Plainbook: Data Science, in Plain Language

Luca de Alfaro
luca@ucsc.edu
University of California
Santa Cruz, California, USA

Mathis Aubert
maubert@ucsc.edu
University of California
Santa Cruz, California, USA

Ranjit Jhala
rjhala@ucsd.edu
University of California
San Diego, California, USA

Eliana Pastor
eliana.pastor@polito.it
Politecnico di Torino
Torino, Italy

Elena Baralis
eliana.pastor@polito.it
Politecnico di Torino
Torino, Italy

Abstract

Jupyter Notebooks have become widely adopted in data science, as they allow the sharing of reproducible computational analysis. They are, however, accessible only to people who understand computer code. To reach the broader audience of scientists interested in data analysis and computation, but unfamiliar with code, we introduce Plainbook, notebooks centered on natural language rather than code.

Plainbook is based on two principles: *promote the natural language descriptions*, and *verify the values*. In Plainbook, the natural language descriptions are preserved, rather than the resulting code; the code is generated automatically from the cell descriptions. As natural language is read top to bottom, Plainbook adopts a linear execution semantics, in which cells are guaranteed to be executed in the order in which they appear; there is no "hidden state" or out-of-order execution as in Jupyter. To allow users who may not understand code to verify the correctness of the computation, we have built into Plainbook verification mechanisms centered on *values* and value inspection. These include mechanisms that focus on individual cells, akin to unit tests, as well as global mechanisms. Both the linear execution semantics, and the verification mechanisms, are underpinned by a *snapshot kernel* that caches execution states and makes execution and verification efficient.

Keywords: Jupyter Notebooks, Data Science, AI Generated Code, Coding in Natural Language, AI Assisted Programming, Conversational Notebooks, Computational Notebooks, Linear Execution Semantics, Unit Testing, Reproducible Research, Software Verification.

1 Introduction

Jupyter Notebooks have become very widely adopted in data science, because they allow users to create and share analysis and results in a way that is *verifiable*, *reproducible*, and *extensible* [15, 19, 24, 26]. A Jupyter Notebook consists of computer code (most commonly Python, but also other languages such as R, Julia, and SQL), along with the output created by the code [27]. The output includes text output and also graphics, such as plots, diagrams, maps, and other visualizations that enable researchers and data scientists to include, in a single document, *both* the code that produced the results and the results themselves. The complete analysis can be shared in a way that makes it easy to *reproduce* and *verify* [32]. Anyone can read the code and run it to reproduce the results and check that the results included in the notebook are indeed generated by the code performing the analysis. Prior to notebooks, users had to keep extra metadata to associate the results with the analysis and code that produced them. Notebooks make this connection explicit: one can simply "re-run" the notebook to reproduce and validate the results. Further, when a notebook and its underlying data is shared, recipients can see how the results are obtained, and can modify the code to perform additional analyses or alter the settings of the analysis in the notebook.

The above is only true if the notebook authors and recipients can *read* computer code. Data science has become very widely used, and its results are often shared with people who are not computer scientists. Furthermore, since AI has become quite proficient in generating code from natural language prompts, notebooks are often created by people who *cannot write* computer code.

For example, consider the recent experience of one of the authors, who was serving on a central university committee that frequently needed to analyze data. To generate the Jupyter notebooks, the author wrote precise descriptions in natural language of the computation to be done in each cell, and prompted the AI to generate the code; the code was then run to generate the results. The author was the *only* computer scientist on the committee. The other committee members could look at the results, but to them the code was so much mumbo-jumbo, and so the verifiability and extensibility advantages of Jupyter notebooks were utterly lost to them. Had the natural language descriptions of the cells, rather than the code, been preserved in the notebooks, the notebooks would have been accessible to all.

In short, recent developments in AI are ushering in a world where computer code is generated via natural language, and computer code is becoming a lower-level language, necessary for program execution but of lesser interest to developers. Thus, the central question becomes: how can we evolve

data science notebooks so that the broader audience of non-programmers (or rather, *natural language* programmers) can benefit from their key strengths of verifiability, reproducibility, and extensibility? We posit that this question can be answered by exploiting the following two observations.

Promote the Descriptions. If the natural language descriptions used to create the cells had been preserved and centered as the primary content, the notebooks would have been far more accessible to the non-programmers on the university committee, enabling them to *modify the descriptions* to regenerate the code to satisfy their analysis needs. Of course, natural language is not quite another programming language. Despite the remarkable advances in AI, the sheer ambiguity of natural language, alongside the residual limitations of AI, make translation from natural language prompts to computer code a process where errors or misunderstandings can (and do!) easily creep in.

Verify the Values. For people to have confidence in the results produced by natural language notebooks, we require mechanisms that let the user systematically *verify* the intermediate values computed *locally* within individual cells, and *globally* across multiple stages of analysis. By focusing verification on the values, we can give the user confidence that they have implemented their intended analysis, without requiring them to inspect the generated code itself.

We used these insights to design Plainbook, a data science notebook designed to bring verifiability, reproducibility, and extensibility to non-programmers. Specifically, Plainbook is organized around the following key design decisions that implement the above observations.

1. Natural language descriptions. Plainbook does not discard the natural language descriptions used to generate cells; instead, it preserves them and presents them as the primary material of that cell.

2. Stable implementation. The code is generated from the cell descriptions via AI. To achieve predictable behavior, we save the code alongside the descriptions, and we ask AI to modify the code only if necessary to reflect changes in the descriptions.

3. Linear execution model. Plainbook adopts a strictly linear top-to-bottom execution model. Each cell is executed in the state resulting from the previous cell's execution. In particular, executing a cell twice is idempotent. This matches the way natural language works: reading a paragraph twice does not change its meaning. This also eliminates the problem of "hidden state" in Jupyter notebooks, where the state depends on the order in which cells have been executed [15, 32]. Eliminating hidden state is beneficial when working with natural language, since there is no convenient way to examine variable values.

To support this execution model, we have developed a *check-pointing kernel* that stores, and makes available for analysis, the checkpoints (the states or data values) *before* and *after* the execution of each cell, and enables notebook cells to be edited and re-run efficiently from their before-states.

4. Value verification. Relying on the check-pointing kernel, Plainbook implements new ways to verify values:

- *cell verification* lets the user use AI to verify that the code in the cell implements its natural language description,
- *cell tests* let users run a particular cell using data that is modified and often simplified per their instructions, *e.g.,* to allow users to test cell behavior in scenarios of interest and under conditions that are conducive to user inspection,
- *global tests* let the user verify relationships about the values of states spanning multiple points in the notebook, *e.g.,* to check that all data having some characteristics is still present in the notebook results after some filtering.

These verification methods can be used both by the user who creates a Plainbook, and by users with whom the Plainbook has been shared, thus implementing the central goal of allowing *verifiable* sharing. Thus, for instance, a user can verify that a cell or a whole notebook implements its natural language description; further, the user can do so using an AI of their choice.

In the following, after a review of related work, we first describe the notebook structure and its execution model. We then describe in detail the natural-language based verification tools we have implemented in Plainbook. Lastly, we describe the implementation, including what information is passed to AI for code generation, how the check-pointing kernel is implemented, and how we ensure that the code and natural language portions remain consistent as the notebook is edited.

