

LEO: an LLM-Powered EDA Overview

by

Sophia Zheng

BS Electrical Engineering and Computer Science, MIT, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

© 2025 Sophia Zheng. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Sophia Zheng
Department of Electrical Engineering and Computer Science
May 18, 2025

Certified by: Arvind Satyanarayan
Assistant Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

LEO: an LLM-Powered EDA Overview

by

Sophia Zheng

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

ABSTRACT

Computational notebooks impose a linear structure that impedes data analysts' sense-making process with overwritten cells, dead-end code, and fragmented logic. This challenge is especially pronounced when analysts either encounter a notebook authored by someone else or revisit a self-authored notebook after significant time has passed. In both cases, understanding the analysis code becomes convoluted and laborious. To address these barriers, we introduce LEO, a computational notebook tool that operationalizes notebook summarization by leveraging large language models to (1) cluster analysis patterns and (2) trace variable use. LEO organizes code into a two-level hierarchy—General Level Sections and Code Level Actions—integrated with in-line textual summaries filtered on the variable-level, further supporting task-driven exploration. We evaluate the system's effectiveness in a user study with five computational notebook users across two realistic use cases. Participants reported that LEO streamlined code comprehension and navigation of undocumented notebooks by allowing them to query variables and traverse code cells with greater ease.

Thesis supervisor: Arvind Satyanarayan

Title: Assistant Professor of Computer Science

Acknowledgments

I am deeply grateful to my supervisor, Arvind Satyanarayan, and my mentor, Dylan Wootton, for their unwavering support and innovative collaboration. Under your guidance, I have grown immensely as a researcher, student, and person.

To the MIT Visualization Group—you've created a home on Stata's 7th floor that has been one of the best parts of my five years at MIT. Thank you to everyone, especially my fellow MEng buddies, for your company and collaboration. Each one of you never fails to inspire me, and I am excited to see all that you continue to accomplish in the future!

To all of the wonderful users who tested LEO from the start—thank you for your time, patience, and feedback! Our tool has no purpose if not for you.

To all my loved ones, family, and friends, words cannot express how lucky I feel that you are in my life. Thank you for all your unconditional love and support. I can't wait to celebrate this milestone with you!

Last but not least, a special thanks to the namesake of this project, my 6-year-old forever puppy, Leo.

Contents

<i>List of Figures</i>	9
<i>List of Tables</i>	11
1 Introduction	13
2 Related Works	17
2.1 Porpoise	17
2.2 InterLink	18
2.3 Generative AI in Data Science	18
2.3.1 WaitGPT	19
3 Usage Scenario	21
4 LEO System	25
4.1 System Overview	25
4.2 Design Goals	25
4.2.1 Formative Studies	26
4.2.2 Seamless Embedding	27
4.2.3 On-Demand Thematic Groupings	27
4.2.4 Dynamic Scope Control	28
4.3 Targets of Summarization	28
4.4 Implementation	29
4.5 Tree Outline	30
4.5.1 Hierarchy Structure	30
4.5.2 Headings and Subheadings	31
4.6 Variables Pane	31
4.6.1 Querying Variables	32
4.6.2 Textual Summaries	33
4.7 Scope Integration	34
4.7.1 One Sentence Summary	34
4.7.2 In-line Textual Summaries	34
4.8 Bidirectionality	35
4.8.1 Tree-to-Notebook	35
4.8.2 Notebook-to-Tree	36

5	Evaluation	37
5.1	Interview Structure	37
5.2	Results	38
5.2.1	Supporting Code Comprehension in Unfamiliar Notebooks	38
5.2.2	Multi-Scope Summarization for Targeted Investigation	40
5.2.3	Complementary User Flow Paths	40
5.2.4	Clarification of Poorly Documented Code	41
5.2.5	Interaction Behavior to Record Sensemaking	42
5.2.6	Code Outputs to Recall Insights	42
5.2.7	Trust of AI Summarization	43
6	Discussion and Future Work	45
6.1	Annotation	45
6.2	Expanding Targets of Summarization	46
A	GPT Prompting	47
A.1	Tree Outline Prompt	47
A.2	Variable Textual Summary Prompt	48
	References	51

List of Figures

3.1	An analyst's workflow with LEO.	24
4.1	Overview of LEO's interface.	26
4.2	Diagram of our data pipeline for processing computational notebooks.	29
4.3	LEO's hierarchical tree outline with headings and subheadings expanded.	30
4.4	Pre-populated entry point for variable selection.	32
4.5	Search to add entry point for variable selection.	33
4.6	Hover to add entry point for variable selection.	33

