Flowco: Rethinking Data Analysis in the Age of LLMs

Stephen N. Freund
freund@cs.williams.edu
Williams College
Williamstown, MA, USA

Brooke Simon
bsimon2000@g.ucla.edu
University of California, Los Angeles
Los Angeles, CA, USA

Emery D. Berger*
emery@cs.umass.edu
University of Massachusetts Amherst / Amazon Web Services
Amherst, MA, USA

Eunice Jun
emjun@cs.ucla.edu
University of California, Los Angeles
Los Angeles, CA, USA

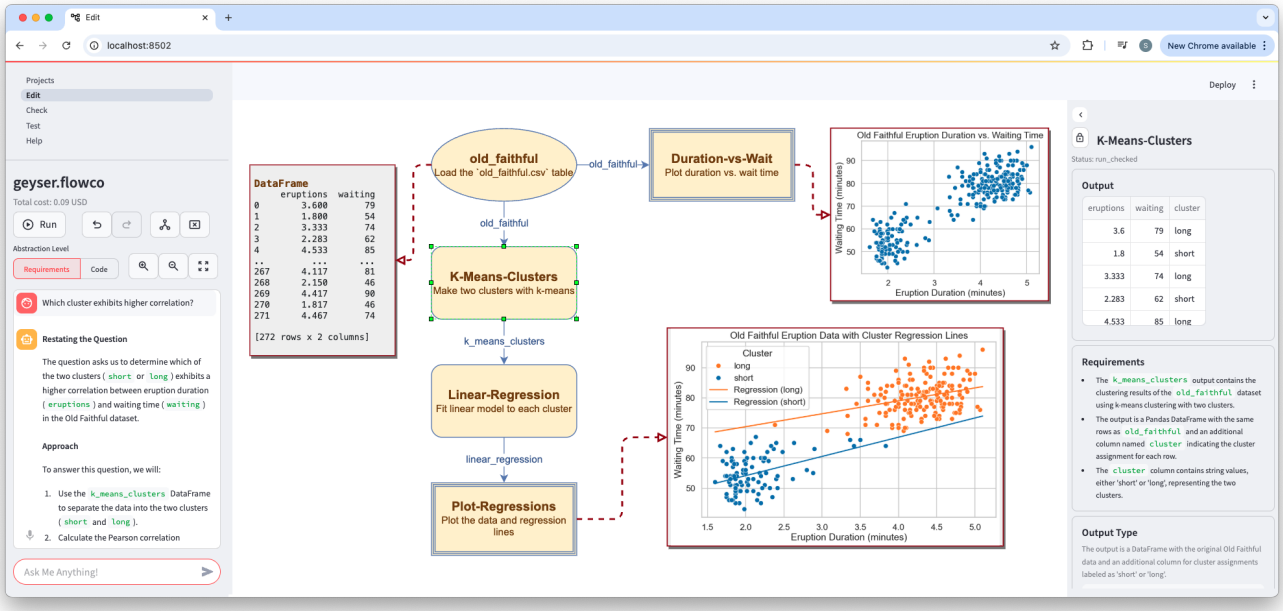


Figure 1: Flowco is a mixed-initiative system that leverages a visual dataflow programming model and LLMs to not just synthesize code but also assist in all stages of development. In this example, the analyst creates a workflow to read Old Faithful geyser data, clusters eruptions by duration, and fits a linear model to each cluster. Flowco translates this graph into executable code via an LLM and exposes a variety of other interactions to assist the analyst, including (i) a chat box with direct access to the data, graph, and outputs as well as (ii) user-defined checks to validate generated code.

Abstract

Conducting data analysis typically involves authoring code to transform, visualize, analyze, and interpret data. Large language models (LLMs) are now capable of generating such code for simple, routine analyses. LLMs promise to democratize data science by enabling those with limited programming expertise to conduct data analyses, including in scientific research, business, and policymaking. However, analysts in many real-world settings must often exercise fine-grained control over specific analysis steps, verify intermediate results explicitly, and iteratively refine their analytical approaches. Such tasks present barriers to building robust and reproducible analyses using LLMs alone or even in conjunction with existing authoring tools (e.g., computational notebooks). This paper introduces Flowco, a new mixed-initiative system to address these challenges. Flowco leverages a visual dataflow programming model and integrates LLMs into every phase of the authoring process. A user study suggests that Flowco supports analysts, particularly those with less programming experience, in quickly authoring, debugging, and refining data analyses.

1 Introduction

Data analysis often involves incrementally transforming and visualizing data, posing hypotheses, and applying statistical methods through highly iterative exploratory workflows. Computational

*Work done at the University of Massachusetts Amherst.

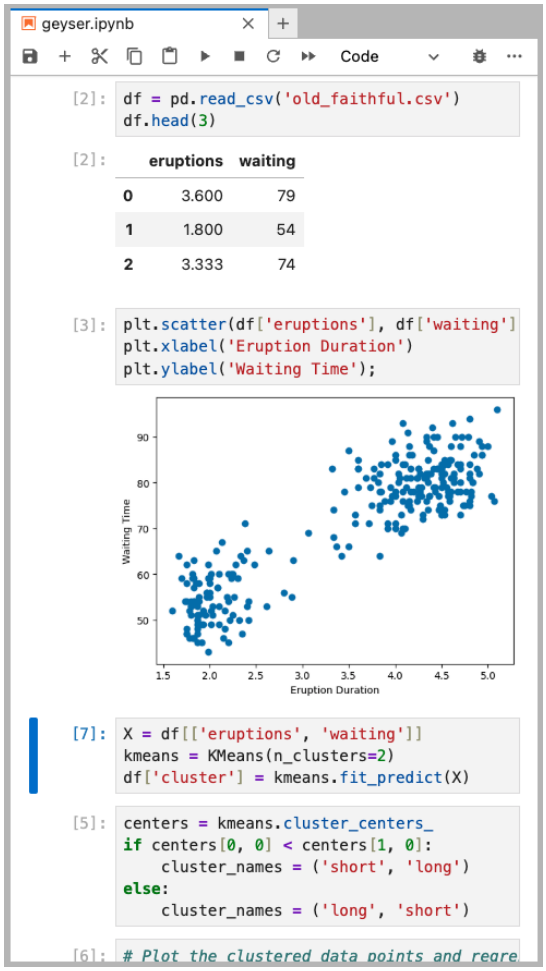


Figure 2: A Jupyter notebook exhibiting a potentially stale variable. To the left of each cell is the execution count, which indicates the order in which the cells were evaluated. The cell labeled with execution count 7 (highlighted) was evaluated after `cluster_names` was initialized in the cell with execution count of 5. The latter depends on the former, meaning that `cluster_names` could be stale.

notebooks support these workflows, but with some limitations and challenges [24, 41], and large language models (LLMs) have the potential to drive the creation of even more robust and lower-overhead analysis authoring tools. However, they too have limitations that must be carefully addressed to ensure successful outcomes in data analysis authoring workflows. This paper presents a new approach for supporting mixed-initiative authoring of data analyses with LLMs that overcomes those obstacles.

Computational notebooks. Figure 2 shows a Jupyter notebook that loads geyser eruption data [31], plots the raw data, and then clusters into two groups in preparation for a linear regression analysis.

The very same features that make notebooks effective for exploratory programming also lead to a collection of well-known impediments to building robust workflows in them [24, 41]. They lack modularity and abstraction, making it challenging to reason about the overall structure of a notebook, track dependencies across cells, and reproduce results. These issues become particularly salient as the number of cells grows and their contents evolve. Further, notebooks are ill-suited for the standard unit testing and validation practices used to help provide assurances about the quality of code in traditional software engineering [38].

The potential for non-linear execution order and stale values can lead to confusion and errors in analysis that are difficult to diagnose and fix. For example, in Figure 2, the cell labeled with execution count 7 (highlighted) was evaluated after `cluster_names` was initialized in the cell with execution count 5. The latter depends on the former, meaning that `cluster_names` could be stale.

Many tools and services aim to recover dependencies [15, 42] and address other shortcomings of notebooks [39, 49]. These tools help mitigate many challenges of using notebooks, but they do not fully address the need for modularity and abstraction in data science. Modularity and abstraction are necessary to ensure the reproducibility of data analyses throughout development.

Large language models. At the same time, large language models (LLMs) present opportunities to quickly author, debug, and iterate on code. LLMs also present new challenges: authoring effective prompts, detecting issues in generated code, and managing possibly drastic differences in the code produced in response to repeated queries or requests for small modifications, among others.

Consider a scientist attempting to use an LLM to generate the above data analysis workflow. They start by asking the LLM to write a script to read in the geyser data. After creating that code, the scientist inspects the first few rows of the data and and then asks the LLM to plot the data as a scatter plot. Seeing the distribution of points, the scientist proceeds to prompt the LLM to write code to cluster the data and compute a linear regression model for each cluster. Running the resulting code results in the error

```
X does not have valid feature names, but LinearRegression
was fitted with feature names.
```

because the regression model was fitted on a pandas DataFrame (which has column names), but predictions are being made on a plain NumPy array in the LLM-generated code. The scientist asks the LLM to fix the error, and also asks the LLM to use the names "short" and "long" to distinguish the two clusters.

After making those changes, the scientist runs the code and sees that the plot is not as expected. The LLM has plotted waiting time on the x-axis and eruption duration on the y-axis, which is the opposite of what the analyst wants. Inspecting the code, they discover the regression was performed on the wrong variables. This time, the scientist finds it easier to directly add a line of code enforcing categories and now asks the LLM to update the rest of the code to match that edit. While the LLM provides an executable script after a few attempts, the scientist remains unsure of the quality of the code or validity of the results.

Mixed-initiative authoring with LLMs for data science. There are many current attempts to combine LLMs and computational notebooks in order to amplify their strengths [4, 9, 32, 36]. These tools focus on incorporating chat interfaces into notebooks. While potentially beneficial in some cases, such use of LLMs can exacerbate

the challenges of producing modular, robust, easily understood, and testable data science workflows. Moreover, these tools do not eliminate the need for programmers to reason about the code they are writing in a notebook.

Flowco. This paper introduces Flowco, a new mixed-initiative system to address these challenges when authoring data analyses. Flowco leverages a visual dataflow programming model and LLMs at every phase of the authoring process. Users draw dataflow graphs in which nodes specify concrete steps in an analysis and edges indicate the flow of information. The Flowco implementation of the geyser analysis is shown in Figure 1. Flowco leverages the classic dataflow model for two key reasons:

- Dataflow graphs provide a natural medium for exploration, design, and communication of data analysis workflows (section 2.4). This insight comes from our observations about external artifacts scientists already rely upon to plan, discuss, and share their data analyses.
- Dataflow graphs enjoy strong modularity, abstraction, and composition properties. Those features not only support the creation of robust, reusable, and reproducible data analysis workflows from a programming perspective but also provide a natural representation of the computation and its components when interacting with LLMs.

