment.

Nod extends many points in the Jupyter and IPython ecosystem. It comprises an embedded IPython kernel, a user-facing Python library, a JupyterLab extension, a Jupyter server endpoint extension, an AST transformer IPython extension, a command-line interface, and a custom implementation of the Jupyter Kernel manager. In total Nod spans ~3000 lines of Python and ~4500 lines of Typescript and is released open-source.

## 5 User Study Design

To understand how Nod is incorporated into and potentially changes the programming practices of users, we gave Nod to 5 Python programmers to use in their real work for approximately 4 weeks in an *in-situ*[66] user study. After an introductory session (~1hr) to introduce participants to Nod and obtain informed consent, participants were instructed to use Nod only as it was useful to them. Researchers met with participants for two semi-structured interview sessions, at approximately midway through the study (~60 min) and at the conclusion ( 90 min), to discuss their experience with Nod. Because we wanted to understand Nod's use across a range of domains and programming practices, we filtered participants only for substantive time Python programming in their regular work (>2h per week) and programming experience beyond an introductory level. Participants were compensated based on their total interview duration at a rate of $30 per hour.

Interview recordings and transcripts were analyzed via a thematic analysis [7]. The first author generated initial codes, from which the first and second authors iteratively generated themes.

We chose an *in-situ* study design for a two primary reasons. First, rather than decide on a success measure ahead of time (e.g. frequency of use of the system), we instead wanted to understand the diversity of ways programmers might find Nod useful among their existing programming environments. Second, we wanted to understand how participants incorporated Nod into (and subsequently changed) their own programming practices within the specific contexts of their different programming environments, computational infrastructures, programming tasks, and surrounding constraints.

[2]Variables must be able to be duplicated in memory to be used by `nodLog()`. Due to the particularities of Python, not all Python variables are copyable in this way. See [58]

Some participants wrote programs primarily on their own during the study period, showing only results to colleagues (P1, P2), while others were writing software collaboratively as part of large teams (P3, P4, P5). Some participants were regularly using LLM-based programming tools (P2, P3, P4) and command-line REPL tools (P1) in their programming work. Each of these distinctions played a role in how each participant used Nod, and what aspects of the system were salient to them.

# 6 Findings

## 6.1 Nod Addressed Challenges with IDEs

Many participants (P2, P3, P4, P5) discussed how Nod enabled them to interactively view and edit their programs while working in a large codebase, something they had previously struggled to do. P2 said "there are pros and cons programming in a Python file—if you have modular code, it doesn't allow you [to] check [intermediate variables, which]—is much nicer with Nod (P2)." Some participants used Nod to build understanding of their programs and inspect dynamic information, and compared the experience of using Nod to using a debugger: P3 said "To be honest, I've already used Nod more than I've used a debugger"(P3), and P4 said "Nod is [now] my debugger (P4)." P3 highlighted how, in contrast to Nod, the breakpoint debugger in their IDE (VSCode [51]) did not support interactive editing and rerunning, while P4 mentioned getting the benefits of a Jupyter Notebook environment while still being able to edit a Python text file in their IDE, using Nod in situations where "you don't want to work *in* a Jupyter Notebook, but you want to still work *with* Jupyter Notebooks (P4)."

## 6.2 Nod Helped Address Challenges With Notebooks

All participants (P1-5) discussed how Nod mitigates some of the challenges they face with Notebooks.

P1 discussed how Nod enabled them to use function definitions much more in their programs because it enabled notebook interactions within function bodies: "[Using Nod has] been really positive... Normally I'm really scared of functions. I'm really stuck in my linear, top-down for-loops, absolutely no function calls. Mostly because I'm just scared of debugging a function later...but now for the first time [with Nod] I've felt less scared of using functions—I'm actually using functions and can debug them after they have been made, which I never would have done before (P1)." P1 described being able to invoke a notebook-on-demand within a function body as "making a function live (P1)." P2 and P3 similarly commented on their difficulties using notebooks with programs which included functions, classes, and packages, which were alleviated with Nod.

Nod also helped mitigate challenges with very long notebooks for some participants (P1, P2, P4), with P4 saying that because Nod only displays a small section of the codebase in a notebook, "I don't have to scroll through everything, I can just have this one function [displayed in the notebook] (P4)."