List of Tables

4.1	Common pain points reported by notebook users we interviewed.	26
5.1	Demographic information of user study participants.	37

Chapter 1

Introduction

Data analysts typically begin their workflow with an Exploratory Data Analysis (EDA), a free-form coding session in which they examine dataset attributes, compute descriptive statistics, and create visualizations to better understand the data [1]. Without the pressure to prove a specific hypothesis, analysts conduct EDAs to explore the data without limitations, often uncovering patterns, validating assumptions, and generating insights otherwise difficult to discover by looking at the raw dataset [2]. Computational notebooks have become the de facto environment for conducting this detective work because of their support for flexible experimentation and rapid iteration [3]. However, this freedom can quickly become convoluted and difficult to follow as analysts execute code cells out of order, fail to delete irrelevant code, and attempt to fit tree-like analyses in linear notebooks [4]. Consequently, these notebooks turn into tangled and unorganized chains of long execution traces that are non-linear, difficult to parse, and hard to reproduce [5]. This exacerbates the cognitive load for analysts to maintain a clear and strong mental map of the notebook [6], encountering friction in reconstructing variable provenance and the chronological flow of code.

Although this issue persists throughout continuous EDA, it is particularly challenging when an analyst makes sense of an unfamiliar notebook. In this paper, an unfamiliar notebook is defined as a notebook authored by someone else or a self-authored notebook that is being

revisited after a significant period of time has passed. Our formative studies reveal that analysts approaching an unfamiliar notebook are most often goal-driven [7]. As the first step of the sensemaking process, analysts must familiarize themselves with the structure and content of the notebook [6]. This process is laborious, making it difficult to accomplish the goal at hand. For example, an analyst tasked with extending a previously conducted EDA will need to examine the notebook authored by their coworker. This task involves both grasping the overall analytical flow and specific transformations applied to each variable to then identify the most interesting insights and continue the EDA as if they were the original author. In practice, this dual requirement is tedious: high-level patterns remain obscured in blocks of code, while detailed usage of dataframes, visualizations, and functions is buried across fragmented cells. To bridge this gap, we leverage notebook summarization to facilitate code comprehension for targeted task completion.

A review of past literature reveals limitations in current attempts to fully support understanding the code of unfamiliar notebooks. Existing tools focus narrowly on either cell-level behavior [8] or isolated variable traces [9], and often present their findings through dense node-edge graphs [10–15].

To address these gaps, we present LEO, a lightweight VS Code extension that automates notebook summarization through a structured framework for understanding notebook code at both the top-level and detailed-level via concise textual summaries. We leverage large language models (LLMs) to generate structured summaries that capture overarching analysis groups and track variable usage to orient analysts making sense of unfamiliar notebooks, supporting their code comprehension and thus streamlining goal completion.

In LEO’s interface, the analysis code is summarized into a hierarchical tree outline populated at two nested levels:

1. General Level Sections — functional analysis groups
2. Code Level Actions — specific operational groups

This hierarchy can then be filtered on the variable-level, seamlessly integrating in-line textual summaries that trace a variable’s usage through the notebook in the context of the analysis groupings. We identify that applying a variable-based filter during summarization effectively traces variable usage throughout the notebook. This leverages a scoped narrowing of summarization to surface relevant information to the analyst for their task. With these levels, LEO acts as a control panel for different scopes of summarization, facilitating the selection of targets in the notebook editor to be explained by the tool. By combining scoped summarization with a clear, two-tiered outline, analysts can switch between big-picture exploration and fine-grained investigation with minimal context switching.

We evaluate LEO’s usability and robustness through five first-use studies on participants with previous data science experience. Users were presented with two unfamiliar notebooks, encompassing both use cases defined earlier. Overall, participants found that LEO’s structured summaries reduced the effort necessary to comprehend unfamiliar code, allowing them to complete the assigned tasks with more ease and confidence than they would have without the tool. All expressed enthusiasm for incorporating LEO into their own analysis workflows.

Future work on LEO can extend its utility by summarizing not only the semantic meaning of code, but also markdown annotations and output artifacts to capture more information and context. The addition of features for users to directly externalize their thought process throughout sensemaking such as built-in annotation further bridges the challenges of context switching [16, 17]. Exploring these directions is promising for extending LEO’s applicability and utility.

Chapter 2

Related Works

There is a rich precedence of computational notebook tools developed to simplify manual EDA tasks ranging from data profiling [18] and notebook organization [19] to interactive data exploration [20, 21]. While some tools utilize notebook summarization [6], they tend to target documentation [22] or educational [23] use cases and often adopt a more narrative style [24–26]. In contrast, our system focuses on concise textual summaries that streamline notebook comprehension for targeted task completion in unfamiliar notebooks by leveraging AI automation.