Flowco translates dataflow graphs into executable code via an LLM and exposes a variety of other interactions to give the analyst fine-grained control. These include ways to edit individual nodes at different levels of abstraction; an "Ask Me Anything!" (AMA) chat agent with direct access to the data, graph, and outputs; and assertion checking and unit testing capabilities to gain confidence in the generated code and analysis.

We perform two different evaluations. First, we demonstrate the applicability of Flowco over a diverse range of analysis goals and complexity. Second, we conduct a user study with 12 students familiar with data science but with a range of programming experience. We find that these analysts are able to successfully use Flowco, find the dataflow programming model helpful for their organization and visualization of multi-step analyses, and preferable to using ChatGPT to perform similar tasks. Participants also found Flowco most useful for getting started with analyses, especially for those with little programming experience.

This paper contributes the following:

- a dataflow programming model featuring multiple abstraction layers that serves as a foundation for reliable LLM-centric programming;
- Flowco, a system for authoring data analysis workflows with LLM assistance at every stage of the process;
- a collection of fully automatic and user-guided LLM-based techniques for designing, implementing, validating the correctness Flowco dataflow graphs; and
- validation that Flowco can effectively support data analysis authoring tasks, as demonstrated by examples and a user study with twelve data science students.

2 Background and Related Work

2.1 Computational Notebooks for Data Science

Computational notebooks, most notably Jupyter Notebook [37], are widely used programming environments for data analysis. Notebooks support rapid iteration and exploration, where analysts generate many code versions as they work through implementation details before converging on a design and implementation ("expand-then-reduce") [24].

However, notebooks can very quickly become complex and difficult to maintain. For example, Kery et al. found that while notebooks do not inherently limit the document length (i.e., number of cells), analysts run into practical length limitations when scrolling across cells to run them in specific orders [24]. A common practice among notebook users is to create new notebooks that contain only the most important cells after initial stages of exploration [10, 24]. Interestingly, Dong et al. found that modularizing code by creating functions and classes was the least common "cleaning" activity among data scientists using notebooks [10]. A key tension is that notebooks are primarily helpful for getting started with and iterating on programs rapidly, but analysts quickly face code maintenance issues that require more involved strategies (e.g., modularization, composition, documentation) that would take them out of exploration [41].

Previous work has contributed several ideas and systems for improving code quality without compromising exploration in notebooks: extracting "clean" notebooks through program slicing [16]; grouping and hiding cells in a notebook through "cell folding" [40]; grouping and annotating cells to support sensemaking [7]; laying cells out two-dimensionally [14]; end-user-defined scoping of variables inside notebooks [39]; and visualizing cell dependencies in a notebook [49]. Flowco is most similar to prior work on dataflow notebooks [27]. While Wenskovitch et al. generate dataflow graphs from notebooks to help users understand their code, Flowco gives users direct control over the dataflow graphs. Unlike dataflow notebooks, Flowco integrates programming support from LLMs. Furthermore, in dataflow notebooks, the data scientist still encounters the challenges of maintaining dependencies during rapid exploration, which may result in multiple versions of the same cell, all of which are represented as different dataflow paths. However, with Flowco, the structure of the dataflow graph captures all dependencies between computation steps, relieving the user from the burden of reasoning about them. Finally, Flowco can also generate a notebook from the dataflow graph.

2.2 Interacting with LLMs

LLMs can further accelerate programming through code generation capabilities, but they come with their own challenges. Recent work has observed that structured problem decomposition and generation can reduce the metacognitive demands and challenges of using generative AI [45]. One technique to decompose problems is through chaining. Chaining has become a standard technique for decomposing complex tasks into subtasks for LLMs to tackle [50]. Wu et al. find that decomposing and chaining tasks leads to improved performance and completion success [50].

Building on this work, Arawjo et al. develop ChainForge [5], a system for constructing prompts and evaluating the outputs of

LLMs. ChainForge provides a visual programming interface that is similar to a dataflow graph. Arawjo et al. report "surprise" that ChainForge users wanted to use ChainForge to specify data analysis pipelines, which they could not easily do. This paper explores this finding further. Flowco also goes beyond this prior work to develop techniques for ensuring reliability of LLM-generated code in the context of data science. Specifically, Flowco provides users with multiple layers of abstraction, automatic change detection and propagation, and behavioral validation to control and understand LLM code generation. While ChainForge focuses on providing user support for prompt engineering, Flowco is focused on using LLMs for data analysis.

CoLadder introduces a hierarchical approach to decomposing programmer intent into sub-goals [52]. An LLM can address each sub-goal. While our goals are similar in supporting intent decomposition and code composition, there are a few major distinctions: (1) Unlike Flowco, CoLadder does not provide any architecture for ensuring that the composed code is executable or correct programmatically. (2) While CoLadder guides users through task decomposition in a domain-agnostic way, Flowco takes advantage of dataflow graphs to describe task decomposition. Notably, Figure 12 in the CoLadder paper is similar to a dataflow graph. By focusing on dataflow graphs, we hypothesize that our work is approachable to data scientists across many different levels of expertise. While not focused on programming with creating inputs to LLMs or programming with LLMS, Graphologue [19] supports decomposition of LLM outputs. Similarly, Sensescape [44] supports visual and hierarchical exploration of LLM outputs in order to support information foraging and sensemaking.

The multiple layers of abstraction that Flowco provides are high-level descriptions, requirements, and code. Zamfirescu-Pereira et al.'s PAIL IDE [53] explores similar layers of abstraction to support users in designing programs. While PAIL is focused on surfacing requirements, design decisions, and code as a way to guide code generation and summarization, Flowco uses these layers of abstraction in order to generate and validate code (section 4.1).

More specifically in the data analysis space, Kazemitabaar et al. explore two different approaches for integrating LLMs into authoring data analyses with a focus on validation: phasewise and stepwise steering [23]. The authors conclude by outlining the need for interaction flows and affordances for steering LLMs. Flowco realizes this.

While prior work has given users more fine-grained control over the process of programming with LLMs, several limitations persist: (1) the absence of assurances that the generated code is correct programmatically, meaning connected pieces of code are not guaranteed to execute (e.g., CoLadder); (2) the lack of compositionality of code fragments generated in different "runs"; and (3) absence of incremental (re)computation when modifications occur in parts of the LLM-generated program.

2.3 Combining Computational Notebooks and LLMs for Data Science

The recent proliferation of code generated by LLMs has also made data science programming more accessible. Given the wide adoption of computational notebooks and the ability to generate code with LLMs quickly, how to combine both computational notebooks and LLMs for data science is an open research topic and the focus of many emerging products [4, 9, 32, 36].

However, the use of LLMs can further exacerbate the challenges of designing robust, modular, and well-test data analyses. Indeed, scientists relying on current conversational interfaces to LLMs must critically assess whether the generated code is correct and aligns with their scientific objectives to avoid errors that threaten statistical and scientific validity, a daunting task for even expert programmers [8].

An alternative approach to putting LLMs into notebooks and notebooks into LLMs is necessary. Our goal is to maintain the benefits of rapid code generation, program exploration, and iteration while also mitigating the issues of modularity, composability, and extensibility. In response, we develop Flowco to prioritize these qualities. A key idea in Flowco is to design for these qualities by leveraging a data structure common in data science and emerging as beneficial in the interactive LLMs literature: dataflow graphs.

2.4 Dataflow Graphs

In our previous observations of and collaborations with scientists, we found that scientists outline key computational steps, connect steps that depend on each other, and refine the steps and connections. How scientists describe, formulate, and communicate their analyses is similar to how one describes and structures dataflow graphs.

Dataflow graphs and programming models are a classic concept in computing with a long and rich history dating back to the 1960s and 1970s [20]. Dryad [18], Naiad [30], and TensorFlow [1], as well as the widely used LabVIEW [26] visual programming language, are among the many prominent contemporary examples of dataflow-based systems.

Visual programming dataflow languages enable users to organize complex computations into manageable, independently understandable, and composable components. Further, dataflow graphs provide a framework well-suited for exploratory programming, where users can quickly prototype and test ideas by interactively modifying the graph and observing the results [20]. A variety of data science and data analytics tools embrace visual dataflow graph programming models for these reasons. Representative examples include Texera [48], Alteryx [3], KNIME Analytics Platform [25], and Altair RapidMiner [2]. There have also been efforts to extract dataflow graphs from existing code [35].

Dataflow graphs in Flowco provide the programming model for organizing, defining, and reasoning about analyses. Further, dataflow graphs provide a natural framework for facilitating the creation, composition, testing, and maintenance of code generated by LLMs. Dataflow graphs are thus not only a useful model for the user but also serve as a foundation for reliable LLM-centric programming. That insight is a key contribution of Flowco.

3 Usage Scenario

The Flowco user interface, as shown in Figure 3, is is divided into three panels: The *project panel* on the left encompasses global actions and the "Ask Me Anything!" (AMA) chat box; the *canvas* in the center is the visual editor for Flowco dataflow graphs; and

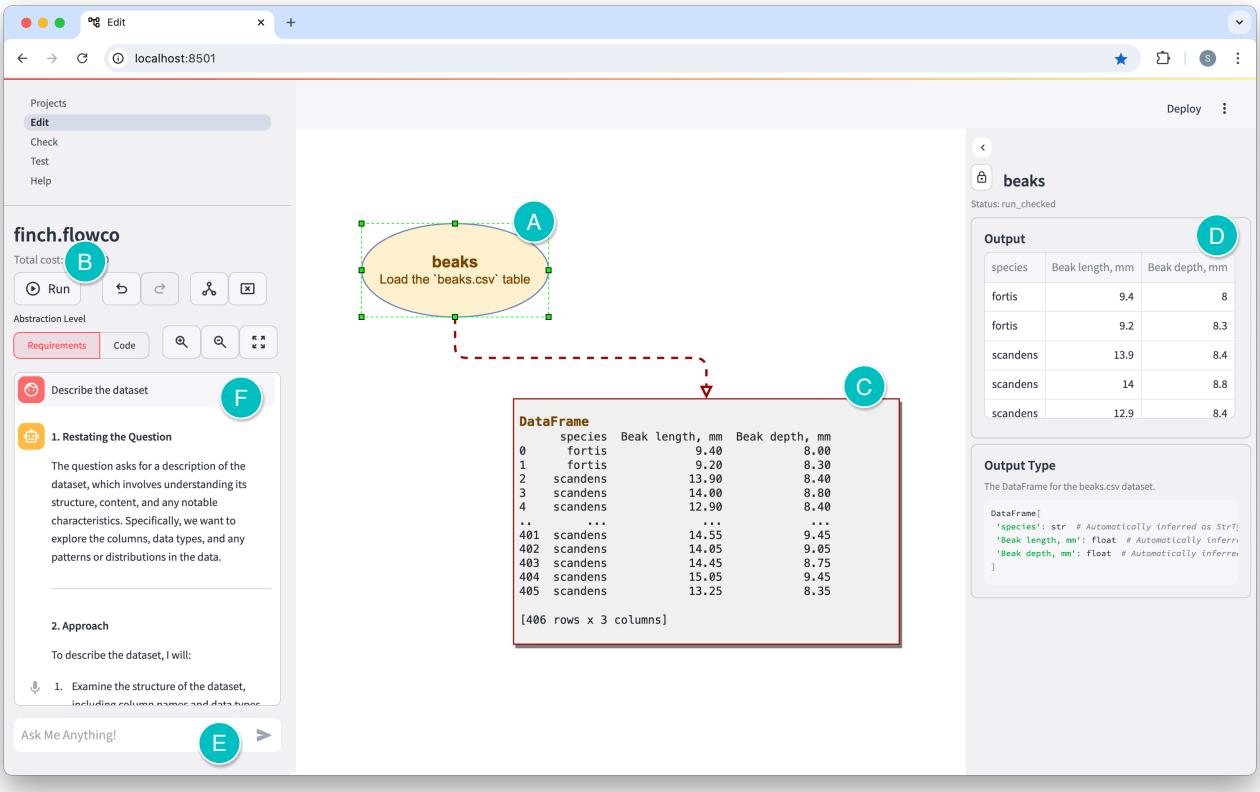


Figure 3: The Flowco editor interface is divided into three panels: (Left) the *project panel* encompasses global actions and the "Ask Me Anything!" (AMA) chat box; (Center) the *canvas* is the visual editor for Flowco dataflow graphs; and (Right) the *details panel* presents details of the selected node during editing. A The user creates a new node to load the dataset in `beaks.csv`. B The user presses the Run button to synthesize code to evaluate the dataflow graph. C After evaluating the node, Flowco provides a sample of the dataset in the canvas. D The user examines the full dataset in the details panel. E The user prompts the AMA chat box to "Describe the dataset". F Flowco responds as it performs a number of analyses on the dataset.