2 Related Work

Translating from natural language to code, and more generally, generating code from natural language interactions, is one of the main uses of large AI models [5, 14, 17, 30, 34]. This use of AI has become so immensely successful that multi-billion dollar companies have made it one of their main lines of business, if not the only one: think of Anthropic with Claude [2], Google with Gemini [8], OpenAI with Codex [5], and Microsoft with Copilot [10]. Our work here certainly breaks no new ground in this respect, and in fact, Plainbook relies on the above AI models for code generation. The novel aspects of Plainbook are implemented on top of the code-generating abilities of the above AI models.

The idea of extending computational notebooks to *conversational* ones, in which computation descriptions augment and supplement code, has been advocated in [33]. That work

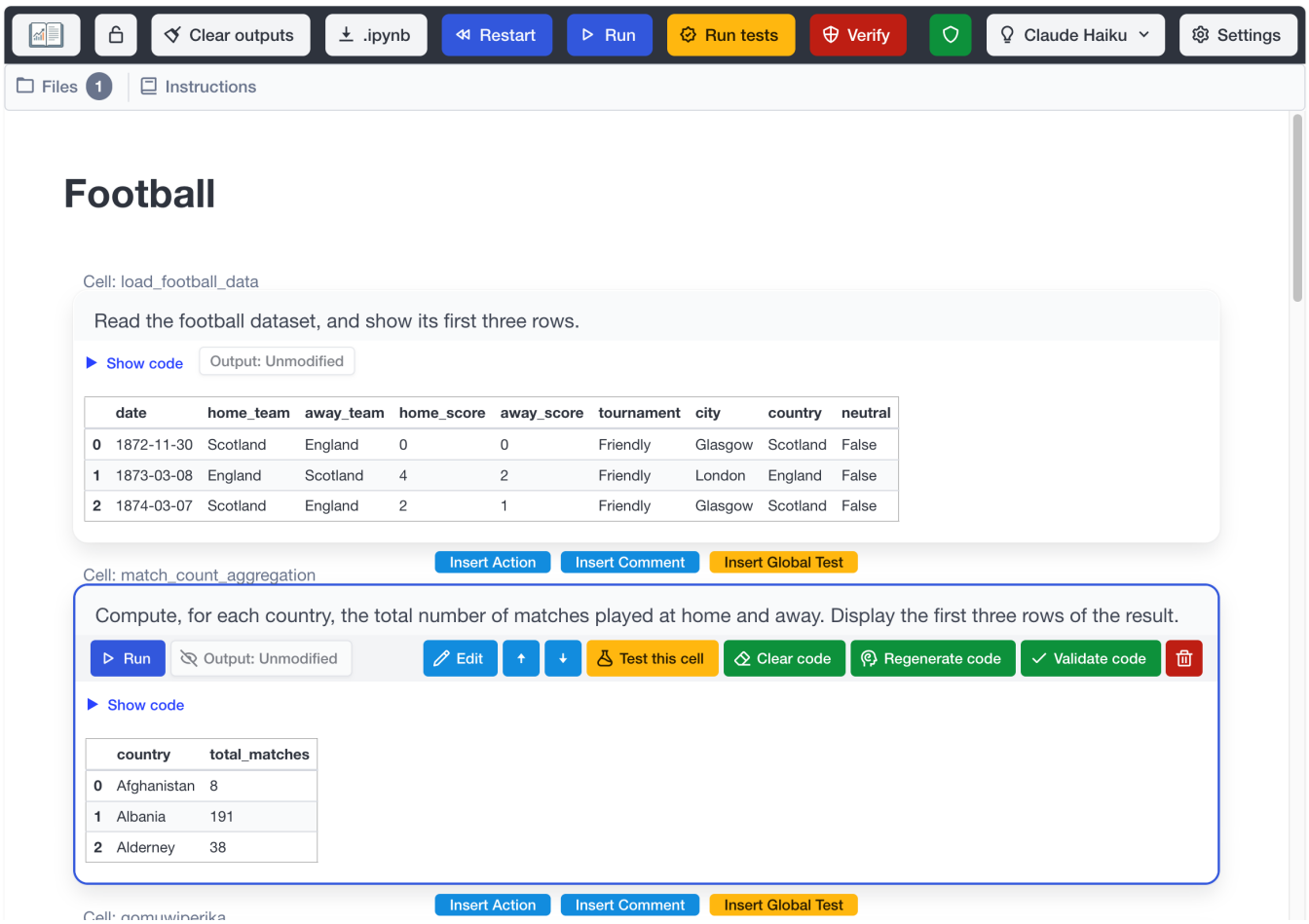


Figure 1. Top bar, and two action cell in PLAINBOOK. The top bar allows users to select the AI model in use, restarting and running all the cells, and using AI to verify that the content of the PLAINBOOK is safe. Below the top bar is a bar with a tab for files, and one for general instructions. The *Files* tab is used to select the files that are used by the notebook (in this case, the football csv dataset), so that AI knows how to access them when generating the code. The first two cells of the PLAINBOOK are shown; the second cell is focused, so that the actions available to the user are shown. Users can run the cell, validate it via unit tests (see Section 4.2), clear the code and regenerating it (possibly with a different AI model), and validate the code with respect to the description (see Section 4.1). The dataset is from [18].

details a broad spectrum of approaches for generating notebook code via interaction with AI agents, and mentions that the on-demand execution semantics of Jupyter notebooks may not mesh well with user interaction based on natural language. In this work, we follow the direction advocated by [33], and we focus on a specific approach, in which the notebook code is generated for descriptions of what each cell should do. We implement the linear execution semantics, similarly to Marimo [1, 12], and we explore how users can test notebooks using entirely natural language.

Google Colab offers both a parallel and a contrasting example [4, 11]. Similarly to PLAINBOOK, in Colab it is possible to create a new cell using AI. Users click to add the cell, then click to ask for AI help. They enter a prompt, and from the prompt, the code is produced. Users are then expected to inspect the code and accept it, and once they do, the cell persists the code — but not the original prompt. People who want to verify or modify the cell can ask AI to explain the cell content, and modify it. But AI, when explaining the cell code, generates a very detailed description of how the code *works*, rather than of what the code *does* at a high level. Hence, the AI-generated description loses the conciseness of the original high-level prompt. AI in Colab also allows users to easily modify the behavior of cells, but naturally the problem is that the original, high-level natural language description of the *current* behavior of a cell is not there. The whole idea of PLAINBOOK is to make the natural language, rather than the code, what is persisted, shown, and edited, and to make the computer code the behind-the-scenes means for execution. This change of perspective has deep consequences for how

users can generate the code, and validate that the (unseen) code responds to their specifications.