2.1 Porpoise

Porpoise provides an interactive overlay for labeling code sections with short, introductory descriptions and supports direct cell annotation [6]. Although both Porpoise and LEO are motivated by improving notebook understanding, Porpoise serves primarily as a design probe to operationalize design, not summarization. Unlike LEO’s LLM-powered groupings, Porpoise relies on heuristic matching against a predefined set of manually classified labels. Its scope is also limited to high-level clustering, leaving out finer-grained details.

2.2 InterLink

InterLink [27] is a Jupyter plugin designed to help analysts read computational notebooks by explicitly visualizing the relationships among text, code, and outputs. It achieves this with a two-column layout: separating the text from the code and outputs. Relevant descriptive text is aligned with its corresponding code and output, displaying these side by side connections for immediate reference. This transforms the linear, scroll-intensive structure of notebooks into an interleaving, digestible narrative. However, InterLink’s effectiveness depends on well-written markdown annotations—an assumption of good documentation that often fails in practice [28]. In contrast, LEO aims to support poorly documented notebooks through automated code summarization, enabling deeper understanding even in sparsely annotated notebooks.

2.3 Generative AI in Data Science

Generative AI is increasingly being leveraged to automate the work practices of data scientists. Most commonly, tools targeting LLM-powered code or insight generation [29] have emerged, automating routine tasks like data processing, feature engineering, and model creation. However, analysts’ responses to these advances remain mixed. Some express skepticism that automation will replace their hands-on workflows, while others are optimistic about a hybrid and collaborative future where AI completes low-level tasks while humans can focus their expertise on domain-specific sensemaking [30].

Instead of generating new code, LEO leverages LLMs to summarize existing notebooks, supporting analysts at the beginning of their sensemaking processes by helping them quickly comprehend what, where, and when code executes, rather than why. By focusing on notebook navigation and code comprehension rather than code synthesis, LEO supplements analysts’ domain expertise without attempting to replace it.

2.3.1 WaitGPT

WaitGPT is an LLM-powered data analysis tool that reimagines a traditional chatbot interface by providing real-time summaries of generated blocks with visual annotations and a node-edge graph to trace data provenance [31]. In particular, LEO prioritizes leveraging LLMs to generate an outline of existing code. We further extend this idea to computational notebooks with a built-in interface. Unlike WaitGPT where code blocks are summarized in isolation, LEO summarizes code blocks within the broader notebook because of the inherent cell structure of computational notebooks. Furthermore, whereas WaitGPT supports on-the-fly sensemaking, LEO is optimized for retroactive comprehension enabling users to understand existing analyses.

Chapter 3

Usage Scenario

To contextualize how LEO aids in notebook comprehension, we present an example use case shown in Figure 3.1. This usage scenario highlights how a user may interact with the tool to better navigate an unfamiliar notebook and understand code they have not previously seen.

In this walkthrough, we imagine that Dana is a data scientist working with the historical archives at the *Wiener Zeitung*, a newspaper organization based in Berlin. Recently, they've been losing readers to the *Salzburger Intelligenzblatt*, and Dana is tasked with analyzing decades-old circulation records and article topics to investigate the factors that drove historical shifts in readership to that rival publication. One of Dana's colleagues has already begun an analysis. They pulled articles from the *Salzburger Intelligenzblatt* using public archives and started a topic modeling analysis. Unfortunately, the colleague became ill before finalizing the study, and Dana has now been tasked with reviewing and completing their notebook.

When Dana opens the notebook in VS Code, she is immediately confronted with lengthy, sparsely documented code. To address this, she opens LEO, an extension developed to summarize computational notebooks and aid in code comprehension. Upon launch, LEO automatically generates a hierarchical outline of the notebook, presenting headings that reflect the content of her colleague's analysis **1**.

Dana expands the **Data Analysis** heading hoping intermediate visualizations might give

her some context. She clicks on the **Plot Data Distribution** subheading which scrolls her notebook editor to a series of charts **2**. However, she notices that each plot references opaque variables like `wz` and `sz`. A `Cmd`+`F` search fails to show her their definitions, suggesting that these variables were introduced by a complex preprocessing pipeline earlier in the notebook.

Noticing that LEO also has `wz` listed as one of the most frequently used variables in the notebook, Dana clicks its pre-populated tag in the variables pane. This instantly augments the outline with in-line summaries, and the one sentence header reveals that `wz` is a dataframe containing *Wiener Zeitung's* cleaned, preprocessed historical text corpus, named after the publication's initials. LEO additionally details the specific transformations applied to `wz`, including column slicing, year-based filtering, and OCR cleanup, giving Dana clarity on the variable's processing steps **3**.