the *details panel* on the right presents details of the selected node during editing. The LLM is an integral part of Flowco, which uses it to support data exploration, dataflow graph creation and editing, code synthesis, validation, and error detection and recovery[1].

The following usage scenario illustrates the most salient aspects of Flowco's design. A researcher, Alex, conducts exploratory, confirmatory, and estimation analyses on a dataset containing beak length and beak depth measurements for two species of finches, Fortis and Scandens [2].

Exploring the Data. Alex begins by clicking on the initial blank canvas to create a new node. A Alex specifies via the node creation dialog that the node should load a dataset from the file `beaks.csv`. B Alex then clicks the Run button. C Flowco then synthesizes requirements and Python code for that node and then runs it, after which Flowco provides a sample of the dataset in the canvas. Alex selects the node and D examines the full dataset in the navigable view presented to the right in the details panel. That panel also includes the type of output produced by that node. In this case, the output is a `DataFrame` with columns for species, beak length, and beak depth.

After inspecting the output, E Alex uses the AMA chat box and asks that Flowco "Describe the dataset." Flowco leverages its underlying LLM to answer that question. The LLM utilizes Flowco to run code it creates to examine the shape and column types for the dataset, to verify that no values are missing, and to compute basic descriptive statistics for each numerical column. F The LLM reports on those steps in its response. The LLM even runs code to generate histograms for the numerical columns and, leveraging its multi-modal capabilities, visually inspects those plots to conclude for beak lengths:

[1]Flowco's default LLM is OpenAI's GPT-4o [34]. That model supports a number of features Flowco leverages, including multi-modal inputs, tool calls, streaming, and structured output.
[2]The data is drawn from a larger dataset [12] gathered during a long-running study of finches on Daphne Major Island, Ecuador [13].

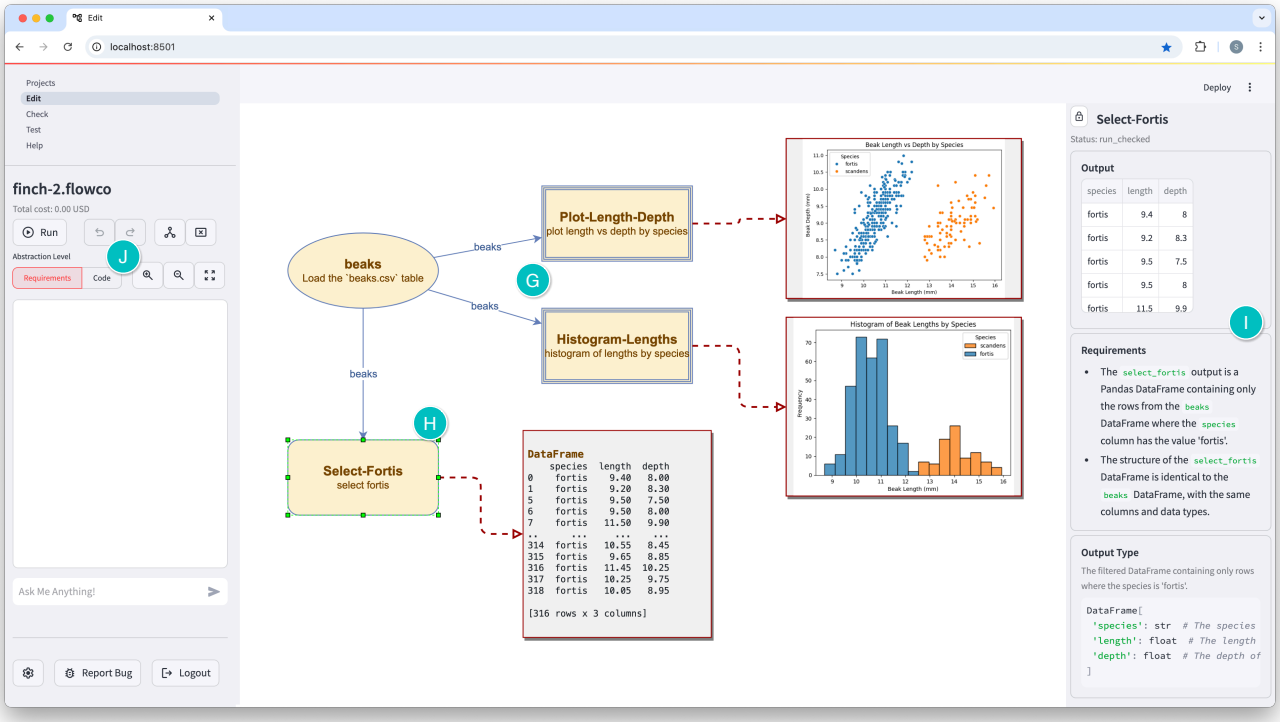


Figure 4: G The user adds two plotting nodes to the graph, as well as H a node to select only the Fortis finches from the dataset. After running the graph, I the user selects Select-Fortis to examine it in the details panel. J The user exposes the synthesized code by selecting the "Code" abstraction level.

> *The distribution appears bimodal, with two peaks around 11 mm and 14 mm. This suggests the presence of two distinct groups or species with different beak lengths.*

and for beak depths:

> *The distribution is unimodal and roughly symmetric, centered around 9 mm.*

Visualizations. Equipped with this knowledge, G Alex adds two more nodes, each of which will produce a plot.[3] The node **Histogram-Lengths** generates a histogram of the beak lengths, and the node **Plot-Length-Depth** plots beak length vs. depth. The 'Run' command again synthesizes requirements (using the structure of the graph, the requirements of predecessor nodes, and the labels to infer intent), Python code, and finally the computed output. After seeing the initial version of those plots, Alex instructs Flowco to "Use different colors for different species" via the AMA chat box. The LLM updates the plotting nodes to reflect that request, leading to the final versions shown in Figure 4.

Computation and Analysis. After inspecting the plots, Alex asks via chat whether the difference in beak lengths between the two species is statistically significant. As before, the LLM leverages its ability to run code and employs a Kruskal-Wallis hypothesis test to conclude that the difference is statistically significant with a p-value of 1.5×10^{-47}.

Alex decides to next focus on the Fortis finch species and estimate the average beak length for that species using the sample in the dataset. To do so, H Alex adds the computation node **Select-Fortis** to select only the rows for Fortis finches from the dataset, clicks Run, and I then selects the new node to examine the node in the details panel. For computation nodes, the details panel shows not just the output and output type, but also the requirements for the node's behavior inferred by Flowco.

They decide to inspect the code for it by J selecting the "Code" abstraction level, which reveals the synthesized Python code at the bottom of the details panel:

```
Code

# put all imports here
def select_fortis(beaks: pd.DataFrame) -> pd.DataFrame:
    filtered_beaks = beaks[beaks["species"] == "fortis"]
    return filtered_beaks
```

As shown in Figure 5, Alex continues to K create nodes to estimate the mean beak length for the Fortis finches using bootstrap resampling and the percentile method. In particular, Alex adds

[3] In Flowco, the shape of a node indicates its behavior: an oval represents a data-loading node, double-bordered rectangle represents a plotting node, and a rounded rectangle represents a computation node.