In Claude Code, Gemini, Codex, Cursor [3, 13], and similar environments, users can develop code starting from natural language descriptions. Even users unfamiliar with code can develop sophisticated systems in such a way, in a process that has become known as *vibe-coding* [7, 22]. Again, the key difference with respect to Plainbook is that the product of such tools is the code: the natural language is not present. It is true that Claude Code, Codex, Cursor, and the other tools can generate detailed and beautifully written explanations of the work done, but this is not equivalent to retaining the cell descriptions. In such tools, code is generated iteratively via a sequence of prompts, which incrementally specify the computation and fix any problems with the implementation. In contrast, in Plainbook the code is generated from a single, comprehensive natural language description for each cell. This natural language specification persists, and the code can at any time be recreated from it; the tests are also in natural language and can be used against any newly generated code. Another consequence of the persistence of natural language is that when a Plainbook is shared, the recipient can directly read the natural-language description of the computation, and then use AI to check that the code implements it.

Literate programming was proposed by Knuth as a way to inter-twine the natural language description of *what* the code does, and *how* the code does it [20]. In spirit, what it does is close to what we aim to do. Of course, at the time when literate programming was created, there was no way to generate code from natural language, and thus it is the natural language itself that contains code primitives as inserts.

Jupyter notebooks are the direct inspiration for Plainbook [19, 27]. Jupyter notebooks have had an enormous impact in the way in which data science and research in computing is done, as they allow for the joint presentation of results, and of the code that generated them [15, 26, 28]. The joint presentation makes it possible for anyone to reproduce the results, and thus validate them. As the original code is present, it can be easily modified to experiment with variations of the analysis, or extended to perform additional tasks. Many environments enable the AI-assisted creation of Jupyter notebooks, among which Colab.

In a Jupyter notebook, each cell can be executed on demand, and the same cell can be executed multiple times. While this provides great flexibility, it also generates the problem of *hidden state*, as the state of the notebook at a certain point may depend on more than the cells currently present in the notebook [15, 21, 26]. For instance, one can execute a cell that creates some variables, then delete the cell, and the variables are still defined (part of the *hidden state*). The flexibility provided by this per-cell execution model is often welcome, but it can also make notebooks harder to create and understand, and it is a leading cause of re-execution and reproducibility failure [25]. Marimo (for Python) [1, 12] and Pluto.jl (for Julia) [9, 29] implement, as Plainbook, a top-to-bottom execution semantics that eliminates the hidden state. The methods used are different. In Marimo, the absence of hidden state is achieved via a sophisticated analysis of the dependency between variables [12]. Thus, Marimo can infer which cells need to be re-run when changes are made to ensure compliance with the semantics; when cells are deleted, the variables they introduced are also removed. In Plainbook, we enforce adherence to the top-to-bottom semantics via our checkpointing kernel. Our approach has some drawbacks, as we will discuss, such as an inability to cope with open files across cells. Nevertheless, our approach is well-suited to support our verification tools. It may be interesting, in future work, to explore the development of a variant of Plainbook that relies on a Marimo-based infrastructure for code execution.

3 Plainbook: Syntax and Semantics

We begin by describing the syntactic *structure* of a Plainbook instance (or notebook, in short) that highlights how it promotes the natural language cell descriptions while ensuring implementation stability, and its semantics or *execution model* that enables validation via value inspections.

3.1 Syntax

Figure 1 shows a Plainbook notebook in action. A notebook consists of a *top bar*, followed by a sequence of *comment*, *action* and *test* cells. The top bar, as shown in Figure 1, enables global operations on the notebook, including running all its cells, restarting from the beginning, and selecting the AI backend used for code generation and verification. Comment cells are identical to Jupyter markdown cells: they are used to provide documentation or explanation within the notebook. Test cells are used to validate the values, and we will describe them at length in Section 4.

The bulk of the computational work of a notebook is carried out in action cells, much like Jupyter code cells. However, the key difference is that the action cells of Plainbook promote the natural-language description of what the cell should do. The code is generated automatically via AI, and it is stored to achieve implementation stability, but it is displayed only upon request. Figure 1 displays the first two actions cells of the Plainbook. In the first, the description instructs the AI to read a football dataset and display its top rows. In the second cell, the description asks to compute the number of matches each country played at home and away, and again display the first rows of the result.

Users can *focus* on a cell at a time by selecting it. Once a cell is focused, a series of buttons and options appear. Some buttons allow the user to *edit* the description and *generate* the code from the description. Other buttons allow *clearing* the current code, so it can be regenerated from a clean slate, and

verifying that the code faithfully implements its description. The buttons and their semantics will be described in more detail below .

Plainbook notebooks are stored in Json, in the format used for Jupyter notebooks extended to accommodate descriptions, as well as testing and validation mechanisms. For action cells and for the test cells discussed in the following, we store both the descriptions and the code in order to achieve implementation stability; obviously, we also store all outputs so that notebooks can convey results and information without the need for re-running them. We denote with $(d_1, c_1), \ldots, (d_n, c_n)$ the n action cells of a notebook, where d_i is the description of cell i, and c_i is the code of cell i, for $1 \leq i \leq n$.

3.2 Semantics

In a Jupyter notebook, cells can be executed in any order, and the state of a notebook depends on the order of cell execution, rather than on the position of cells in the notebook [25]. For instance, if cells are deleted, their effects persist in the notebook state, yielding what is known as the *hidden state* of Jupyter notebooks. The flexibility afforded by arbitrary execution order can enable quick experimentation, and is useful when users can easily access the notebook variables, which carry the state. However, hidden state is a hindrance for natural language users, who cannot easily keep track of the variables, and of how the code in the cells updates them.

Plainbook is driven by language, and the cell descriptions constitute a narration of the computation. In a narration, there is no notion that reading something twice changes its meaning. That is, a user who cannot see the variable state more directly might be baffled by the fact that running the same description one more time changes the state of the notebook.

Linear Execution: Cells and States. Thus, we designed Plainbook to have a *linear execution* semantics, where the cells are executed in order, and each cell is guaranteed to be executed in the state that results from the *previous* cell's execution. Suppose that $c_1, \ldots, c_n$ is the code associated with the cells of a notebook. Execution begins from the empty state s_0. The code c_i of each cell i is executed in state s_{i-1}, and produces state s_i. This linear semantics ensures that each cell i yields a *unique* state s_i that is precisely the result of executing $c_1, \ldots, c_i$ starting from a blank slate s_0. This linear semantics ensures that executing a cell i is idempotent: the resulting state s_i is simply the result of executing $c_1, \ldots, c_i$ in sequential order. In addition to eliminating hidden state, Plainbook's linear semantics crucially allows the user to associate a deterministic set of *values* s_i to each cell i, which then lets them validate those values, as described in Section 4.

The checkpointing kernel. While a state s_i *can* be computed simply by processing the notebook from the beginning, in practice, that would be terribly inefficient, as a user may want to iteratively work on a single cell, and it would be wasteful to rerun the entire history on each iteration. To support our linear semantics, we have implemented a *checkpointing kernel* that enables state caching and makes state (re-)construction efficient (see Section 5).

4 Validation and Testing

Jupyter notebooks are centered on code, with the assumption that users are able to understand the code they are writing, and can read the code present in notebooks that are shared with them. However, when notebooks are created by, or shared with natural-language programmers, the code itself is inaccessible, and thus the key advantages of Jupyter notebooks — verifiability, reproducibility, and extensibility — are lost. Plainbook preserves these benefits, by making the natural language descriptions, rather than the code, front and center.