While skimming the in-line summary for `sz`, Dana notices the subheading **Prepare and Visualize Issues Data**. Clicking this node jumps the notebook to its associated code cell, where she finds `sz` alongside the variable `df_issues`. She initially assumes `df_issues` must capture data quality issues in the corpus, but the displayed bar chart only shows article counts by year. To clarify its purpose, Dana hovers over `df_issues` and selects the **Summarize the variable "df_issues"** command **4**.

Dana then clicks on the in-line summary sentence detailing the dataframes initialization, navigating her to the relevant code in the editor **5**. Between the in-line summaries and the code, she discovers that `df_issues` is not data quality issues, but rather the original `wz` and `sz` datasets grouped by year, representing the number of annual publications. By surfacing the variable's usage and derivation steps, LEO clarified the semantic meaning of `df_issues`, helping Dana in correcting her initial misinterpretation.

Through this iterative process, she is able to understand her colleague's EDA process and confidently proceeds where they left off, exploring topic-modeling analysis to identify the reasons behind shifts in readership.

1

Search for variables...

top_n_bigrams x get_top_ngram x
df_news x sz x wz x

This notebook performs data preparation, analysis, and visualization for textual and statistical information about two datasets, SZ and WZ.

- Initial Setup
- Data Loading and Cleaning
- Data Analysis
- Advanced Data Processing
- Text Similarity and Ngram Analysis
- Cleaned Text Analysis

```

import os
import csv
import pathlib
import pandas as pd
import numpy as np
import seaborn as sns
import nltk
import re
import statistics
import string
from collections import Counter
from collections import defaultdict
from sklearn.model_selection import train_test_split
import os
import nltk
import re
import statistics
import string
from collections import Counter
from collections import defaultdict
from sklearn.model_selection import train_test_split
import nltk
import gensim
import spacy
import sklearn
import torch
import libsvm
import librosa
import sentence_transformers
import requests
import tomotopy

from matplotlib_venn import venn2, venn2_circles, venn2_unweighted
from matplotlib_venn import venn3, venn3_circles
  
```

2

Search for variables...

top_n_bigrams x get_top_ngram x
df_news x sz x wz x

This notebook performs data preparation, analysis, and visualization for textual and statistical information about two datasets, SZ and WZ.

- Initial Setup
- Data Loading and Cleaning
- Data Analysis
 - Aggregation and Basic Statistics
 - Numerical Analysis
 - Plot Data Distribution
 - Temporal Trends in Data
- Advanced Data Processing
- Text Similarity and Ngram Analysis
- Cleaned Text Analysis

```

sns.displot(wz, x='num_words').set(title='WZ word distribution')
  
```

3

Search for variables...

top_n_bigrams x get_top_ngram x
df_news x sz x wz x

This variable is a dataframe containing processed data from a TSV file of "WZ" newspaper issues.

- Initial Setup
- Data Loading and Cleaning
 - Load Datasets
 - The data for "wz" is loaded from a TSV file with UTF-8 encoding.
 - Remove Unnecessary Columns
 - Non-essential columns are removed using column slicing.
 - Extract Numerical Data
 - A "year" column is extracted from the "manifest_id" using regex matching.
 - Filter Dataset by Criteria
 - Extracted years are cleaned and converted to integers.
 - Rows with years before 1789 are filtered out.

```

sns.displot(wz, x='num_words').set(title='WZ word distribution')
  
```

4

Search for variables...

top_n_bigrams x get_top_ngram x
df_news x sz x wz x

The word distributions and trends are visualized.

Temporal Trends in Data

Advanced Data Processing

Merge and Categorize Datasets

Prepare and Visualize Issues Data

Text Analysis

The text entries are split into individual tokens and cleaned of non-alphanumeric characters.

A frequency analysis of cleaned tokens is performed.

Text Similarity and Ngram Analysis

Text Similarity Calculations

Building Ngram Models

Common ngrams such as bigrams and pentagrams are identified.