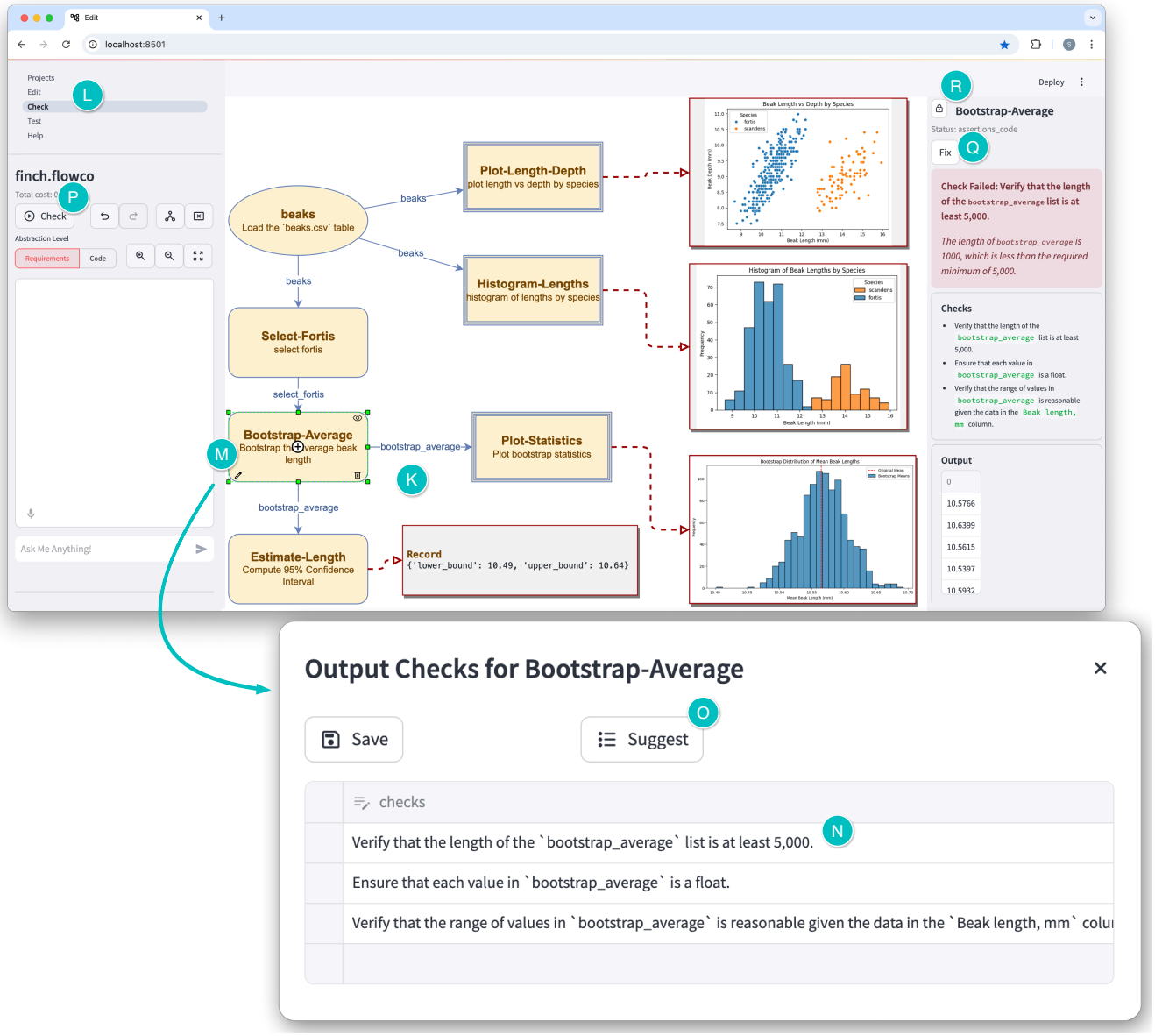


Figure 5: Flowco enables the user to validate run-time assertion checks on node outputs via the Checks view. After K adding nodes to estimate the mean beak length for the Fortis finches L the user switches to the Checks view and M clicks the pencil icon that appears while hovering over Bootstrap-Average to bring up the dialog box shown below the screenshot. N The user manually adds the check "Verify that the length of the `bootstrap_average` list is at least 5,000", and then O clicks the Suggest button to have Flowco suggest several additional checks for that node. After saving the checks the dialog closes and P the user then clicks the Check button to verify that all checks pass. Q Flowco reports a failure for the Bootstrap-Average node.

Bootstrap-Average to create an array of resampled means and then adds **Estimate-Length** to compute the 95% confidence interval for the mean beak length. Alex also adds the node **Plot-Statistics** to plot the resampled means for visual inspection. After adding those nodes, Alex runs the graph to compute a 95% confidence interval of approximately 10.49 – 10.64.

Upon seeing the results, Alex decides to make several small edits to the dataflow graph to improve the presentation. Via AMA chat, Alex adds an additional requirement to **Plot-Statistics** that the

original mean should be shown on the plot and saves the change[4]. Outputs from all predecessors of a node are available, so **Plot-Statistics** is able to compute the original mean directly from the **beaks** output. Alex also asks that the result of **Estimate-Length** be rounded to two decimal places. The final graph is shown in Figure 5.

Checks and Unit Tests. To gain confidence in the computation performed by Flowco and the results, Alex then adds assertions to check the node outputs in the graph. Specifically, (L) Alex switches from the Edit view to the Checks view at the top of the Project panel and then (M) clicks the edit pencil icon that appears while hovering over **Bootstrap-Average**. The dialog box shown in Figure 5 appears. Alex (N) adds one check manually: "Verify that the length of the `bootstrap_average` list is at least 5,000", and then (O) clicks the Suggest button. Flowco suggests several additional checks for that node. Alex saves those checks and then repeats the process for the other nodes in the graph.

(P) Alex then clicks the Check button. Flowco synthesizes code to run each check and verifies that they pass. In this case, (Q) Flowco reports a failure for the **Bootstrap-Average** node. indicating that only 1,000 resamples were used by the bootstrap process.

That number of resamples was not specified by Alex in the requirements, and the choices made during code generation were inconsistent with the expectations Alex made explicit in the checks. Upon seeing that check failure, Alex clicks the Fix button, leading Flowco to adjust the node's requirements and code to address the issue. Flowco then re-runs that node and all downstream nodes, and all checks subsequently pass.

Alex then selects the Test view from the Project panel and adds a collection of unit tests to each node to gain greater confidence that the nodes are robust to changes in their inputs, due to either changes in the data or preceding nodes. Alex chooses to add some by hand and asks Flowco to suggest others. For example, Flowco suggests the following tests for **Select-Fortis**:

- Test with a DataFrame containing multiple species, including 'fortis'.
- Test with a DataFrame containing no 'fortis' species.
- Test with an empty DataFrame.
- Test with a DataFrame where all rows are 'fortis'.
- Test with a DataFrame containing NaN values in the species column.

Alex runs a testing pass, during which Flowco generates code to produce appropriate inputs for each test, calls the node's code on those inputs, and verifies the outputs are as expected.

For **Bootstrap-Average** and **Select-Fortis**, all checks and tests are quantitative in nature. Flowco also supports qualitative checks for nodes that produce visual outputs, such as a check that "the plot uses a color overlay to distinguish between the two species" for **Histogram-Lengths**. Flowco utilizes the LLM to visually inspect the output plot and determined whether the check passes or fails.

[4]Such changes can also be made directly rather than through chat, as illustrated in section 5.4.

4 Programming Model and System Architecture

Flowco enables data scientists to effectively formulate, implement, and reason about data science computations while leveraging LLMs throughout the process. Reliability and correctness are paramount in this context and necessitate the adoption of a programming model and system whose design places sufficient guardrails on the LLM to achieve high confidence in the correctness of the results. This section first presents the Flowco programming model, and then how it utilizes LLMs while ensuring reliability.

4.1 Key Dataflow Graph Properties

Flowco's dataflow-based programming model enjoys several key properties that make it a natural fit both for authoring data science workflows and for interacting with LLMs.

Modularity and Composability. Users encapsulate each analysis step as a distinct node in the dataflow graph, and users add explicit edges to communicate information between steps. This approach allows complex computations to be structured as manageable, readily understood, and composable components.

Abstraction and Refinement. Users can specify and reason about the behavior of a node at different levels of abstraction, from brief summaries to detailed requirements to actual code.

- The summary label, shown in the canvas, gives a high-level description to aid in organizing and understanding the dataflow graph.
- The requirements refine that summary to precisely capture the node's behavior.
- The code refines those requirements to an executable form.

The user may choose which level of abstraction to work at for any particular task and may even choose to fully ignore the lower layers of abstraction.

Explicit Dependencies. Nodes within a dataflow graph are stateless and free of externally visible side effects, meaning that each node's output depends solely on its input and that the edges explicitly capture all dependencies between nodes. Thus, evaluation order is derived directly from the graph structure: a node executes only after its input nodes complete. Moreover, a node need only recompute its output when its inputs or implementation changes.

4.2 System Architecture

Flowco is deployed as a web service. It is written in Python and uses the Streamlit [43] framework to provide a web interface. Flowco can use a variety of LLMs, but the default is OpenAI's GPT-4o model [34], which has proven effective across the variety of tasks that Flowco performs. Further, that model's API supports multi-modal inputs, tool calls, streaming, and structured output, all of which Flowco uses. Flowco's primary internal data structure is the dataflow graph. The user interface is built around views of and modifications to this graph, as are Flowco's collection of LLM agents and mechanisms for ensuring reliability. The most salient aspects of those features and agents are described in the next section.

5 Ensuring Reliability

This section outlines Flowco's mechanisms and guardrails for ensuring reliability.

5.1 Code Synthesis and Evaluation

Flowco adopts modular code generation to constrain the scope of any individual LLM operation to small, well-defined, and isolated synthesis tasks. In particular, each LLM synthesis step is restricted to a single node in the dataflow graph, and to a single abstraction layer within that node. This modular approach ensures that the LLM's output is limited in scope, focused and more easily understood than larger, more expansive code generation tasks.

Flowco's code synthesis and evaluation process for a node is divided into three distinct steps:

- **Requirements Step.** The inputs to this step are the dataflow graph both in text-based JSON and as an image[5], as well as the requirements for all predecessor nodes, which serve as the preconditions for incoming values. Nodes are processed topologically to ensure the availability of the predecessor requirements but can otherwise be processed in parallel. Flowco first uses the LLM to check the consistency of the preconditions. Flowco then generates a precise description of the node's behavior, in the form of a prose bullet list, and also the node's output type, in the form of an extended Python type. These extended types capture additional information beyond what is present in regular types. For example, they include the element type and meaning for a list, and the column names, types, and descriptions for a dataframe.
- **Code Step.** The inputs to this step are a node's preconditions, requirements, and output type. Flowco uses the LLM to refine the requirements into an executable Python function whose parameter types match the incoming data and whose return type matches the output. As illustrated in Figure 6, Flowco provides a template for this function, as well as a pydoc documentation string including descriptions of the parameters, return value, preconditions, and requirements. Code generation for a node may proceed as soon as that node's requirements are available.
- **Evaluation Step.** The inputs to this node are the node's code and the input values. As before, evaluation proceeds in topological order and exploits parallel evaluation whenever possible. Flowco employs a shared pool of Python kernels for this step.

To build and run a graph, Flowco schedules agents to perform each individual synthesis step for each node, respecting the dependencies between nodes and exploiting parallelism wherever possible.

5.2 Error Detection and Repair.

It is, of course, possible that the LLM fails to produce correct code. To mitigate synthesis failures, Flowco employs three low-level repair mechanisms during evaluation. First, Flowco checks that a node's code is syntactically well formed and asks the LLM to correct any syntax errors. It provides the LLM with the details of the error and the node's requirements, output type, and code to help the LLM understand the context of the error. Second, Flowco catches all run-time errors and again asks the LLM to correct them. It provides the LLM with details about both the node and the error in this case. Finally, Flowco validates the output value against the synthesized extended output type to ensure that the output meets the node's requirements, again asking the LLM to repair the node if the output does not match the expected type.

In all cases, Flowco first attempts a local repair to the offending node. If the error persists after three repair attempts, Flowco gives the user the option to attempt a global repair via an LLM agent that has access to the entire dataflow graph. The LLM is directed to provide a minimal set of changes that fixes the problem.

[5]Evidence suggests that leveraging redundancy between vision and text can enhance model performance [47].

```python
# put all imports here
def select_fortis(beaks: pd.DataFrame) -> pd.DataFrame:
  """
  select fortis

  This function has the following behavior:
  - The the result output is a Pandas DataFrame containing a subset
    of rows from the `beaks` DataFrame where the species is 'fortis'.
  - The the result DataFrame retains the same columns and column
    names as the `beaks` DataFrame.

  Args:
    beaks (DataFrame[
     'species': str  # The species name as a string.
     'Beak length, mm': float  # The length of the beak in millimeters.
     'Beak depth, mm': float  # The depth of the beak in millimeters.
    ]): The structure of the dataset representing bird species and
        their beak dimensions.

      Preconditions:
      - beaks is the dataframe for the `beaks` dataset.

  Returns:
    DataFrame[
     'species': str  # The species of the bird.
     'Beak length, mm': float  # The length of the beak in millimeters.
     'Beak depth, mm': float  # The depth of the beak in millimeters.
    ]: The output is a filtered DataFrame containing only rows where
       the species is 'fortis', retaining all original columns and
       column names.
  """
  # put code here
  ...
```

Figure 6: When Flowco asks the LLM to generate code for Select-Fortis during the compile step outlined in section 5.1, it provides a template containing the function signature and a pydoc string describing the parameters, preconditions, and requirements.