Users describe computation using natural language, and rarely look at the code, evaluating it mostly from the results it produces. This new usage mode raises two questions. How can users be confident that the code implements their descriptions? And, if the notebook has been authored by others and shared with them, how can they be sure that executing it is safe?

Our key observation is that while Plainbook users may not be able to evaluate code, they can understand data. Thus, Plainbook provides methods for verifying implementation correctness based on AI, and on data inspection. Next, we describe two mechanisms that Plainbook provides for *validation* and *testing*, which can be applied *locally* to individual cells, or *globally* across multiple cells.

4.1 Validation

Cell Validation. By clicking on the *Validate code* button of a particular cell, users can ask AI to verify that that cell's natural language specification is consistent with its code implementation. To perform the validation, Plainbook sends all information used to generate the code to AI; the AI system prompt asks the AI model to reply *Yes* if the code corresponds to the description, and to reply *No* and provide diagnostic information otherwise. If the validation result is negative, the user can click on *Regenerate code*, and Plainbook will ask AI to regenerate the code, taking into account the problem diagnostics. Figure 2 illustrates an example in which validation fails to compute the total number of years in which a country has played. The diagnostic explains that the generated code is incorrect because of a "double-counting" error and what the correct approach should be.

Safety via cross-validation. Obviously, if one uses for verification the *same* AI API used for code generation, the answer is very likely to be that the code is correct: the strength of the check lies in the ability to use a *different* AI than the one that generated the code, avoiding dependence on a single

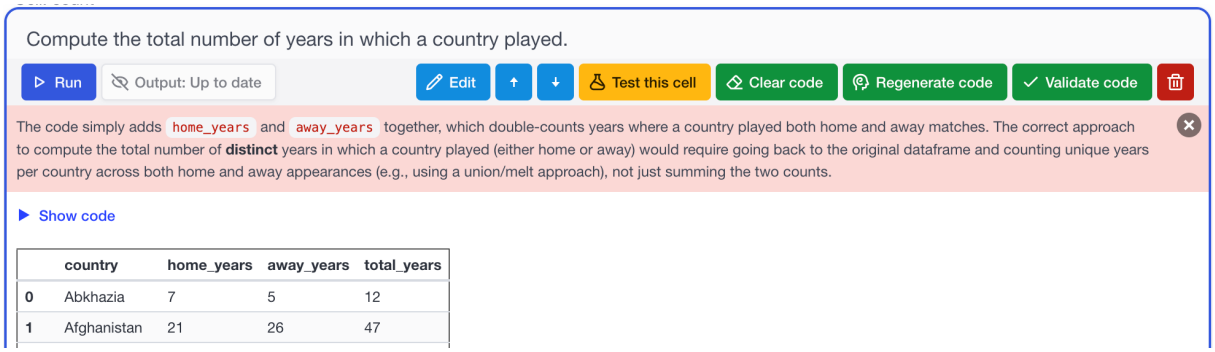


Figure 2. PLAINBOOK can ask AI models to validate a cell's code against its description. In this case, validation fails, and PLAINBOOK displays diagnostic explanations returned by AI.

AI source. Indeed, it has been shown that AI models can contain hidden vulnerabilities, buried in the models in order to perform attacks [16, 23, 31]. The attacks can be quite specific, triggered only when the code is generated in particular domains, or for specific applications, and within a range of target dates; the attacks can be quite hard to detect, as the AI API would perform normally for most users under most circumstances. By allowing the cross-validation of code with respect to multiple independent AI APIs, PLAINBOOK side-steps these vulnerabilities.

Global Validation. When a user opens a notebook shared by someone else, they can use the top-bar *Verify* button (see Figure 1) and ask PLAINBOOK to check that the entire notebook's cells correctly implement their descriptions. Additionally, this process checks that the notebook as a whole does not contain any dangerous operations, defined as operations that can delete files, leak information, or alter the setup of the local host. These validations are not perfect, of course, as code can be obfuscated. Nevertheless, the ability to use an AI API of the recipient's choice in verification adds a layer of protection.

4.2 Cell Testing

While AI is a powerful tool for validating notebook correctness, it does not eliminate the need for a more direct verification method, in which the user directly assesses the correctness of the results. Since the user may not understand code, we have built verification tools that are based on data, namely, *cell tests*, which exploit PLAINBOOK's linear execution semantics — and lack of hidden state — to enable users to systematically verify the values produced by the cells.

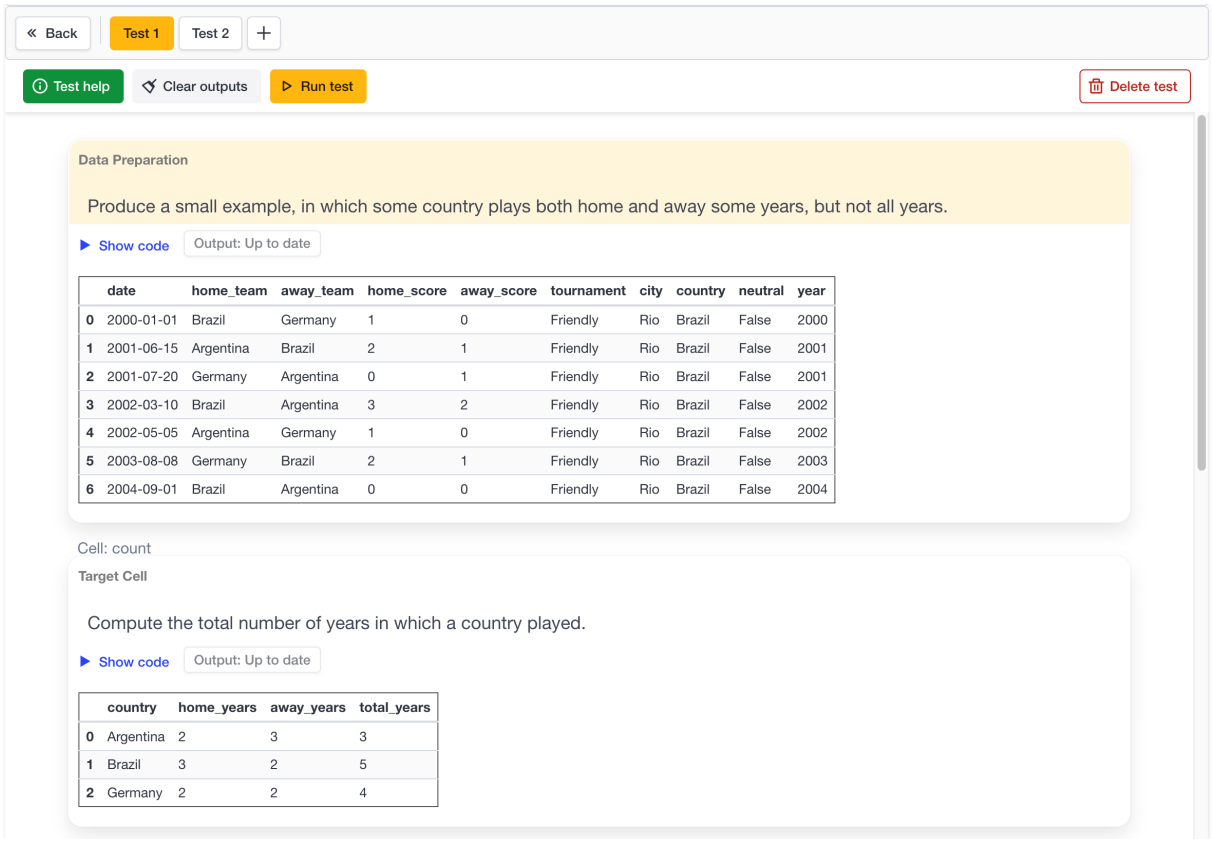
Cell Test Flow. The execution flow used to perform a cell test on a target cell i is depicted in Figure 4. Recall (Section 3.2) that in the normal execution flow of the notebook, cell i is executed in state s_{i-1}, which is the result of executing all cells from the beginning of the notebook up to $i-1$. In the cell test flow, a *data preparation* cell α is used to simplify some components of the state s_{i-1}, producing a new state s_α in which the code c_i of the target cell is executed. The user can then either directly examine the output, or write a validation cell (not used in the above Figure 3) to check the resulting state s_β.