For P1, executing sections of their script in a REPL could lead to confusion over which global variable was being edited: "before it was really easy to not realize what you were editing with misnaming variables (P1)", and that with Nod, "I don't have to be as cognizant of what variables I'm accessing and making sure the right thing is being saved to the right variable, and [that] I'm not using a variable from the top of the file (P1)." Because Nod supported notebook interactions inside of function bodies, P1 began writing more of their code within functions, which reduced the challenges they faced with global variables in the REPL.

## 6.3 Participants used Notebooks-on-Demand to "Zoom In" Within a Larger Codebase to Understand, Edit, and Present

*Using Nod To Understand the Codebase.* Many participants (P1, P2, P3, P4) discussed using a Notebook-On-Demand to understand sections of their code through manipulating and executing the notebook code and inspecting the value of intermediate variables. P2, working on a machine learning model, phrased this idea most poignantly: "what I'll do after [working in Nod] is go to the blackboard and write some theory, so what I need [from the notebook-on-demand] is an understanding(P2)", and that using Nod helped them bridge their theoretical understanding with a section of their codebase:"connect[ing] equations to live simulations (P2)."

*Using Nod to Understand Pieces of A Codebase in Isolation.* P3 highlighted how Nod encouraged understanding by isolating a small part of their codebase to work through in a Notebook-on-Demand: "[Nod] lets you zoom into that part of the [codebase]...where now I'm just focusing on [one part of my code] (P3)." P2 and P4 also found working in a new environment with just a small section of their codebase helpful for building an understanding of the code without being overwhelmed.

*Using Nod with LLM-Generated Code.* P2, P3, and P4 also used LLM-based tools to generate code during the study period, and used Nod to "zoom into (P3)" generated code to validate and understand. P2 commented on how Nod supported them in reasoning more closely with LLM-generated code by tweaking values and inspecting intermediate values: "The feeling of knowing is different. When you tune things yourself you feel that you know things better. If someone asks a question you're more prepared to answer it, rather than just having a broad understanding of the thing (P2)."

P2 and P4 also used LLM-based tools to generate functions specifically from which to invoke a notebook-on-demand. P2 used this strategy build an understanding of particular parts of their programs: "I write down all the code, then I ask [the LLM] to make a sandbox [function] for me [which includes `notebook()`] (P2)."

*Using Nod to Communicate With Code.* P2 also used Nod to generate Notebooks-on-Demand which served as presentation aids. Because Nod converts Markdown cells to Python comments, P2 could create a function in their codebase to become a "presentation-ready notebook (P2)" when they called Nod, enabling them to build a Notebook presentation without having to leave their codebase. They said they could use Nod to "embed this whole presentation-style notebook in the code itself. That's a much better workflow (P2)."

### 6.4 Participants Used Ephemeral Notebooks-on-Demand as a Staging Ground to Experiment with Changes Before "Committing" into the IDE

*Parallel Editing and Comparison.* P2 wanted to experiment and contrast different configurations and resulting outputs for a machine learning model. They ran multiple instances of Nod to generate Notebooks-on-Demand on the same underlying source function, enabling them to edit each in parallel and maintain four concurrent alternative editors concurrently. They said of Nod that "its nice to have a self-contained [system] that disappears...I don't need [to save the notebooks], I just wanted to understand something (P2)."

*Notebooks-on-Demand were an Environment for Thinking Before Committing to Changes in an IDE.* P3 and P5 used a Notebook-On-Demand to edit their programs without immediately committing changes to their IDE source code; Nod provided a place to make code changes focused on finding a solution, only retaining code which should be incorporated into the codebase. P5 said that a notebook-on-demand provided an environment to gradually develop code into a final source code change: "having some sort of ephemeral environment where I'm not committing things, where I can try stuff out, where I can test things, where I can come up with useful exemplars, and think about what should go in my docstrings, my documentation...that's my perfect use-case for Nod (P5)" P3 phrased the idea this way: (P3)"I like the idea of coming here and playing around, touching the code but nothing in my codebase actually gets touched. I get to play around with it, mess it up, nothing is affecting what I already have [in my IDE] until I want it to be affected. That's one thing I find interesting [about Nod]." (P3)

P5 additionally remarked on how the ephemerality of a notebook-on-demand enables them to focus on building an understanding of the program without focusing on making changes are "good to be shared"(P5): "keeping it low-stakes lets me develop the theory of the program." (P5).