5.3 Observability and Understanding

Flowco's dataflow graphs, abstraction layers, and modular synthesis strategy provide a clear and structured way to observe and understand each individual step of the synthesis process in isolation. In particular, each individual LLM operation is restricted to modifying a specific part of a node, eliminating the possibility that any other parts of that node or other nodes are inadvertently changed.

The dataflow graph also provides sufficient structure to enable the LLM to readily aid the user in understanding the computation and outputs. This is best demonstrated by the AMA chat agent, which serves as the primary direct interface between the user and the LLM. The AMA chat agent equips the LLM with several capabilities to enable it to handle a wide variety of user queries and requests. First, the LLM can inspect the entire dataflow graph and the contents of each node, including output values and plots. Second, the LLM can run code snippets in the context of the graph,

allowing it to evaluate expressions, run tests, and perform analyses that reference any part of the graph. Finally, the LLM can even change the dataflow graph directly by adding, removing, or modifying nodes and edges. The agent employs the tool call mechanism provided by the LLM [33] to implement these capabilities, and requests that the LLM provide an explanation of its reasoning and rationale when performing any of these actions.

5.4 Robust Updates

Users may iteratively refine or change the dataflow graph either via direct edits in the canvas or via AMA chat. Such changes may (i) modify the graph structure by adding or removing nodes or the edges between them; or (ii) modify a node at any level of abstraction: description, requirements, or code. The former necessitates examining and possibly modifying downstream nodes to accommodate the change. The latter additionally necessitates examining and ensuring consistency among the abstraction layers within the modified node. Flowco handles these cases through a variety of techniques for change propagation both within and between nodes.

User edits. The user may directly edit a node's components by clicking on the pencil icon that appears while hovering over a node while in the edit view[6]. Returning to our usage scenario in Section 3, suppose Alex wishes modify the **Plot-Statistics** node to show the original sample's mean in the generated histogram. Alex clicks on the pencil icon for that node to bring up the dialog box shown in Figure 7[7]. (S) Alex then adds "The plot includes the original sample mean" to the requirements, at which point (T) a warning appears indicating that the node's summary label and code may not be consistent with the new requirements. (U) Alex then clicks the requirement's change propagation button, which updates the summary label and the code to reflect the new requirements. Once the node is returned to a consistent state, the Save button is enabled, and Alex can save the changes and return to the graph editor.

The node editor provides several other features. The Check Consistency button uses the LLM to check the node's title, label, requirements, and code for consistency, reporting any warnings for the user to address. The Regenerate button uses the LLM to regenerate the node's components using their current values as a guide and choosing how to resolve any inconsistencies. Both are useful in practice, as demonstrated in the user study presented in Section 7. A node-specific AMA agent is also available to explain the node's behavior and make changes conversationally. Chat is often the most expedient way to make small changes, as the AMA agent ensures consistency among the constituent parts as it makes modifications.

Change propagation. Flowco leverages the explicit dependencies captured by the dataflow graph structure to propagate changes through the graph during its build process. Specifically, when a node's incoming/outgoing edges or requirements are modified, Flowco invalidates all downstream nodes and resynthesizes their requirements and code under the upstream change. Flowco instructs the LLM to make the smallest set of modifications necessary to address incoming changes to each node. The user may lock nodes they do not wish Flowco to modify while propagating changes via the lock toggle button in the details panel (marked as (R) in Figure 5). When a node is locked, the LLM checks the node for consistency with upstream modifications but does not make any changes to it.

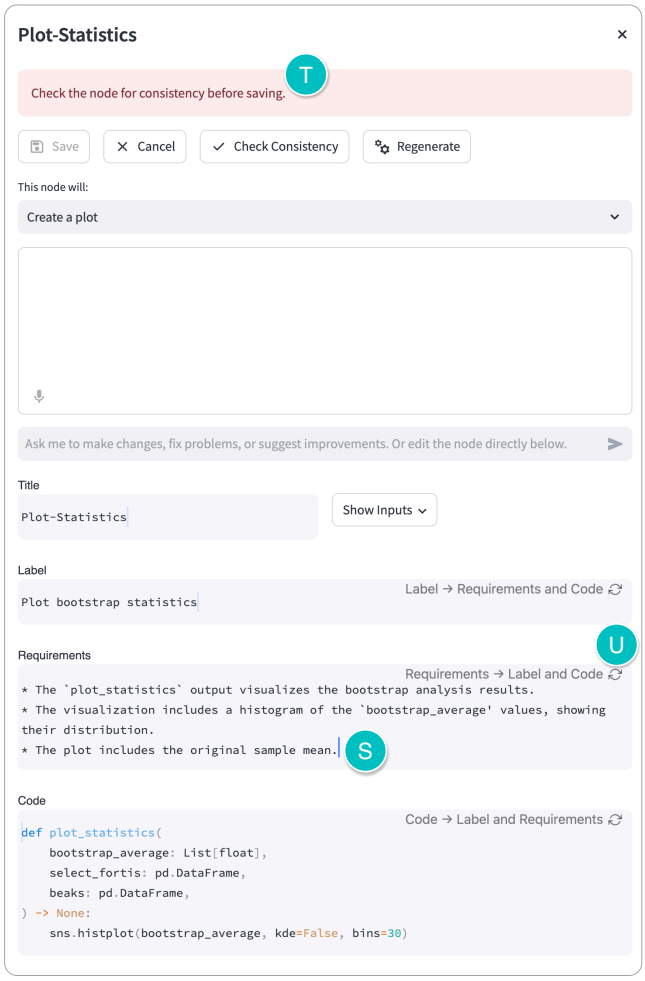


Figure 7: The user directly modifies the components of a node via an editor dialog box. The dialog box allows the user to directly edit the node's title, summary label, requirements, and code. The user may also propagate changes in one component to others. For example, (S) Alex adds a requirement, which (T) brings up a warning that the node's different components may not be consistent. (U) The user then clicks the propagation button to update the summary label and code. The editor also supports making modifications via chat, checking consistency between the components, and regenerating the components from scratch.

[6]This is similar to editing the checks for a node while in the checks view, as shown in (M) of Figure 5.

[7]The code may be hidden if the user is working at the "Requirements" level of abstraction.

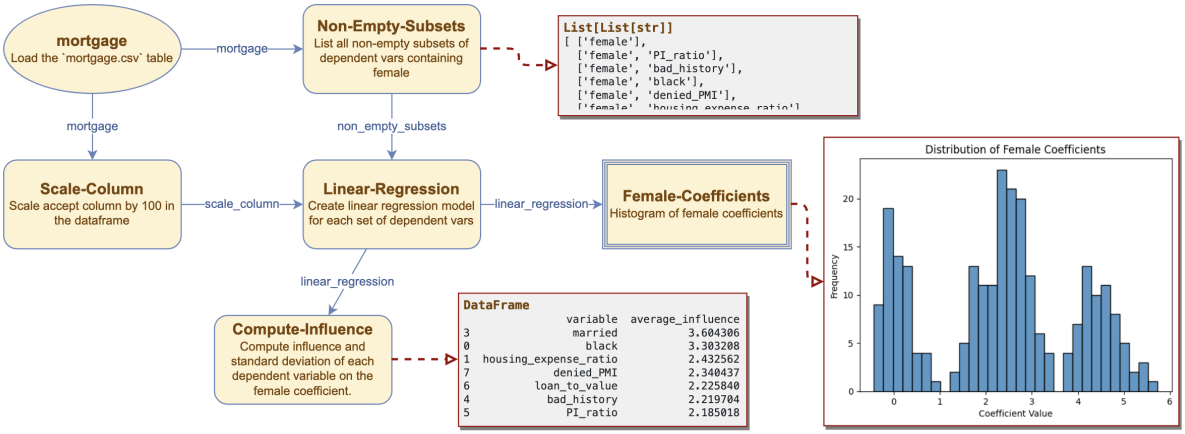


Figure 8: A multiverse analysis presented in 6.2 that explores how a female applicant's likelihood of mortgage approval varies depending on different combinations of control variables [29].

5.5 Validation: Assertions and Unit Tests

To further increase confidence in the correctness of the computation, Flowco provides several user-facing guardrails that allow validation of node behavior and detection of errors introduced by either the user or the LLM.

Assertion Checks. First, Flowco supports run-time assertions, which are simple checks on a node's output that serve as safeguards to catch unexpected behaviors, incorrect assumptions, or violations of key invariants. Assertions are particularly helpful when dealing with evolving data inputs, subtle bugs in synthesized code, or misunderstandings in analysis logic. The user can write assertions manually or have the LLM suggest them based on the node's requirements and code.

As illustrated in Section 3, Flowco has a Checks view separate from the main Edit view for inspecting, modifying, and checking the assertions associated with each node. This design choice is intended to make it easy to develop and edit dataflow graphs without assertions getting in the way until they are desired. Checks are expressed as a list of prose statements about the output that may also refer to the node's inputs. For data loading and computational nodes, the assertions are most often quantitative statements, such as "the output mean is greater than 0." Flowco translates quantitative assertions into Python code that performs the appropriate test. Nodes producing plots may also have qualitative assertions, such as "The histogram is bimodal." For those, Flowco leverages vision capabilities of the LLM to validate whether the assertion holds.

Unit Tests. Second, Flowco lets users attach unit tests to individual nodes through a Tests view. Like assertions, these tests are expressed as prose and may be written by the user or suggested by the LLM. Each test defines a specific input scenario and the expected behavior of the node under that scenario. These tests are particularly useful for verifying that a node handles edge cases correctly, produces meaningful results on representative inputs, and remains stable in the face of changes upstream. For each test, Flowco generates Python code to create appropriate input values, run the node's code, and check the output against the expected behavior. As with assertions, Flowco supports both quantitative and qualitative unit tests.

Error Reporting and Repairs. Together, assertions and unit tests help detect errors that arise during analysis construction or evolve over time. When failures occur, Flowco reports them to the user and provides a Fix button to automatically address the problems. Flowco uses the same repair mechanisms described earlier in section 5.2. It first attempts localized fixes within the impacted node and then escalates to an optional global repair strategy if necessary.

6 Example Analyses Using Flowco

We demonstrate the Flowco's ability to express real-world data analyses ranging from what may be found in introductory data science classes to a complex multiverse analysis and a logistic regression with cross validation.