Cleaned Text Analysis

EDA.ipynb > M word distribution > df_issues = pd.concat((wz_issues, sz_issues)).reset_index(drop=True)

```

+ Code + Markdown | Run All | Restart | Clear All Outputs | Jupyter Variables ... Python 3.12.1
1795 52 52 52 52
1796 55 55 55 55
1797 54 54 54 54
1798 53 53 53 53
1799 52 52 52 52

```

```

sz_issues['newspaper'] = 'sz'
(variable) df_issues: DataFrame
Summarize the variable "df_issues"
df_issues = pd.concat([wz_issues, sz_issues]).reset_index(drop=False)

```

```

df_issues
year ocr split manifest_id num_words newspaper
0 1789 104 104 104 104 wz
1 1790 104 104 104 104 wz
2 1791 105 105 105 105 wz
3 1792 104 104 104 104 wz
4 1793 105 105 105 105 wz
5 1794 105 105 105 105 wz

```

5

Search for variables...

df_issues x top_n_bigrams x
get_top_ngram x df_news x
sz x wz x

newspapers, wz and sz into a single dataframe.

Initial Setup

Data Loading and Cleaning

Data Analysis

Advanced Data Processing

Merge and Categorize Datasets

Prepare and Visualize Issues Data

The dataframe is formed by grouping and counting issues per year for each newspaper.

It includes columns for issue counts, newspaper identifiers, and years.

The data supports visualization of issue distribution by year and newspaper type.

Text Analysis

Text Similarity and Ngram Analysis

Cleaned Text Analysis

Users > sophiazheng > Downloads > EDA.ipynb > M word distribution > df_issues = pd.concat((wz_issues, sz_issues))

```

+ Code + Markdown | Run All | Restart | Clear All Outputs | Jupyter Variables ... Python 3.12.1
wz_issues
ocr split manifest_id num_words
year
1789 104 104 104 104
1790 104 104 104 104
1791 105 105 105 105
1792 104 104 104 104
1793 105 105 105 105
1794 105 105 105 105
1795 104 104 104 104
1796 105 105 105 105
1797 103 103 103 103
1798 104 104 104 104
1799 52 52 52 52

```

```

wz_issues['newspaper'] = 'wz'
sz_issues = sz.groupby('year').count()

```

Figure 3.1: An analyst's workflow with LEO.

Chapter 4

LEO System

4.1 System Overview

LEO is a tool that generates structured summaries of computational notebooks to streamline notebook comprehension. LEO’s interface, depicted in Figure 4.1, contains two main sections: a tree outline and a variables pane. Through the tree outline, LEO supports a top-down exploration of notebooks. The tree is populated with nested LLM-generated headings, clustering different analysis steps present in the code (Figure 4.1 **b**). The variables pane supports a bottom-up investigation by allowing analysts to select any variable defined in the notebook to track its usage (Figure 4.1 **a**). Selecting, or querying, a variable generates in-line textual summaries that track variable use, targeting the semantic meaning of code across the notebook (Figure 4.1 **c**).

4.2 Design Goals

Our implementation of LEO was guided by several overarching design goals derived from formative studies in addition to our literature review.

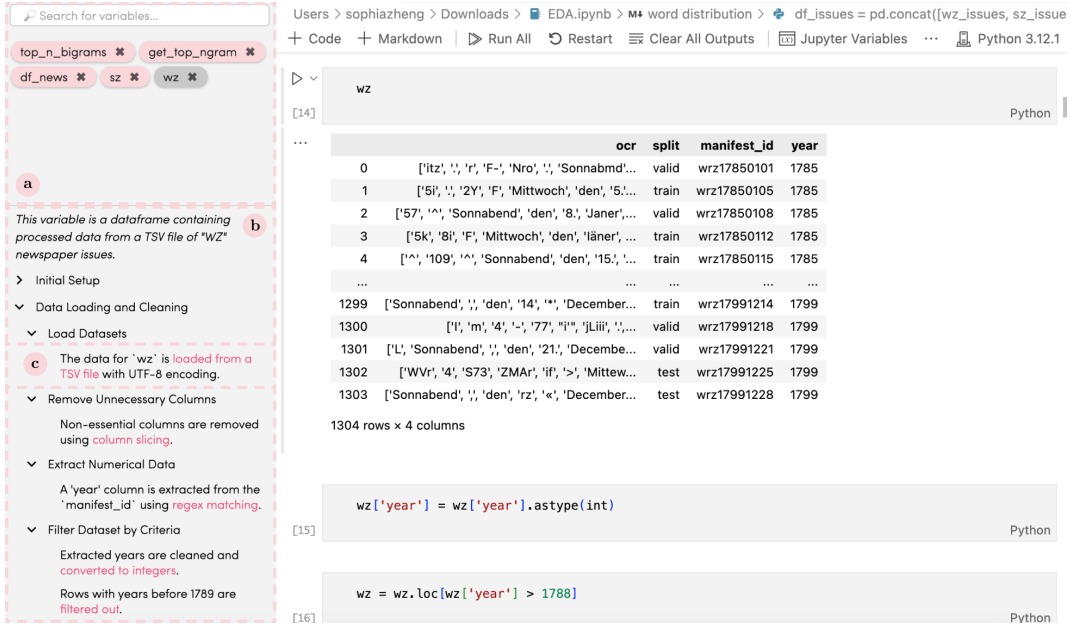


Figure 4.1: Overview of LEO’s interface.