### 6.5 Nod Supported Organization Into Functions—Even for Functions Never Edited in Nod

Our participant pool contained (only) one participant, P1, who had previously always prioritized the dynamic manipulation available in notebooks over the code organization benefits of functions. Over the course of the study, P1 started using functions *because of knowing the dynamic interactions were still available via Nod.*

For P1, just having the option to generate a notebook-on-demand changed the way they wrote code in their IDE. Before using Nod, P1 relied on an IPython REPL for their programming work, writing Python scripts without any functions so that sequential IPython cells could be executed in the REPL (equivalent to a basic notebook functionality). Because this interaction was essential for their development workflow, any code under development could not be in a function body: "The only time I'll use a function is if I'm 100% sure I'm never going to edit it (P1)."

Nod removed the tradeoff P1 made between using functions and composing their programs line-by-line with persisted intermediate variables: "before, I couldn't get my knife in there, I couldn't get my finger in the function, so I would always literally take it out and flatten the whole thing [when I needed to edit a function body]. But now that I know I have the ability to edit within the function [with Nod] I'm not scared of them anymore (P1)". By opening a notebook-on-demand, they could program in a familiar way while also using functions.

While P1 was very enthusiastic about this change ("This whole thing is functions that I'm actually working on. This is crazy (P1)"), what surprised us most was the affect *just the option* of using Nod could have on their programming practice. P1 emphasized that Nod played a strong role in their programming even when a notebook-on-demand wasn't being used. They said that "just knowing that I have Nod if I really do get stuck and I could always go edit [the function body in a notebook-on-demand] helps a lot (P1)" and "I didn't need Nod [to write this function body], but I wouldn't have [made a function at all] if I didn't have Nod (P1)." The ability to invoke a notebook-on-demand whenever it was needed enabled P1 to write code inside of functions, even if a notebook-on-demand was ultimately not needed.

## 7 Discussion

### 7.1 Ephemeral Programming Environments

Because notebooks-on-demand were a temporary and disposable programming environment, they seemed to support participants in writing code in order to understand their programs and discover solutions in addition to making edits intended to be retained in the source code (see Sections 6.3 and 6.4). A notebook-on-demand seemed to encourage a separation of concerns between the two environments: participants treated code written in the IDE as more permanent, "production-ready" (P3) code meant to be shared with others, while the process in a notebook-on-demand could focus on writing code to "to discover what [the program] should do" [70], moving select changes back to the IDE at the end. In other words, because a notebook-on-demand was a programming environment that could be forgotten about as soon as the right change was discovered or an insight revealed, some participants found it easier to write code that was not intended to be integrated into persistent source files, but was instead oriented towards experimentation, learning, and reasoning (for which a notebook environment was well suited).

Thinking of a programming environment as *ephemeral* might open up interesting possibilities for programming systems design. Programmers engage in many interactions with their programs other than writing code that will live in the codebase long-term. Despite our widespread recognition of this, we often focus our programming environment designs on the process of writing the long-lived code. What kinds of programming environments might we build if we only need them to last as long as it takes to answer a question or discover a change?

### 7.2 Programming Systems on-Demand

Nod takes an existing programming environment (JupyterLab) and makes it *on-demand*: invokable on a user's existing program. While JupyterLab was designed as a free-standing programming environment, Nod uses the execution-time information of a running program to generate a custom JupyterLab environment. As a *style*

of bringing programming environments to users, we are excited to explore how more programming environments can be made "on-demand." Rather than thinking about a given programming environment as the *single* place a user will edit their program, how might we reorient existing editors to be used *on-demand* by users?

From a programming systems research perspective, what kinds of representations of programs might enable a more flexible interoperability between environments on a single program? Of course the most obvious answer is static program text, but as we saw in Nod, the dynamic information associated with a given program execution was crucial for getting the benefits of a notebook environment on-demand. How might future work explore more ways—and perhaps eventually more general ways—of supporting users in editing a single program via multiple environments on-demand?

### 7.3 Programming Systems Interventions into the *Relationships* Between Systems

Nod aimed to give programmers the benefits of both a notebook environment and IDE. For many of our participants, Nod addressed many key challenges with notebooks (see Section 6.2) while enabling richer dynamic interactions from an IDE (see Section 6.1). Nod achieves this not by trying to incorporate the interactions of one environment into the other, but instead thinking about how the two environments can work *in tandem* with each other.

In this sense, Nod supports new interactions by intervening in the *ecology* [8, 31, 46] of programming systems—focusing not on changing a single system but changing the *relationship* between systems. In doing so, Nod could support users with both notebook environment and IDE interactions by asking each environment to do *less*, focusing on their relative strengths instead of trying to support all of programmers' diverse needs.