6.1 Clustering and Linear Regression

Figure 1 illustrates how an analyst might use Flowco to explore the Old Faithful geyser dataset. The dataset contains two columns: eruptions (duration of eruption in minutes) and waiting (waiting time between eruptions in minutes). After loading the data, the analyst creates a scatter plot to visualize the relationship between eruption duration and waiting time (**Duration-vs-Wait**). The analyst then performs k-means clustering to partition the data into two clusters based on the eruption duration and waiting time (**K-Means-Clusters**), fits a linear regression model to each cluster (**Linear-Regression**), and visualizes the clustered data with regression lines (**Plot-Regressions**). The analyst works entirely within the visual editor to create the graph shown in the figure. After viewing the original output, the analyst makes AMA chat requests to: (1) name the clusters "short" and "long" and (2) adjust the colors of the final plot.

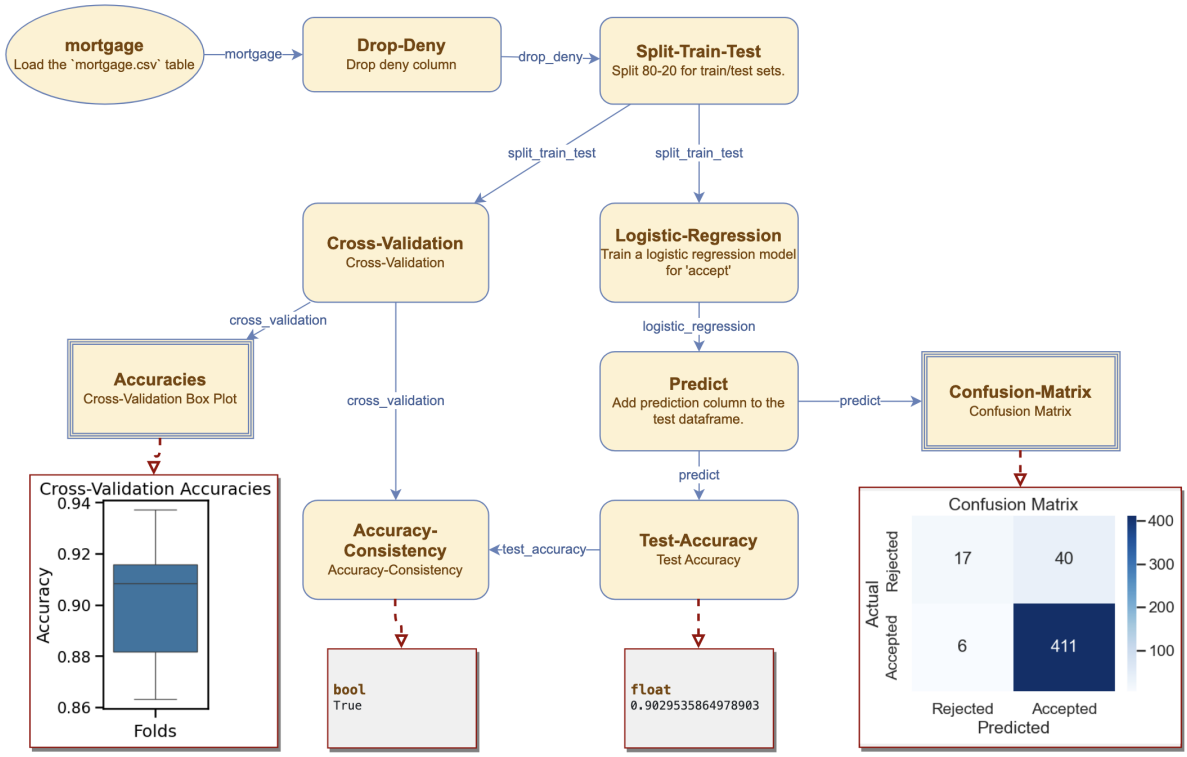


Figure 9: A logistic regression presented in section 6.3 that predicts whether a mortgage application will be accepted. The dataflow graph also includes accuracy measurement and cross validation.

6.2 Multiverse Analysis

Figure 8 presents an analysis demonstrating Flowco's capacity to support sophisticated analyses. The analysis repeats a previously published multiverse analysis on how a female applicant's likelihood of mortgage approval varies depending on different combinations of control variables [29]. The analyst designs a dataflow graph to captures the key steps of that analysis, including enumerating all combinations of control variables (**Non-Empty-Subsets**), fitting a linear regression model for each combination (**Linear-Regression**), and plotting a histogram capturing the variability of the estimated effect of being female on mortgage acceptance rates and the sensitivity of that effect to the choice of control variables (**Female-Coefficients**). As in the original study, the additional influence analysis concludes that the married and black control variables have the greatest influence on the coefficient for female (**Compute-Influence**). Flowco enables the author to explore the multiverse of possible models in a structured and systematic way without writing a single line of code.

6.3 Confirmatory Analysis: Logistic Regression

The authors of the earlier multiverse analysis [29] note that logistic regression is a better fit than linear regression for this problem because of the binary outcome of accepting or rejecting a mortgage application. In Figure 9, the analysis applies logistic regression to the mortgage data. After removing the "deny" column (**Drop-Deny**), the analyst splits the data into training and testing sets (**Split-Train-Test**) and performs logistic regression on all control variables (**Logistic-Regression**). The analyst applies that model to the testing dataset (**Predict**) and determines the accuracy to be 0.903 (**Test-Accuracy**). The analyst also makes a confusion matrix for the testing dataset.

The analyst then performs a 10-fold cross validation on the training data to construct a robust estimation of the model's expected performance on unseen data (**Cross-Validation**). The box plot below the **Accuracies** node shows a median accuracy between 0.90 and 0.91, with little variation, demonstrating the logistic model consistently performs well across different subsets of the data. The median is close to the accuracy reported in the **Test-Accuracy** node, as expected. The analyst adds the final **Accuracy-Consistency** node to make that expectation explicit.

7 User Evaluation

While the examples demonstrate Flowco's ability to author real-world analyses, we wanted to assess how Flowco could support users relatively new to data analysis. Three research questions motivated our user evaluation of Flowco:

- **RQ1 - Authoring success.** Can participants use Flowco to successfully author analyses?

		Experience			Node Creation in Flowco	
PID	Major	Stats	Programming	Python Comfort	Direct	AMA
P1	Data Science	8	7	Intermediate	0	5
P2	Data Theory	7	7	Intermediate	2	5
P3	Data Science	3	5	Intermediate	1	7
P4	Data Science	6	6	Intermediate	2	9
P5	Computer Science	4	9	Fluent	5	0
P6	Data Science	7	4	Not at all	5	3
P7	Data Theory	9	6	Fluent	5	0
P8	Data Science	6	6	Beginner	5	1
P9	Data Science	9	8	Very Comfortable	7	0
P10	Data Science	5	4	Intermediate	2	5
P11	Data Science/Software	8	7	Very Comfortable	6	0
P12	Data Science	4	3	Not at all	1	4

Table 1: Participant background and experience. All participants are students. Statistics and Programming Experiences are self-reported on a scale from 1 (very inexperienced) to 10 (very experienced). The "Python Comfort" column is the self-reported comfort with programming in Python on a scale of "Not at all" to "Extremely." The last two columns count the number of nodes participants created via direct editing on the canvas (Direct) or via AMA chat (AMA) during the user study (Section 7).

- **RQ2 - Reactions to programming model.** What are participants' reactions to Flowco's dataflow programming model?
- **RQ3 - User experience compared to other tools.** How does the experience of using Flowco compare to experiences with LLMs and other tools (e.g., scripting, computational notebooks)?

To answer these questions, we conducted a first-use study with 12 participants recruited through mailing lists, Slack groups, and professional networks related to data science. Participants self-reported their experience conducting statistical analyses, using LLMs for data analysis, and programming in computational notebooks. The study participants had a wide range of analysis and domain backgrounds (see Table 1 for a summary). On a ten-point scale, participants ranged from 3 to 9 (mean: 6.3) for statistics experience and from 3 to 9 (mean: 6) for programming experience, with 1 as "very inexperienced" and 10 as "very experienced." In addition, participants ranged from "Not at all" fluent in Python to "Fluent," with the majority self-reporting as "Intermediate," "Very Comfortable," or "Fluent." Of the twelve participants eleven are students in a Data or Statistics field and one is a student in Computer Science.

7.1 Study Procedure

After giving consent, the participants watched a 12-minute tutorial video on Flowco. Then, the researcher presented participants with a dataset of airlines and their safety [11] and asked the participants to conduct an exploratory data analysis for seven minutes. After the initial exploration phase, the researcher asked participants to spend another seven minutes answering the following question: "To what extent can we say that fatal accidents between 1985 and 1999 predict fatality rates between 2000 and 2014?" Participants were asked to think aloud while using Flowco. The seven-minute time frames were based on pilot studies in which participants were able to use Flowco to analyze datasets of similar complexity within seven minutes. When Flowco was not acting as expected, for example due to bugs or LLM response latency, the researcher assisted the participants in resolving the issue while remaining as neutral as possible. When participants asked questions about how to use the system, the researcher provided limited guidance to ensure all participants had as equal an experience as possible.

Once participants arrived at a conclusion and were satisfied with their analysis, the researcher engaged them in a semi-structured interview about their experiences with Flowco and how the system compares to other ways in which participants have used LLMs and computational notebooks for data analysis. Each study lasted approximately one hour, and participants were compensated $25 for their time. All studies were conducted on Zoom. Video and audio were recorded for analysis.

7.2 Analysis and Findings

We thematically analyzed researcher notes and audio transcriptions to answer our motivating research questions.

7.2.1 Graph creation. During the unguided analysis, participants created dataflow graphs containing between five and ten nodes (median: 6.5). Four out of the twelve participants began by creating a summary statistics node, such as calculating mean values across columns or asking the LLM to generate "summary statistics" (P1, P9, P10, P12). Seven of the participants created nodes via both (i) direct editing of the graph on the canvas and (ii) AMA chat. However, they primarily relied on one approach or the other. Four used direct editing exclusively, and one used AMA exclusively.

7.2.2 Authoring with ease. After the tutorial, all participants reported feeling able to use Flowco proficiently. Participants mentioned that they were able to gain this proficiency quickly. For example, P12, who had no experience programming with Python, expressed, "I think it's really a good sign that I was able to learn something like this in seven minutes or less because I'm kind of

a slow learner and it ended up being really helpful." Participants continued to gain confidence as they progressed through the exploratory analysis (P1, P2, P3, P6). Despite some minor challenges and bugs, all participants reported that Flowco's outputs matched their desired outcomes.

Participants mentioned some specific tasks in Flowco that posed a challenge to them, but only P4 and P6 encountered problems they could not resolve. Both were related to code generation errors that Flowco could not repair. These point to the need for more robust repair mechanisms [6, 28, 51] in the future. Some participants, such as P8, explained, ". . . even if I did struggle [editing the canvas] I could have used the 'Ask Me Anything!' part to assist me."

7.2.3 Benefits of the dataflow graph programming model. Most participants found the graph-based model beneficial. Several participants mentioned that the graph format organized the analysis in an easier manner to interpret compared to other tools they have used (P3–4, P6–8, P12). P3 reflected, "I think this flow is super helpful to keep track of what you've coded and what you've done so far." Participants also mentioned how Flowco helped them understand the process of data analysis (P4), see how analysis steps inform each other (P8), and keep track of and work with multiple datasets (P7).