4.2.1 Formative Studies

We conducted formative studies early in LEO’s development to identify and understand pain points experienced by computational notebook users to guide our design.

Issue Type	Pain Points
Structural	<ul style="list-style-type: none"> • Linear cell layout makes it hard to connect code to functionality • Out-of-order or interleaved cells are difficult to keep track of • Long notebooks are difficult to divide into meaningful sections • Even small edits can substantially lengthen the notebook • No intuitive way to separate code into modules or files
Interaction	<ul style="list-style-type: none"> • Cannot link comments or notes to specific variables or cells • Lack of dynamic updates (i.e. changes in one cell don’t propagate automatically)
Technical	<ul style="list-style-type: none"> • Runtime environment inconsistencies and frequent re-installation requirements • Difficult to perform with large datasets

Table 4.1: Common pain points reported by notebook users we interviewed.

Table 4.1’s top pain points reflect a struggle of sensemaking and code tracking imposed by the strictly linear environment of notebooks. To scope our solution, we target two realistic use cases:

1. Newly navigating a notebook authored by someone else

2. Reentering a self-authored notebook after a long time has passed

When asked about their purposes for entering unfamiliar notebooks, users responded that they are often task oriented. One example described by a user was that she sometimes enters a colleague’s notebook to extend their EDA, but only has interest in one specific attribute of the original dataset. However, she still needs to understand the content of the notebook at a high-level, as well as the more detailed usage of that specific attribute to complete this task. Without support, these tasks quickly become overwhelming and unwieldy due to the complexity of notebooks, often containing hundreds of variables. LEO is designed to address these challenges.

4.2.2 Seamless Embedding

Because of the rapid, iterative nature of analysis workflows, it is crucial to integrate notebook tools smoothly with the notebook layout to prevent workflow disruption [29, 30]. Our interface design will prioritize lightweight notebook navigation [31] to minimize the amount of context-switching analysts have to do throughout their analysis while seamlessly surfacing information that maximizes context preservation.

4.2.3 On-Demand Thematic Groupings

In the formative studies, users gave strong signals that a tree outline was helpful to gain a quick, concise overview of the code, with one participant exclaiming that the tree is “good for me as a developer.” For example, users stated that they often forget if variables are used and whether they are renamed throughout the notebook, requiring them to expend mental energy on manually investigating that variable’s usage. Ideally, our tool would help reduce this cognitive overload, supporting the analyst in more easily and confidently approaching their task by scaffolding the onset of their sensemaking process with digestible summaries. With the tool, we wanted to prioritize the ability to transform something unfamiliar into a

familiar structure with less words.

Some users mentioned that more user customization could be desirable such as adjusting the tone or depth of the LLM-generated summaries, but we chose to omit this in the system as a tradeoff to reducing more cognitive load and responsibility on the user, opting to focus more on the strong scaffolding and guidance the structured summaries could provide.

4.2.4 Dynamic Scope Control

In an earlier version of the tool, we experimented with LLM-generated textual summaries at the scope of the entire notebook. When presented with this scope of text, one user was skeptical. While it was accurate by content, it described certain actions across multiple sentences while describing others in just one sentence. Particularly, she was surprised it only described what she felt was the “main point of her notebook” in one sentence out of the entire paragraph. The AI placed emphasis misaligned with the user’s understanding of her own notebook.

If the scope is too broad, the LLM may generate a textual summary that does not place importance on the same functionality envisioned by the analyst. At the same time, the LLM was starting to reason why operations were conducted in the notebook, a design decision we purposely stray from. Because an overall textual summary of the entire notebook was insufficient, controlling the scope of summarization by filtering on variables became an important interaction element. This informed us on what scopes of summarization to process and our choice to focus on operationalizing summarization for notebook comprehension at the beginning of the sensemaking process.

4.3 Targets of Summarization

The entities an analyst would want to summarize on (content) must exist at some level (scope). When trying to identify targets of summarization, we first enumerated all combinations of

content (code, outputs, or markdown) at different scopes (cell, block, or notebook).