Intervening into the relationship between systems instead of a single system also enables supporting interactions that would be confusing, difficult, or impossible to support in a single interface. Nod generates a programming environment that corresponds to a particular execution of the program, which does not have an obvious home in the IDE's static representation of source code. Beyond alleviating tensions, notebooks-on-demand could go further—potentially supporting new dynamic interactions, since they could work alongside an IDE rather than replace it. Programming systems can do more when they do not have to do everything.

## 8 Limitations and Future Work

As an in-situ study over ~4 weeks, the primary limitation of our study was the time required both of participants and from researchers. Interviewing, qualitative analysis, and supporting study participants with bug fixes and feature requests during the study period was the major limitation on how many study participants we could reasonably support. Recruiting more participants would have enabled us to see how Nod was incorporated among more programming practices, which could have enriched our findings.

Future work will focus on supporting some of the needs which surfaced at the end of the study: more flexibility for how non-editable text is displayed and managed in the notebook, better support for parallel, isolated Nod instances in the same directory, and support for adding Nod as an error hook, so that users can automatically invoke Nod at the site of a Python exception.

## 9 Conclusion: Interventions into the *Relationships* between Systems

This paper explores how to give users the benefits of two very different programming environments while editing one program. To do this, we built a system that can generate a Jupyter Notebook environment *on-demand* from the execution of a Python script. In a user study, we found that Nod helped mitigate challenges for notebooks and IDEs, enabled participants to understand and reason about small sections of code through dynamic notebook interactions, and provided ephemeral spaces for exploration and discovery. Making a notebook environment *on-demand* enabled Nod to also provide additional dynamic interactions, exploring how systems can play *more* to their strengths when they can be used in concert with other environments.

While Nod adds features to the JupyterLab interface, its primary intervention is not inside of a single environment but in changing the *relationship* between programming environments. Jakubovic, Edwards, Petricek have encouraged a shifting of perspective from programming *languages* design to programming *systems* design [29]. Nod raises the question, how might we shift perspectives from designing programming *systems* to designing an *ecosystem* of multiple interacting programming environments? What might we learn about programmers and programming systems by taking the *relationships* between programming systems as the sites of our research interventions?

## Acknowledgments

## Ethics and Privacy Statement

The user research discussed in this paper was approved by the IRB of the authors' institution.

## References


[1] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit - Run Behavior in Programming and Debugging. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. ISSN: 1943-6106. doi:10.1109/VL/HCC51201.2021.9576170

[2] Alex Hall. [n. d.]. Birdseye. https://github.com/alexmojaki/birdseye

[3] Association of Universities for Research in Astronomy (AURA). 2024. nrs_fss_s200a1_s200a2_workaround. https://github.com/spacetelescope/jwst-caveat-examples/blob/main/NIRSPEC_Fixed_Slit/nrs_fss_s200a1_s200a2_workaround.ipynb

[4] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 572–583. doi:10.1145/3180155.3180175

[5] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, 25–29. doi:10.1109/vlhcc.2017.8103446

[6] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *Proceedings of the 10th Conference on Innovative Data Systems*, Heiko Mueller (Ed.). Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands, 16.

[7] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. QID: Q30476519. doi:10.1191/1478088706qp063oa

[8] Susanne Bødker and Clemens Nylandsted Klokmose. 2011. The Human–Artifact Model: An Activity Theoretical Approach to Artifact Ecologies. *Human–Computer Interaction* 26, 4 (Dec. 2011), 315–371. doi:10.1080/07370024.2011.626709

[9] Souti Chattopadhyay, Zixuan Feng, Emily Arteaga, Audrey Au, Gonzalo Ramos, Titus Barik, and Anita Sarma. 2023. Make It Make Sense! Understanding and Facilitating Sensemaking in Computational Notebooks. arXiv:2312.11431 [cs]. doi:10.48550/arXiv.2312.11431

[10] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376729

[11] Luyu Cheng, Bairui Su, and Lionel Parreaux. 2026. "Live in the Editor": Low-Tech In-Situ Feedback for Plain-Text Live Programming. In *Proceedings of the Extended Abstracts of the 2026 CHI Conference on Human Factors in Computing Systems*. ACM, Barcelona , Spain, 1–5. doi:10.1145/3772363.3798318

[12] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Santa Barbara CA USA, 341–354. doi:10.1145/2908080.2908103

[13] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 33, 4 (April 2024), 101:1–101:34. doi:10.1145/3641539

[14] Robert A DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3411764.3445267

[15] Peter Deutsch. [n. d.]. The LISP Implementation for the PDP-1 Computer, 1964. ([n. d.]).