Participants specifically stated that Flowco's model made it easier to manage data and computation than computational notebooks (P3, P11). P3 noted that "when you're working with pandas it just can get a little messy . . . Between the cleaning and transforming data, and then creating plots, creating models, it can be easy to get kind of lost or forget where everything came from. So I think even here being able to see just the lines of how we got from the original dataset is super helpful. . ."

7.2.4 Benefits of Flowco's deep LLM integration. Participants with experience using LLMs for data analysis said that Flowco's integrated LLM worked better for them than using other LLM tools (P1, P3–P7, P9, P11, P12). Participants mentioned that they trusted Flowco's code more than code obtained directly from an LLM, such as ChatGPT (P3, P5, P7). When asked why, P5 explained, "Knowing that there is a layer of prompting that was carefully engineered, and that outputs are being checked for type matching and such, I trust this a lot more than just asking ChatGPT." Participants also liked not having to toggle between ChatGPT and their tool, meaning that they did not have to copy and paste between the LLM and a code editor (P1, P4, P6, P9, P11, P12). Other benefits of this integration that participants noted were the speed with which they could analyze data and the decreased effort required. P3 described, "I would definitely consider using this over ChatGPT . . . I'll have to put [in] less effort. I can just put my dataset here and then I can visualize multiple things together."

7.2.5 Benefits for non-experts and data exploration. Participants speculated that Flowco would be (i) particularly beneficial for beginners and (ii) best for introductory or exploratory data analysis (P3, P4, P6, P9, P11). P6 noted it would be useful in these contexts "because it's more visual and easier to understand." In addition, seven participants expressed interest in continuing to use Flowco for their work (P1–P3, P5, P7, P8, P12). Five described a desire to use Flowco over their current tools, particularly for preliminary exploratory analysis or "quick scratch work" (P2, P5, P3, P8, P12). P2 noted that they would use Flowco to "explore ideas that I have that I maybe I don't have time to code. . ." Additionally, participants found that Flowco's simplicity allowed them to write their analysis with greater ease than in other tools. P7 described, "I think it's definitely less effort on the part of the person who's doing the data analysis because you can give commands to an LLM and basically have it do everything for you. . ."

Some participants expressed concern that Flowco would not be able to give correct code for more complex problems (P2, P5, P6, P9, P12). This appeared to be a concern based mostly on their prior experiences using LLMs because, with the exception of P6, these participants did not encounter issues with Flowco's code generation. Several participants also expressed concern that graphs with many nodes could become "messy" and confusing quickly (P3, P11). P3 explained, "I would say based on the way I tend to do exploratory data analysis or just exploring a dataset the whole flow concept here would get very messy."

7.2.6 LLM latency. Participants successfully authored analyses quickly using Flowco (RQ1), although the underlying communications with LLMs introduced noticeable latency. Seven out of twelve participants noted this as a challenge. For instance, P5 mentioned some confusion arising from latency, noting, "at times I didn't know if I was waiting on the system or if the input wasn't even picked up." However, only two participants, P7 and P10, indicated that latency would likely deter them from using Flowco. For others, the benefits of Flowco outweighed the frustrations caused by latency.

Advances in LLM performance [46] will make latency less of an issue over time. In addition, Flowco can be extended to include various well-known techniques to mitigate latency, including more broadly using asynchronous processing for LLM requests while the user is working on other tasks, prefetching LLM responses that are likely to be needed in the background, and employing lower-latency models when feasible.

7.2.7 Usability issues and fixes. The study revealed three specific usability issues with Flowco. First, some participants wished for an easier way to view the dataset, results, and code (P1, P6, P7, P10), as the right-hand panel showing that information was not always noticeable or easily navigable. Several Flowco features have been refined to make that information more accessible, including more extensive use of layovers. Second, Flowco's programming model originally restricted users from connecting a node producing a plot to another node because plotting nodes do not produce output values processable by downstream nodes. This behavior was not clear to some participants (P2, P6) and caused confusion when they wished to structure their graphs in that way. Flowco now allows plot nodes to have child nodes to match this usage pattern, and in keeping with the Flowco dataflow model, those nodes have access to the outputs of all ancestors in the graph. Third, several participants (P5, P9) encountered an interface refresh bug that caused confusion over whether Flowco was waiting for the LLM to respond or for the user to provide input. That has been fixed in the current version.

8 Discussion, Limitations, and Future Work

The findings from the user study support the following answers to the research questions posed in Section 7.

RQ1: Overall, participants found Flowco easy to learn and beneficial for preliminary analysis. All participants were able to successfully author analyses independently after watching the tutorial, and the visual programming model enabled participants to clearly organize their computation as a graph. Even the least experienced programmer was able to complete the analysis presented to them. The study also found that Flowco could help more experienced participants quickly "sketch" analyses. In other words, it seems that Flowco not only lowers the barriers to analyses for novices but also could equip expert users with new capabilities.

RQ2: Flowco's deep LLM integration increased both ease of use for participants as well as their confidence in the results.

RQ3: After the study, eight of the twelve participants said they would consider using Flowco in the future. Two participants preferred Flowco to computational notebooks, and almost all participants preferred Flowco to using LLMs directly.

These results suggest that Flowco's mixed-initiative approach (i) strikes a balance between user control and LLM generation and (ii) provides useful abstractions and interactions for users to inspect and validate generated code. In this way, Flowco is one example of how to realize some of Horvitz's principles for mixed-initiative systems [17] with LLMs. The remainder of this section outlines a number of promising directions for future work.

Evaluation with real-world data scientists. One limitation of the user study is its focus only on data science students. While non-experts are one of the primary intended audiences for Flowco, that focus does limit what conclusions can be drawn from the study. Future work will build on these findings by evaluating Flowco across real-world data scientists with a broader range of expertise and skills. Of particular interest is how Flowco performs in the context of different domains or types of data analysis tasks, such as hypothesis testing, modeling, or causal inference.

Statistical validity. Flowco's architecture prioritizes the reliable generation of executable code. In data science, it is all too easy for analysts to inadvertently employ statistical techniques or models that are inapplicable to their data or research question. Such mistakes can lead to invalid conclusions without warning. In fact, a previous study found that even expert tutorials selected statistical tests that were inapplicable based on the explicitly stated assumptions about data properties [21]. A critical line of future work is extending Flowco's guardrails to check statistical correctness [21, 22]. The deep integration of LLMs into Flowco is a benefit here since LLMs can leverage this additional guidance (e.g., as checks) to generate statistically valid analysis code.

Hierarchical graphs. Participants voiced concern about how "messy" dataflow graphs could become with more complex analyses. A key challenge of visual programming environments is managing the complexity of large computations, especially when restricted to a single view [20, 44]. Techniques from hierarchical graph visualization and interaction are an interesting avenue to explore for Flowco. This approach may involve, for example, supporting collapsible subgraphs, providing views that present different parts of the graph at different levels of detail, or changing the programming model to explicitly support hierarchical decomposition of graphs into subgraphs that can be viewed and manipulated independently.

Sophisticated workflow design. The user study demonstrates the key benefits of employing a dataflow programming model for data science tasks. However, Flowco presently requires analysts to define graphs at the level of individual nodes, which can be labor intensive for sophisticated problems. An interesting avenue for future work is exploring how Flowco can support decomposition of large analysis tasks into manageable graphs more automatically. This may involve providing the user with templates capturing common analysis idioms, as well as more advanced AMA chat mechanisms that support the conversational creation of whole dataflow graphs.

Integration with computational notebooks. Finally, some study participants expressed to a desire transition between Flowco and computational notebooks. While Flowco supports exporting analyses as Jupyter notebooks, tighter integration with notebooks while still retaining the key benefits of Flowco remains for future work. Deployment studies tracking when analysts switch modalities will inform this line of work.

9 Conclusion

Flowco bridges the gap between the flexibility of code and the approachability of visual interfaces, enabling users——especially those with limited programming experience——to more effectively author, debug, and iterate on data analysis workflows. By integrating LLM assistance throughout the development process, Flowco supports a more interactive and exploratory approach to data science that offers a number of clear advantages over existing authoring tools, including computational notebooks and existing LLM-based tools. Flowco additionally emphasizes correctness and transparency, providing users with the tools to understand and validate the code generated by LLMs and the behavior of the system.

10 Availability

Source code and additional information is available at https://github.com/stephenfreund/flowco.

11 Summary of Author Contributions

Stephen Freund created/contributed to the key ideas behind the work and implemented the Flowco prototype and collaborated on writing the paper. Brooke Simon actively tested and gave feedback on the interface, contributed to the user study's design, conducted the user studies, and collaborated on data analysis and writing the paper. Emery Berger helped develop the core ideas and guided the design of Flowco through numerous discussions and provided feedback on the paper. Eunice Jun helped develop the core ideas and guide the design of Flowco, contributed to the user study's design and data analysis, and collaborated on writing the paper.

Acknowledgments

We thank Kyla Levin for valuable suggestions and feedback. We thank our participants for their time and feedback. We thank members of the UCLA Computation & Discovery Lab for testing Flowco, trying out tutorials, and giving feedback on drafts of this paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. USENIX Association, 265–283.

[2] Altair. 2025. Altair RapidMiner. https://altair.com/altair-rapidminer. Accessed: 2025-03-16.

[3] Alteryx. 2025. Alteryx Designer Cloud. https://www.alteryx.com/products/designer-cloud. Accessed: 2025-03-16.

[4] Anaconda. 2023. Anaconda Assistant Launches to Bring Instant Data Analysis, Code Generation, and Insights to Users. https://www.anaconda.com/blog/anaconda-assistant-launches-to-bring-instant-data-analysis-code-generation-and-insights-to-users Accessed: 2024-10-19.

[5] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena L Glassman. 2024. Chainforge: A visual toolkit for prompt engineering and llm hypothesis testing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–18.

[6] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *CoRR* abs/2403.17134 (2024). https://doi.org/10.48550/ARXIV.2403.17134 arXiv:2403.17134

[7] 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 *CHI*. ACM, 1–12.

[8] Bhavya Chopra, Ananya Singha, Anna Fariha, Sumit Gulwani, Chris Parnin, Ashish Tiwari, and Austin Z Henley. 2023. Conversational challenges in ai-powered data science: Obstacles, needs, and design opportunities. *arXiv preprint arXiv:2310.16164* (2023).

[9] Databricks. 2023. Introducing Databricks Assistant, a Context-Aware AI Assistant. https://www.databricks.com/blog/introducing-databricks-assistant Accessed: 2024-10-19.

[10] Helen Dong, Shurui Zhou, Jin LC Guo, and Christian Kästner. 2021. Splitting, renaming, removing: a study of common cleaning activities in Jupyter notebooks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 114–119.

[11] FiveThirtyEight. 2025. Airline Safety. https://github.com/fivethirtyeight/data/blob/4c1ff5e3aef1816ae04af63218015066e186c147/airline-safety/airline-safety.csv. Accessed: 2025-03-16.