For example, recall the problem of counting the unique years in which a country played, and the failure of the cell validation in Figure 2. A user can verify by inspection that the cell in Figure 2 has been fixed by creating a *cell test*, shown in Figure 3 In the cell test, a *Data Preparation* cell is used to produce simplified data, that sub-samples the full input data set according to some criteria specified in natural language. The code of the *target cell* of the test can now be run on this simplified data, and the user can verify by inspection that the output is correct, or specify a validation cell to check it.

Testing without code modification. PLAINBOOK's linear semantics allow a user to test *any* cell without any modification to the cell code. As shown in the cell test flow in Figure 4, the *same* code of a cell (here, c_i) can be run in *two* different environments: the regular environment (namely, s_{i-1}), and the test environment (namely, s_α). In contrast, with Jupyter notebooks, one can only test code that is *wrapped* in functions or methods, so that it can be called under both regular and test settings, and consequently, much code ends up not being testable.

Testing without data generation. A second benefit of PLAINBOOK's linear semantics is that users can leverage the execution model to create suitable test data for each cell. In a typical unit test, a user is faced with the chore of cooking up appropriate test data from scratch. Instead, in PLAINBOOK, the user already has suitable test data, namely, the state s_{i-1} in which the target cell normally runs (see Figure 4 again). All they need to do is simplify or modify some components in order to produce inpectable tests, which itself can be done via natural language in a *data preparation* cell.

Testing multiple functionalities. Finally, a single target cell can have multiple tests associated with it. For instance, in Figure 3 there are two tests, as indicated in the tab bar; only the first one is displayed. The tests allow users to test



	date	home_team	away_team	home_score	away_score	tournament	city	country	neutral	year
0	2000-01-01	Brazil	Germany	1	0	Friendly	Rio	Brazil	False	2000
1	2001-06-15	Argentina	Brazil	2	1	Friendly	Rio	Brazil	False	2001
2	2001-07-20	Germany	Argentina	0	1	Friendly	Rio	Brazil	False	2001
3	2002-03-10	Brazil	Argentina	3	2	Friendly	Rio	Brazil	False	2002
4	2002-05-05	Argentina	Germany	1	0	Friendly	Rio	Brazil	False	2002
5	2003-08-08	Germany	Brazil	2	1	Friendly	Rio	Brazil	False	2003
6	2004-09-01	Brazil	Argentina	0	0	Friendly	Rio	Brazil	False	2004

	country	home_years	away_years	total_years
0	Argentina	2	3	3
1	Brazil	3	2	5
2	Germany	2	2	4

Figure 3. A cell test for verifying that the target cell of Figure 2, once fixed, correctly computes the number of unique years in which a country has played. The Data Preparation cell sets up a simple example, enabling the user to assess the correctness of the target cell by inspecting its output.

different functionalities of the target cell. For instance, one or two tests can be used to test the cell under normal data, and further tests can indicate its behavior under pathological or missing data.

4.3 Global Testing

Plainbook' linear execution model allow us to implement *global tests*, which let users to specify assertions that relate the state of the notebook across *different points* in the execution, to check that the overall computation, over the standard data, satisfies properties of interest.

Specifying Global Tests. Global tests crucially depend on Plainbook's linear semantics: specifically, that the result of executing the first i cells always produces the state s_i. To enable global tests, Plainbook automatically generates short nicknames for each such state s_i, which are then used to create *namespaces* that hold the (checkpointed) state s_i. Global tests can then use the cell nicknames to specify the states from which the variable values should be taken, thereby writing properties or assertions that relate the values of different states. Figure 5, shows a global test that checks that all the countries defined after another reference cell nicknamed `cell_country` are preserved at the current cell.

Implementing Global Tests. Plainbook executes global tests, by constructing a composite namespace in which all namespaces (states) of previous cells are included as sub-name-spaces. In this composite namespace, the variable `x` of a cell with `nickname` can be accessed as `__state__nickname.x`. Thus, the composite namespace allows an assertion to *simultaneously* access the states of *all* previous cells. The AI system instructions for global tests detail this access scheme, enabling AI to generate test code that contains the appropriate variable accesses.

In our experience, users mainly use global tests to check overall properties, and in particular, that data is appropriately preserved or discarded during filtering.

5 Implementation

Plainbook's key design goals: promoting the natural language computation descriptions, and enabling value validation, rely crucially upon two abilities. First, the AI needs to be able to use a cell's description, and other context information available from previous computation, to *generate*

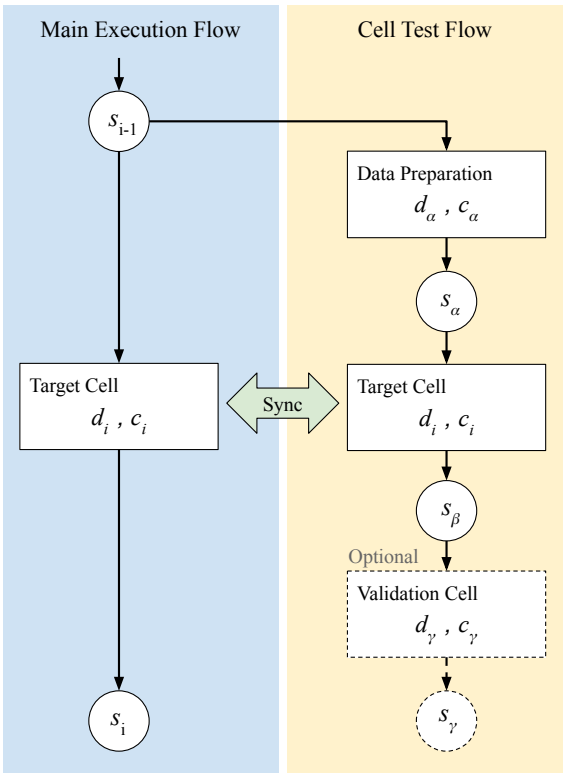


Figure 4. Execution flow of a cell test. To test a target cell c_i, Plainbook executes it in a parallel flow, where the data s_{i-1} has been simplified into s_α by a data preparation cell. Users can check s_β by inspection, or via a validation cell.