In our formative studies, users expressed that the tool was most helpful for poorly documented notebooks, leading us to summarize only the semantic content of code, omitting markdown and outputs. Applying multi-scope summarization to code can be achieved through hierarchical grouping of analysis steps with broad functionality at the top-level and progressively more detailed operations after. Furthermore, we found it most salient that filtering code on a specific variable can also produce these summaries at the different scopes:

1. Cell: All code in one cell related to the variable
2. Block: All code across multiple cells related to the variable
3. Notebook: Traces the variable's usage throughout the entire notebook

These findings directly inform our summarization framework, described in the following subsections.

4.4 Implementation

The following sections detail our lightweight interface for presenting computational notebooks after they have been processed by our data pipeline as shown in Figure 4.2. Subsections will describe their corresponding stage in the data pipeline.

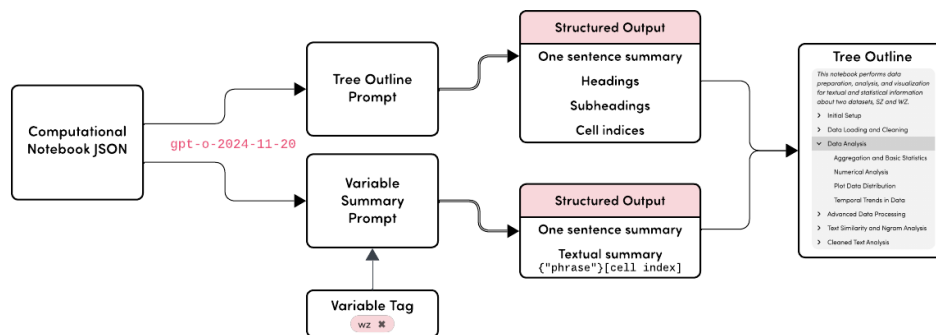


Figure 4.2: Diagram of our data pipeline for processing computational notebooks.

4.5 Tree Outline

The tree outline is populated with two-level heading descriptors summarizing the notebook’s entire analysis code.

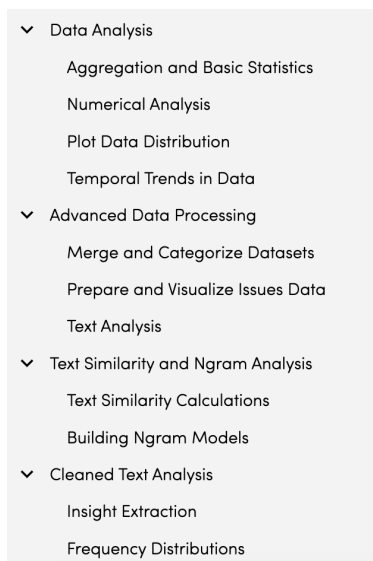


Figure 4.3: LEO’s hierarchical tree outline with headings and subheadings expanded.

4.5.1 Hierarchy Structure

The hierarchical structure of the tree provides clear signposts and thematic navigation, giving users an “information scent” [32] to guide them through complex notebooks [33]. It visually represents the organization of the structured summaries on two levels of clustering code, making it easy for users to understand the relationships between headings and subheadings. By grouping code cells on functionality, we leverage Gestalt’s proximity principle [34]. Users will naturally associate node groupings with their respective code sections, highlighting coherent analysis patterns. Simultaneously, the continuity principle is reinforced via indentation levels, encoding hierarchical relationships and shifts in granularity of the heading descriptors.

While the tree supports both expansion and collapsing of nodes, its initial state collapses all nodes. We follow the principle of progressive disclosure to prevent information overload

on first view, enabling users to incrementally expand headings of choice and gradually reveal details as relevant to the task at hand [35].

Together, these features minimize cognitive overhead, empowering analysts to swiftly traverse the outline and digest the tool’s summary headings by acting as a semantic table of contents for users, especially in poorly documented notebooks lacking any organization. LEO makes navigation easy while only targeting summarized information at the scope pertinent to the user.

4.5.2 Headings and Subheadings

We organize the tree into exactly two levels of hierarchy—General Level Sections and Code Level Actions. We found in testing that this balances the broad, notebook-agnostic headings with more notebook-specific subheadings. At the top-level, each heading describes a common analysis step (e.g. “Data Cleaning”, “Feature Engineering”, and “Visualization”) universally seen in notebooks. Expanding these reveals a second tier of subheadings, disaggregating the headings into more concrete and notebook-specific actions as shown in Figure 4.3. For example, “Data Analysis” is composed of “Numerical Analysis” and “Temporal Trends in Data”, encoding information specific to the dataset loaded in that unique notebook.