[12] Peter R. Grant and B. Rosemary Grant. 2013. Data from: 40 Years of evolution: Darwin's finches on Daphne Major Island. https://datadryad.org/stash/dataset/doi:10.5061/dryad.g6g3h. Accessed: 2025-03-16.

[13] Peter R. Grant and B. Rosemary Grant. 2014. *40 Years of Evolution: Darwin's Finches on Daphne Major Island*. Princeton University Press.

[14] 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. *Graphics Interface 2023* (2023).

[15] Andrew Head, Fred Hohman, Titus Barik, Steven Mark Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, 270:1–12. https://doi.org/10.1145/3290605.3300500

[16] 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*. 1–12.

[17] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 159–166.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, Paulo Ferreira, Thomas R. Gross, and Luís Veiga (Eds.). ACM, 59–72. https://doi.org/10.1145/1272996.1273005

[19] Peiling Jiang, Jude Rayan, Steven P Dow, and Haijun Xia. 2023. Graphologue: Exploring large language model responses with interactive diagrams. In *Proceedings of the 36th annual ACM symposium on user interface software and technology*. 1–20.

[20] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *ACM Comput. Surv.* 36, 1 (2004), 1–34. https://doi.org/10.1145/1013208.1013209

[21] Eunice Jun, Maureen Daum, Jared Roesch, Sarah E Chasins, Emery D Berger, Rene Just, and Katharina Reinecke. 2019. Tea: A High-level Language and Runtime System for Automating Statistical Analysis. In *Proceedings of the 32nd Annual Symposium on User Interface Software and Technology*. ACM.

[22] Eunice Jun, Audrey Seo, Jeffrey Heer, and René Just. 2022. Tisane: Authoring Statistical Models via Formal Reasoning from Conceptual and Data Relationships. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–16.

[23] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Zachary Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving Steering and Verification in AI-Assisted Data Analysis with Interactive Task Decomposition. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology, UIST 2024, Pittsburgh, PA, USA, October 13-16, 2024*, Lining Yao, Mayank Goel, Alexandra Ion, and Pedro Lopes (Eds.). ACM, 92:1–92:19. https://doi.org/10.1145/3654777.3676345

[24] 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*. 1–11.

[25] Knime. 2025. KNIME Analytics Platform. https://www.knime.com/knime-analytics-platform. Accessed: 2025-03-16.

[26] Jeffrey Kodosky. 2020. LabVIEW. *Proc. ACM Program. Lang.* 4, HOPL (2020), 78:1–78:54.

[27] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*.

[28] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[29] Yang Liu, Alex Kale, Tim Althoff, and Jeffrey Heer. 2021. Boba: Authoring and Visualizing Multiverse Analyses. *IEEE Trans. Vis. Comput. Graph.* 27, 2 (2021), 1753–1763. https://doi.org/10.1109/TVCG.2020.3028985

[30] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[31] Old Faithful Dataset. 2025. https://www.kaggle.com/datasets/janithwanni/old-faithful/data. Accessed: 2025-03-16.

[32] OpenAI. 2024. Improvements to Data Analysis in ChatGPT. https://openai.com/index/improvements-to-data-analysis-in-chatgpt/ Accessed: 2024-10-19.

[33] OpenAI. 2025. OpenAI Function Calling. https://platform.openai.com/docs/guides/function-calling. Accessed: 2025-03-16.

[34] OpenAI. 2025. OpenAI GPT-4o. https://platform.openai.com/docs/models/gpt-4o. Accessed: 2025-03-16.

[35] Evan Patterson, Robert N. McBurney, H. Schmidt, Ioana Baldini, Aleksandra Mojsilovic, and Kush R. Varshney. 2017. Dataflow representation of data analyses: Toward a platform for collaborative data science. *IBM J. Res. Dev.* 61, 6 (2017), 9:1–9:13. https://doi.org/10.1147/JRD.2017.2736278

[36] Project Jupyter. 2024. Jupyter AI Documentation. https://jupyter-ai.readthedocs.io/en/latest/ Accessed: 2024-10-19.

[37] Project Jupyter. 2024. Project Jupyter. https://jupyter.org/ Accessed: 2024-10-19.

[38] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting best practices for collaboration with computational notebooks. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW1 (2022), 1–41.

[39] Eric Rawn and Sarah Chasins. To Appear. Pagebreaks: Multi-Cell Scopes in Computational Notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*.

[40] 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 (2018), 1–12.

[41] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[42] Shreya Shankar, Stephen Macke, Sarah E. Chasins, Andrew Head, and Aditya G. Parameswaran. 2022. Bolt-on, Compact, and Rapid Program Slicing for Notebooks [Scalable Data Science]. *Proc. VLDB Endow.* 15, 13 (2022), 4038–4047. https://doi.org/10.14778/3565838.3565855

[43] Streamlit. 2025. Streamlit. https://streamlit.io/. Accessed: 2025-03-16.

[44] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling multilevel exploration and sensemaking with large language models. In *Proceedings of the 36th annual ACM symposium on user interface software and technology*. 1–18.

[45] Lev Tankelevitch, Viktor Kewenig, Auste Simkute, Ava Elizabeth Scott, Advait Sarkar, Abigail Sellen, and Sean Rintel. 2024. The Metacognitive Demands and

Opportunities of Generative AI. In *CHI*. ACM, 680:1–680:24.

[46] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2024. Efficient Large Language Models: A Survey. *Trans. Mach. Learn. Res.* 2024 (2024). https://openreview.net/forum?id=bsCCJHbO8A

[47] Jiayu Wang, Yifei Ming, Zhenmei Shi, Vibhav Vineet, Xin Wang, Sharon Li, and Neel Joshi. 2024. Is A Picture Worth A Thousand Words? Delving Into Spatial Reasoning for Vision Language Models. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[48] Zuozhi Wang, Yicong Huang, Shengquan Ni, Avinash Kumar, Sadeem Alsudais, Xiaozhen Liu, Xinyuan Lin, Yunyan Ding, and Chen Li. 2024. Texera: A System for Collaborative and Interactive Data Analytics Using Workflows. *Proc. VLDB Endow.* 17, 11 (2024), 3580–3588. https://doi.org/10.14778/3681954.3682022

[49] 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, 1–10.

[50] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*. 1–22.

[51] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[52] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–20.

[53] JD Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Björn Hartmann. 2025. Beyond Code Generation: LLM-supported Exploration of the Program Design Space. *arXiv preprint arXiv:2503.06911* (2025).

A FLOWCO's AMA Chat Response to "Describe this dataset" in Section 3

The following transcript is FLOWCO's response to the AMA prompt "Describe this dataset" in the exploratory steps of the scenario in Section 3.

1. Restating the Question

The question asks for a description of the dataset, which involves understanding its structure, content, and any notable characteristics. Specifically, we want to explore the columns, data types, and any patterns or distributions in the data.

2. Approach

To describe the dataset, I will:

1. Examine the structure of the dataset, including column names and data types.
2. Summarize the data to understand its size and basic statistics.
3. Check for missing values or anomalies.
4. Explore the distribution of key variables, such as `Beak length, mm` and `Beak depth, mm`.

3. Steps of Analysis

I will perform the following steps:

1. **Inspect the structure of the dataset**: This includes column names, data types, and a preview of the data.
2. **Summarize the dataset**: Generate descriptive statistics for numerical columns.
3. **Check for missing values**: Identify if any columns have missing data.
4. **Visualize distributions**: Plot histograms or boxplots for numerical columns to understand their distributions.

Let me start by inspecting the dataset.

Okay, I ran some code:

```
# Inspect the structure of the dataset
beaks.info()
```

The dataset contains 406 rows and 3 columns. Here's the structure:

- **`species`**: A categorical column (object type) representing the species of the bird.
- **`Beak length, mm`**: A numerical column (float64 type) representing the length of the beak in millimeters.
- **`Beak depth, mm`**: A numerical column (float64 type) representing the depth of the beak in millimeters.

There are no missing values in any of the columns.

Next, I will generate descriptive statistics for the numerical columns to understand their distributions.

Okay, I ran some code:

```
# Generate descriptive statistics for numerical columns
beaks.describe()
```

The descriptive statistics for the numerical columns are as follows:

- **`Beak length, mm`**:
  - Mean: 11.35 mm
  - Standard deviation: 1.63 mm
  - Minimum: 8.70 mm
  - Maximum: 15.94 mm
  - 25th percentile: 10.20 mm
  - Median (50th percentile): 10.90 mm
  - 75th percentile: 11.80 mm
- **`Beak depth, mm`**:
  - Mean: 9.12 mm
  - Standard deviation: 0.71 mm
  - Minimum: 7.50 mm

  - Maximum: 10.98 mm
  - 25th percentile: 8.60 mm
  - Median (50th percentile): 9.10 mm
  - 75th percentile: 9.69 mm

These statistics suggest that the beak length has a wider range and variability compared to the beak depth.
Next, I will visualize the distributions of these numerical columns to better understand their patterns.

Okay, I ran some code:

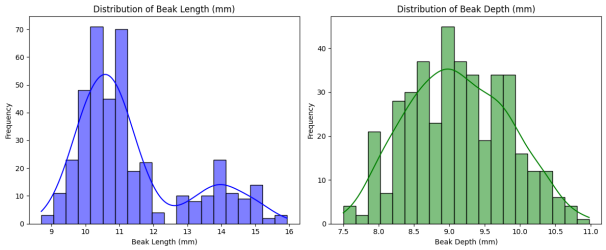
```
# Visualize the distributions of numerical columns
import matplotlib.pyplot as plt
import seaborn as sns

# Plot histograms for beak length and depth
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

sns.histplot(beaks['Beak length, mm'], bins=20, kde=True,
             ax=axes[0], color='blue')
axes[0].set_title('Distribution of Beak Length (mm)')
axes[0].set_xlabel('Beak Length (mm)')
axes[0].set_ylabel('Frequency')

sns.histplot(beaks['Beak depth, mm'], bins=20, kde=True,
             ax=axes[1], color='green')
axes[1].set_title('Distribution of Beak Depth (mm)')
axes[1].set_xlabel('Beak Depth (mm)')
axes[1].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



The histograms reveal the following about the distributions of the numerical columns:

1. **Beak Length (mm)**:
   - The distribution appears bimodal, with two peaks around 11 mm and 14 mm. This suggests the presence of two distinct groups or species with different beak lengths.
   - The range spans from approximately 8.7 mm to 15.9 mm.
2. **Beak Depth (mm)**:
   - The distribution is unimodal and roughly symmetric, centered around 9 mm.
   - The range spans from approximately 7.5 mm to 11 mm.

These patterns suggest that the dataset may represent multiple species with distinct beak characteristics, particularly in terms of length. The depth, however, shows less variability and a more uniform distribution.

4. Conclusion

The dataset contains information about bird species and their beak measurements. Key observations include:

- The `species` column categorizes the data, likely corresponding to the two distinct groups observed in the beak length distribution.

- Beak length shows greater variability and a bimodal distribution, while beak depth is more consistent and symmetric.

This dataset is well-suited for exploring relationships between species and beak characteristics, potentially revealing ecological or evolutionary patterns.