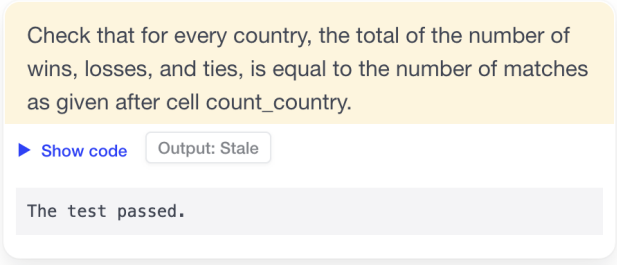


Figure 5. A global test. The test relates the state immediately preceding the test, with the notebook state after cell `count_country`, which created a count of countries. The test checks that all the football matches that each country played are properly accounted.

the code that faithfully implements the user's natural language description for that cell. Second, we need to be able to execute the generated code and faithfully implement Plainbook's linear execution semantics. In Sections 5.1 and 5.2 respectively, we describe how Plainbook implements these two key capabilities.

Python as the Foundation. Python serves as the foundational language for Plainbook, powering both its code generation and execution engines. This choice has three motivations. First, AI is quite proficient at generating Python, as training data is very abundant, including in the data science realm. Second, Python is already in use in (Jupyter) notebooks, and has widely-used libraries for data science and visualization. Third, the meta-programming and dynamic execution capabilities of Python make it easy to implement the kernel and testing harnesses we built into Plainbook, as we shall describe below.

5.1 Generation

Plainbook generates code for each cell by querying an AI API with a combination of *global* system instructions, and a *local* (cell-specific) prompt. (Currently, Plainbook supports Claude Code or Gemini, but other AI APIs can be added easily).

Local Prompt. The prompt for generating the code c_i for a cell i was determined via experimentation, and includes:

- The **description** d_i of the cell to be generated. The AI is instructed to return code that implements this description.
- If a previous code implementation **code** c_i for cell i already exists, we include it in the request, asking the AI to revise the code, if necessary, to implement d_i. The AI request is phrased so that the code is revised only if the current implementation does not implement the description.
- The **descriptions** d_j and **code** c_j of all previous cells $1 \le j < i$. This creates context for the AI, so that it can understand the previous work done in the notebook, and the variables and modules that are in the scope.
- The **variables** defined in the state s_{i-1} in which the cell will be executed. We provide to the API a list of variables, each with type information (but not value information). For Pandas dataframes, we include information on the names and types the columns, so that the AI can understand how to operate on the dataframe.
- Optionally, the **output** of the previous cell c_{i-1}. This is useful to allow descriptions to refer to that output as well; for instance, one can write "increase that value by 10%, and select all data where the price is above that threshold".

Global System Instructions. In addition to the above, some global contextual information regarding files and domain-specific instructions are added to each AI request:

- **Files.** An AI API does not have access to the user's filesystem, and cannot know where files of interest are. To allow for file access, Plainbook includes a *files tab* where users can select the locations of files used

in the notebook, such as datasets. In this way, the AI knows which paths to include in the code.
- **Instructions.** Some projects have domain-specific background knowledge that is useful for AI generation: for example, methods for accessing data, other APIs available, and so forth. Plainbook includes a *instructions tab* where users can specify such background information.

For example, Figure 1 shows the beginning a notebook to analyze a football (soccer) dataset. The *Files* tab specifies the location of the `football.csv` file in the user machine's filesystem. The AI API (here, Claude Haiku) is thus made aware of the location of files that the user might mention, and is able to translate the cell's description into code that loads the dataset.

Privacy and Cost Considerations. Including cell output in the AI input can pose a privacy risk, as cell output can include sensitive information such as customer identities or data. For this reason, Plainbook enables to toggle on and off the sharing of cell outputs with AI on a per-notebook basis. In any case, to reduce AI cost, we do not include graphics such as plots and figures in the information relied to AI. We do not include the value of the variables (beyond the names of dataframe columns) in the information passed to AI. Similarly to outputs, including these values incurs the risk of leaking confidential information. Further, having as context the code $c_0, \ldots, c_{i-1}$ that generated the variables seems sufficient in practice.

5.2 Execution

Plainbook's linear execution semantics are the key to enabling hidden state and associating a deterministic values that can be validated of each cell and across multiple cells (Section 4) However, a naive implementation of the linear execution semantics, which, simply re-runs the notebook from the beginning each time a cell is executed, would be terribly inefficient.

A better approach could be to repurpose the Jupyter kernel, where cells are executed in the current kernel state: the resulting state becoming the new current state. To implement Plainbook's linear semantics, specifically, to execute cell i, we could check the index of the last executed cell: if it was $i-1$, we could simply execute cell i; otherwise, we restart the kernel and execute cells $1, 2, \ldots, i$. Sadly, while better than the naive implementation, this repurposed kernel would still be rather inefficient. In particular, users often refine a particular cell, until they are satisfied with its behavior. Re-running the notebook each time from the beginning would make this common use case most inefficient.

A Checkpointing Kernel. To implement our linear semantics efficiently, we have developed a *checkpointing kernel*, which stores the states obtained after executing each cell in the proper order. Recall that in our linear model, execution starts the empty state s_0, and each cell c_i is executed in state s_{i-1} to produce state s_i. With checkpointing, if cell c_i is executed multiple times, for instance, as the user is iteratively refining their prompt, only the code c_i needs to be run, producing each time new states s_i.

Checkpoints as Python Namespaces. The checkpointing kernel stores states as Python namespaces. This induces one limitation: the states cannot track *external* state, such as the state of open files. Fortunately, our typical users have the same limitation. Users who perform data science in natural language typically read and write data within the scope of individual cells, rather than carrying open file descriptors across multiple cells. To support the need to restart execution (for example, to re-read external files that changed), Plainbook includes top-bar button to *clear* all kernel states, ensuring a full restart.

Implementing the Plainbook Kernel. We have implemented Plainbook as a web application that runs from localhost, similarly to the classic `jupyter notebook` module. The checkpointing kernel is implemented as a stand-alone Python process, communicating with Plainbook via HTTP; a kernel is created automatically for each notebook. Relying on HTTP(s) makes it possible to run the kernel on a different machine from where the UI is located, even though we are not using this capability in the current version of Plainbook.

5.3 Orchestration

To efficiently and interactively implement Plainbook's linear execution semantics, the Plainbook kernel needs to orchestrate the re-generation and re-execution of cells, depending on the specific operations performed on the notebook. Orchestration is tricky because (recall, from Section 5.1) the prompt for generating the code for cell c_i includes descriptions of the *variables* defined in the state s_{i-1} which provide important context for the code generation task.