[16] Helen Dong. 2021. A qualitative study of cleaning in Jupyter notebooks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1663–1665. doi:10.1145/3468264.3473490

[17] Edward Banner. [n. d.]. Pynt. https://github.com/ebanner/pynt/blob/master/README.md

[18] Mark Richards Ford, Neal. [n. d.]. *Fundamentals of Software Architecture*. https://www.oreilly.com/library/view/fundamentals-of-software/9781492043447/ch03.html

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

[20] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, Denver Colorado USA, 579–584. doi:10.1145/2445196.2445368

[21] Jesse Harden, Elizabeth Christman, Nurit Kirshenbaum, Mahdi Belcaid, Jason Leigh, and Chris North. 2023. "There is no reason anybody should be using 1D anymore": Design and Evaluation of 2D Jupyter Notebooks. In *Proceedings of Graphics Interface 2023*. Graphics Interface, Victoria, British Columbia, Canada, 13. https://openreview.net/forum?id=Gkogn48LeI

[22] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–12. doi:10.1145/3290605.3300500

[23] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 379–390. arXiv:1608.02829 [cs.HC]. doi:10.1145/2984511.2984575

[24] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, New Orleans LA USA, 281–292. doi:10.1145/3332165.3347925

[25] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: a lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 654–664. doi:10.1145/3180155.3180165

[26] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, San Francisco CA USA, 1–18. doi:10.1145/3586183.3606733

[27] Ruanqianqian (Lisa) Huang, Brian Hempel, Yining Cao, James D. Hollan, Haijun Xia, and Sorin Lerner. 2026. Tidynote: Always-Clear Notebook Authoring. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*. ACM, Barcelona Spain, 1–17. doi:10.1145/3772318.3791919

[28] Ruanqianqian Lisa Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. 2025. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1243–1255. ISSN: 1558-1225. doi:10.1109/ICSE55347.2025.00232

[29] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023), 13. arXiv:2302.10003 [cs]. doi:10.22152/programming-journal.org/2023/7/13

[30] Joseph Weizenbaum. 1964. OPL-I: An Open Ended Programming System Within CTSS. (April 1964).

[31] Heekyoung Jung, Erik Stolterman, Will Ryan, Tonya Thompson, and Marty Siegel. [n. d.]. Toward a Framework for Ecologies of Artifacts: How Are Digital Artifacts Interconnected within a Personal Life? ([n. d.]).

[32] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, Québec City QC Canada, 737–745. doi:10.1145/3126594.3126632

[33] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. 2011. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, Santa Barbara California USA, 217–224. doi:10.1145/2047196.2047225

[34] Mary Beth Kery. 2021. *Designing Effective History Support for Exploratory Programming Data Work*. Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, Pennsylvania, United States.

[35] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Lisbon, Portugal, 147–155. doi:10.1109/vlhcc.2018.8506576

[36] 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 *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–11. doi:10.1145/3173574.3173748

[37] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[38] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. doi:10.1109/TSE.2006.116

[39] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance (TaPP'17)*. USENIX Association, USA, 17.

[40] Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. 2013. How tools in IDEs shape developers' navigation behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Paris France, 3073–3082. doi:10.1145/2470654.2466419

[41] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2012. Blaze: supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*. ACM, Austin Texas USA, 2195–2200. doi:10.1145/2212776.2223775

[42] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–7. doi:10.1145/3313831.3376494

[43] Haolin Li and Michael Coblenz. 2026. A Grounded Theory of Debugging in Professional Software Engineering Practice. *Proceedings of the ACM on Software Engineering* 3, FSE (June 2026), FSE049:1082–FSE049:1103. doi:10.1145/3797077

[44] Yanna Lin, Leni Yang, Haotian Li, Huamin Qu, and Dominik Moritz. 2025. InterLink: Linking Text with Code and Output in Computational Notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–15. doi:10.1145/3706598.3714104

[45] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29. doi:10.1145/3408991

[46] Peter Lyle, Henrik Korsgaard, and Susanne Bødker. 2020. What's in an Ecology? A Review of Artifact, Communicative, Device and Information Ecologies. In *Proceedings of the 11th Nordic Conference on Human-Computer Interaction: Shaping Experiences, Shaping Society*. ACM, Tallinn Estonia, 1–14. doi:10.1145/3419249.3420185

[47] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1093–1101. doi:10.14778/3447689.3447712

[48] Marimo, Inc. [n. d.]. Marimo. https://marimo.io/

[49] Robert C. Martin (Ed.). 2009. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ.