Suppose, for example, that a user generates code for cells 1 through 10, and then goes back and *changes* the description of cell 4, so that now perhaps also the content of a variable has changed meaning. At this point, the Plainbook kernel must also mark the code for cells 4 through 10 as invalid. At the same time, the kernel need not *regenerate* those cells eagerly. Perhaps the user wants to *iterate* applying other changes to cell 4. Nevertheless, the kernel must remember that *eventually*, if the user wants the values for the cells $4-10$, then the code and values for those cells must be regenerated.

Plainbook implements orchestration by using *indices* to track cell validity, and then using the indices to compute the minimal set of cells that need to be generated or executed to efficiently implement Plainbook's linear semantics for any operation performed on the notebook.

Kernel State: Indices. To determine the cells that need code regeneration or execution, Plainbook tracks the indices of the last action cell:=

- whose *code* is valid: i_{code},
- whose *output* is valid: i_{out}, and
- whose code was *executed*: i_{exec}.

Code and output are preserved in the stored notebook, making it possible to load a notebook and display it complete with its results, as usual. Correspondingly, the indices i_{code} and i_{out} are stored with the notebook. The Plainbook kernel maintains the invariant that

$$i_{code} \geq i_{out} \tag{1}$$

which captures the intuition that valid output can only be generated by valid code.

Conversely, the states of the execution kernel are not stored; rather, when a notebook is read, the states must be re-created by rerunning the notebook from the beginning. Consequently, the execution index i_{exec} is not stored with the notebook, and is set to 0 when a notebook is read or created.

Kernel State: Effects of Operations. Next, we describe how the kernel *effects* the indices according to the operations performed on the notebook. Certain operations like code generation and execution each have a *precondition* on the indices that must be true before the operation can be performed. As we explain shortly, the kernel recursively regenerates or re-executes cells as needed, to ensure that the preconditions for the requested operation are satisfied.

Reading the notebook. When the notebook is read, Plainbook sets $i_{exec} := 0$, and we leave i_{code} and i_{out} unchanged from the values they have in the stored notebook. This allows users to read a notebook, whether produced by themselves or shared with them, and still consider the code and output valid, unless the notebook is further modified. Note that a notebook is fully verifiable, as both the associations of descriptions and code, and the associations of code and output can be checked at any time, as discussed in Section 4.

Editing the notebook. When the description of cell i is edited, or when cell i is deleted or inserted, or when cells i and $i+1$ are swapped in the notebook, Plainbook invalidates the code and output beyond position i:

$$\begin{aligned} \text{Effect :} \quad & i_{code} := \min\{i_{code}, i-1\} \\ & i_{out} := \min\{i_{code}, i-1\} \end{aligned}$$

Code generation. Code can be generated for a cell i when $i_{out} \geq i-1$, so that the information on the variable context for c_i is available. If code can be generated, then the code c_i becomes valid, and the outputs for any cell beyond $i-1$ become stale, as c_i still needs to be run.

$$\begin{aligned} \text{Precondition :} \quad & i_{out} \geq i-1 \\ \text{Effect :} \quad & i_{code} := i \\ & i_{out} := \min\{i_{out}, i-1\} \end{aligned}$$

Code execution. Code can be executed if it is valid, and if the output from the previous cell is valid, indicating that the state s_{i-1} in which to execute c_i is present. Note that executing a cell does not cause the output of subsequent cells to become stale. The conditions for executing cell i are:

$$\begin{aligned} \text{Precondition :} \quad & i_{code} \geq i \\ & i_{exec} \geq i-1 \\ & i_{out} \geq i-1 \\ \text{Effect :} \quad & i_{out} := \max\{i, i_{out}\} \\ & i_{exec} := \max\{i, i_{exec}\} \end{aligned}$$

Execution in Plainbook is meant to be idempotent, and for this reason, if $i_{out} \geq i$, when the user calls for the execution of cell i, the kernel is not used; rather, we simply use the cached state s_i and the cached output of the previous execution.

Orchestration using Kernel Indices. Recall, from Section 3 and Figure 1, that Plainbook has a variety of buttons that enable the user to generate (or regenerate) the code for a single cell and run either a single cell or the entire notebook.

When one of the buttons is pressed, calling for execution or code generation, Plainbook invokes an orchestration engine that computes the minimum set of code generation and execution that is needed to comply with the request, and performs the sequence of generations and executions. For instance, assume that:

- A notebook with n cells is read, so that $i_{code} = i_{out} = n$,and $i_{exec} = 0$.
- The description of cell i is edited.
- The execution of cell k, for $i < k \leq n$, is requested by clicking on the Run button of cell k.

Then, the orchestration engine would, in order:

- Run cells $1, \ldots, i-1$ to generate the context s_{i-1} and the output and variables needed for generating the code c_i of i;
- Generate the code c_i for cell i, and run it;
- For each cell $j \in \{i+1, \ldots, k\}$, in order, first generate the code c_j using the context of $j-1$, and then run it.

The orchestration engine is implemented in the Javascript front-end of Plainbook, so that the sequences of code generation and execution can be easily interrupted if desired. The back-end, driving the kernel execution, checks the preconditions and implements the effects of all notebook operations, thus ensuring the consistency of the notebook state.

6 Discussion

We put Plainbook to the test by implementing the main data analysis tasks in the university committee.

6.1 Strengths

This exercise was successful. Most often, individual analysis steps were simple, and they could be described via short prompts that generated obviously correct results.

Iterative. For complex tasks, the ability to perform *cell tests* to verify the computation performed by cells was essential. In these more complex cases, we proceeded in an iterative fashion, refining a cell's prompt until the results seemed correct. The linear execution semantics and checkpointing kernel were helpful in such an iterative approach, as we could focus on perfecting the current cell, without having to worry about restarting the execution from scratch to avoid carrying over the bad state from the previous cell version. Once the cell seemed to work, we could use *cell tests* to try it on smaller data and verify by inspection that it behaved correctly. Generating simplified test data via natural language was quick and effective. The example reported in Figure 3 was typical: in cell tests, we sought simple data, where the results could be checked by inspection; we often specified some aspects of the simplified data, to test corner cases.

Accessible. Plainbook notebooks were usable by those with little coding experience. Even among those familiar with code, they became a favorite for an unexpected reason: they were often *faster* to use than Jupyter notebooks augmented with AI. In Plainbook, one needed only to type a prompt (e.g., "group the students by major, and plot their graduation time") and click *Save and Run* (or press shift+Enter). With Colab or VSCode, one needed more clicks to call up the AI helper, then generate the code, accept it, and run it.

Collaborative. When performing data analysis collaboratively in the haste of a meeting, this speed difference mattered. It also mattered that any participant could suggest a prompt to be run (whereas only a few would have been able to suggest code). The ability to iterate on a cell without having to worry about rerunning the notebook from the beginning (the need for which is quite unclear to the uninitiated) was also a positive.

6.2 Limitations and Future Work

Configuring API Keys. The primary barrier to adopting Plainbook was the requirement for AI API keys. While the Plainbook settings provide direct links to the necessary provider pages, the process of securing keys and configuring billing accounts remained a significant hurdle for users.