[50] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional evaluation with direct manipulation. *Proceedings of the ACM on Programming Languages* 2,

OOPSLA (Oct. 2018), 1–28. doi:10.1145/3276497
[51] Microsoft. [n. d.]. Visual Studio Code. https://code.visualstudio.com/
[52] Observable, Inc. [n. d.]. Observable. https://observablehq.com/
[53] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 511–525. doi:10.1145/3453483.3454059
[54] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. doi:10.1145/361598.361623
[55] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29. doi:10.1109/MCSE.2007.53
[56] Luna Phipps-Costin, Michael MacLeod, Alex Vo, Tiffany Nguyen, Joe Gibbs Politz, Shriram Krishnamurthi, and Benjamin S Lerner. [n. d.]. Combining Interactive and Whole-Program Editing with Repartee. ([n. d.]).
[57] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2021. Understanding and improving the quality and reproducibility of Jupyter notebooks. *Empirical Software Engineering* 26, 4 (May 2021), 65. doi:10.1007/s10664-021-09961-9
[58] Python Software Foundation. [n. d.]. copy — Shallow and deep copy operations. https://docs.python.org/3/library/copy.html
[59] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting Best Practices for Collaboration with Computational Notebooks. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW1 (April 2022), 87:1–87:41. doi:10.1145/3512934
[60] Sebastian Raschka and Randal S. Olson. 2015. *Python machine learning: unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*. Packt Publishing open source, Birmingham Mumbai.
[61] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9. doi:10.22152/programming-journal.org/2019/3/9
[62] Eric Rawn and Sarah E. Chasins. 2025. Pagebreaks: Multi-Cell Scopes in Computational Notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3706598.3713620
[63] Lars Reimann and Günter Kniesel-Wünsche. 2023. An Alternative to Cells for Selective Execution of Data Science Pipelines. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE/ACM, Melbourne, Australia, 129–134. doi:10.1109/ICSE-NIER58687.2023.00029
[64] M.P. Robillard, W. Coelho, and G.C. Murphy. 2004. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering* 30, 12 (Dec. 2004), 889–903. doi:10.1109/TSE.2004.101
[65] Derek Robinson, Neil A. Ernst, Enrique Larios Vargas, and Margaret-Anne D. Storey. 2022. Error identification strategies for Python Jupyter notebooks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 253–263. doi:10.1145/3524610.3529156
[66] Yvonne Rogers and Paul Marshall. 2017. *Research in the Wild*. Springer International Publishing, Cham. doi:10.1007/978-3-031-02220-3
[67] 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 (July 2019), e1007007. doi:10.1371/journal.pcbi.1007007
[68] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (Nov. 2018), 150:1–150:12. doi:10.1145/3274419
[69] Shreya Shankar, Stephen Macke, Sarah Chasins, Andrew Head, and Aditya Parameswaran. 2022. Bolt-on, Compact, and Rapid Program Slicing for Notebooks. *Proc. VLDB Endow.* 15, 13 (Sept. 2022), 4038–4047. doi:10.14778/3565838.3565855
[70] Mary Shaw. 2020. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020), 1–44. doi:10.1145/3480947
[71] Jeremy Singer. 2020. Notes on notebooks: is Jupyter the bringer of jollity?. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 180–186. doi:10.1145/3426428.3426924
[72] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers on - CASCON '10*. ACM Press, Toronto, Ontario, Canada, 174–188. doi:10.1145/1925805.1925815
[73] Krishna Subramanian, Ilya Zubarev, Simon Völker, and Jan Borchers. 2019. Supporting Data Workers To Perform Exploratory Programming. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3290607.3313027
[74] the Squeak Project. [n. d.]. Squeek. https://squeak.org/
[75] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW (Nov. 2019), 39:1–39:30. doi:10.1145/3359141
[76] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and restoring reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 138–149. doi:10.1145/3324884.3416585
[77] Zijie J. Wang, Katie Dai, and W. Keith Edwards. 2022. StickyLand: Breaking the Linear Presentation of Computational Notebooks. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3491101.3519653
[78] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3411764.3445527
[79] John Wenskovitch, Jian Zhao, Scott Carter, Matthew Cooper, and Chris North. 2019. Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure. In *2019 IEEE Visualization in Data Science (VDS)*. IEEE, Vancouver, BC, Canada, 1–10. doi:10.1109/VDS48975.2019.8973385
[80] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 152–165. https://doi.org/10.1145/3379337.3415851

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009