External State. The chief limitation of the checkpointing kernel is its inability to track external (to Python) state. In practice, users tend not to leave files open across notebook cells, as the concept of a "file descriptor" is not a natural one for non-programmers. However, database access is more problematic. Once connected to a database, if a cell with the prompt "add to the database a student with a major in Biology" is executed twice, the result is not idempotent: two students are added. It remains future work to extend the linear semantics to databases via transactions and logs. So far, Plainbook is amenable to analysis tasks where data can be read, and written, in one fell swoop that fits in a single cell. Fortunately, many data science tasks fall in this category.

Code Regeneration. In settings where Plainbook can be used, there are still rough spots in usability, which we plan to address in future work. Most significantly, when a user modifies the prompt of one of the early cells in the notebook, Plainbook regenerates the code and runs all subsequent cells (see Section 5.3). The code is regenerated because if we change the code c_i for a cell i, the variables available for the execution of cell $i + 1$ (and more subtly, the *meaning* of the variables available) may change. Therefore, to ensure correctness, we regenerate the code of cell $i + 1$ before executing it, so that the changes in variable availability and meaning can be accounted for. Regenerating the code of all cells that follow a modified cell is both slow and overly conservative: typically, the code of most, if not all, such cells is still valid. In future work, we will seek to limit code regeneration by asking AI to first identify which cells need their code regenerated.

Code Availability

Plainbook is an open source project [6]. The code of both Plainbook and the checkpointing kernel that it uses is released under the BSD 3-clause license. Plainbook can be installed using the pypi `pip` package manager.

References

[1] Akshay Agrawal and Myles Scolnick. 2023. *marimo - an open-source reactive notebook for Python.* https://github.com/marimo-team/marimo Original-date: 2023-08-14.

[2] Anthropic. 2024. *The Claude 3 Model Family: Opus, Sonnet, Haiku.* https://www.anthropic.com/news/claude-3-family Model Card.

[3] Anysphere, Inc. 2026. *Cursor: The AI Code Editor.* https://cursor.sh Developed by Anysphere.

[4] Ekaba Bisong. 2019. *Google Colaboratory.* Apress, Berkeley, CA, 59–64. doi:10.1007/978-1-4842-4470-8_7

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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. doi:10.48550/arXiv.2107.03374 arXiv:2107.03374 [cs].

[6] Luca de Alfaro. 2026. *Plainbook*. https://github.com/lucadealfaro/plainbook GitHub repository, accessed: 2026-06-29.

[7] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – a Grey Literature Review. *arXiv* (2025). doi:10.48550/arxiv.2510.00328

[8] Gemini Team, Google. 2023. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805* (2023).

[9] Migran N. Gevorkyan, Anna V. Korolkova, and Dmitry S. Kulyabov. 2023. Julia language features for processing statistical data. *Discrete and Continuous Models and Applied Computational Science* 31 (2023), 5–26. doi:10.22363/2658-4670-2023-31-1-5-26

[10] GitHub. 2026. *GitHub Copilot*. https://github.com/features/copilot AI-powered code assistant.

[11] Google. 2026. *Google Colaboratory*. https://colab.research.google.com/ Accessed: 2026-05-08.

[12] Péter Ferenc Gyarmati, Dominik Moritz, Torsten Möller, and Laura Koesten. 2025. A Composable Agentic System for Automated Visual Data Reporting. *arXiv* (2025). doi:10.48550/arxiv.2509.05721

[13] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2026. Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. In *23rd International Conference on Mining Software Repositories (MSR '26)*. ACM.

[14] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (Nov. 2024), 1–79. doi:10.1145/3695988

[15] 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. doi:10.1109/ICSE55347.2025.00232 tex.ids= huangHowScientistsUse2025b ISSN: 1558-1225.

[16] Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M. Ziegler, Tim Maxwell, Newton Cheng, Adam Jermyn, Amanda Askell, Ansh Radhakrishnan, Cem Anil, David Duvenaud, Deep Ganguli, Fazl Barez, Jack Clark, Kamal Ndousse, Kshitij Sachan, Michael Sellitto, Mrinank Sharma, Nova DasSarma, Roger Grosse, Shauna Kravec, Yuntao Bai, Zachary Witten, Marina Favaro, Jan Brauner, Holden Karnofsky, Paul Christiano, Samuel R. Bowman, Logan Graham, Jared Kaplan, Sören Mindermann, Ryan Greenblatt, Buck Shlegeris, Nicholas Schiefer, and Ethan Perez. 2024. Sleeper Agents: Training Deceptive LLMs that Persist Through Safety Training. doi:10.48550/arXiv.2401.05566 arXiv:2401.05566 [cs].

[17] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (Feb. 2026), 1–72. doi:10.1145/3747588

[18] Mart Jürisoo. 2024. International football results from 1872 to 2024. https://www.kaggle.com/martj42/international-football-results-from-1872-to-2017. Accessed: April 1, 2026.

[19] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, and Kyle Kelley. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: players, agents and agendas: proceedings of the 20th International Conference on Electronic Publishing*. IOS press, 87.

[20] Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111. https://academic.oup.com/comjnl/article-abstract/27/2/97/343244

[21] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-Grained Lineage for Safer Notebook Interactions. doi:10.48550/arXiv.2012.06981 arXiv:2012.06981 [cs].

[22] Christian Meske, Tobias Hermanns, Esther von der Weiden, Kai-Uwe Loser, and Thorsten Berger. 2025. Vibe Coding as a Reconfiguration of Intent Mediation in Software Development: Definition, Implications, and Research Agenda. *arXiv* (2025). doi:10.48550/arxiv.2507.21928

[23] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[24] Elizaveta Pertseva, Melinda Chang, Ulia Zaman, and Michael Coblenz. 2024. A Theory of Scientific Programming Efficacy. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–12. doi:10.1145/3597503.3639139

[25] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE, 507–517. https://ieeexplore.ieee.org/abstract/document/8816763/?casa_token=SYvtfPxWrAQAAAAA:Bd5tTsysiHUvaruWodNagYTi7bA9f3B5_mJhOM0TXAksn4UJGO066Snq7W9Jlib1bhuWsh5BfA

[26] 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 (2021), 65.

[27] Fernando Pérez and Brian E. Granger. 2007. IPython: a system for interactive scientific computing. *Computing in science & engineering* 9, 3 (2007), 21–29.

[28] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. Using the Jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 1–2.

[29] Nicholas W. M. Ritchie. 2022. Reproducible Spectrum and Hyperspectrum Data Analysis Using NeXL. *Microscopy and Microanalysis* 28 (2022), 478–495. doi:10.1017/s143192762200023x

[30] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, and Tal Remez. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023). https://arxiv.org/abs/2308.12950

[31] Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. 2024. SALLM: Security Assessment of Generated Code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops* (Sacramento, CA, USA) *(ASEW '24)*. Association for Computing Machinery, New York, NY, USA, 54–65. doi:10.1145/3691621.3694934

[32] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–30. doi:10.1145/3359141

[33] Thomas Weber and Sven Mayer. 2024. From Computational to Conversational Notebooks. doi:10.48550/arXiv.2406.10636 arXiv:2406.10636 [cs.HC].

[34] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023). https://arxiv.org/abs/2311.10372