Adopting a two-level hierarchy aligns with Miller’s Law, as the tree rarely populates with more than 5-7 headings, each with 3-5 subheadings, keeping users’ efforts to remember the content of the much larger notebook within reasonable capacity [36]. It also facilitates a shallow information hierarchy, further reducing navigation effort. For full details on how we prompt the LLM to generate both levels, see Appendix A.1.

4.6 Variables Pane

In formative studies, users most commonly would attempt to summarize variables out of all code constructs. Thus, LEO’s variables pane supports persisting and querying any variable

in notebook content.

4.6.1 Querying Variables

Variables are represented as clickable tags labeled with their names, which users can add or remove (persistent) and select or deselect (ephemeral). When a variable is queried, it is automatically added to the variables pane as a tag. Tags can be removed by clicking its “×” button. Adding a tag automatically selects, and clicking it will toggle between deselection and selection.

Users can query variables through three entry points, with two directly within the variables pane and a third embedded in the VS Code notebook editor.

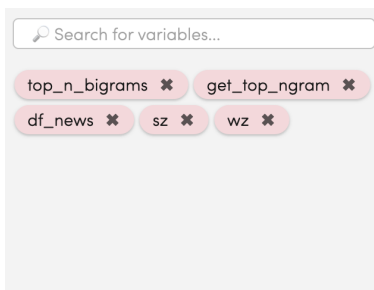


Figure 4.4: Pre-populated entry point for variable selection.

1. Pre-populated by top frequency: At the start, the pane is seeded with the five most frequently referenced variables in the notebook as defined by direct calls and accesses as seen in Figure 4.4. This initial seeding provides immediate recommendation of variables to begin investigating and reduces the barrier to entry for analysts who might otherwise feel unsure how to interact with a blank interface.
2. Search-and-filter: A search bar at the top of the pane offers an auto-suggest dropdown including every variable name used throughout the notebook, sorted in descending order of frequency for similar signposting reasons as above. As the analyst types, the dropdown filters to matching variables, enabling targeted discovery even in large, complex notebooks. This can be shown in Figure 4.5.

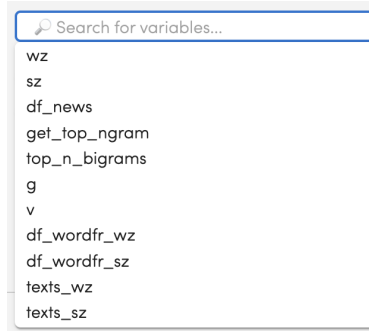


Figure 4.5: Search to add entry point for variable selection.

3. In-editor hover tooltip: We extend VS Code’s native hover tooltip [37] for Python identifiers to include a command to query variables as seen in Figure 4.6. Normally, the tooltips display type and documentation information. Embedding an entry point here provides a lightweight user flow to examine variables without leaving the notebook context.



Figure 4.6: Hover to add entry point for variable selection.

By supporting persistent tagging via multiple workflows, LEO minimizes context-switching and cognitive overhead, enabling analysts to surface and store relevant variables as they navigate complex analysis pipelines.

4.6.2 Textual Summaries

Querying a variable produces a concise textual summary filtered on the scope of the variable. This summary traces the queried variable’s usage in the context of the broader notebook, presenting information more relevant to their task. We still retain context of the entire notebook as we are simply applying a variable-based filter to the summarization. This approach identifies the code cells relevant to a variable, regardless of how many distinct

sections (headings) it may span and can break down the series of chained operations typical of fluent programming practices [38]. Consequently, these scoped textual summaries capture data transformations and code steps specific to each variable. Our formative studies confirmed that explicitly tracking data provenance in the current state of the code is essential for notebook comprehension as participants expressed frustration with frequently losing track of variable usage, especially after reassignment. By automating what was once a manual, tedious process, LEO enables analysts to rapidly understand and contextualize variables. For the complete prompt used to guide the LLM in generating these variable-level summaries, see Appendix A.2.

4.7 Scope Integration

So far, we’ve described two scopes of summarization:

1. A top-level hierarchical summarization of general analysis activities
2. A detailed-level textual summarization that traces variable usage

Importantly, our system integrates both scopes across its dual-pane interface.

4.7.1 One Sentence Summary

A dynamic, one sentence summary appears above the tree outline, updating in real time to reflect context. When no variable is queried, it describes the notebook’s high-level purpose. However, once a variable is queried, it instead summarizes that variable’s role in the notebook.

4.7.2 In-line Textual Summaries

The previously described variable-level summaries are seamlessly embedded within the tree outline’s hierarchical structure. Each summary is split into individual sentences, which are then matched to tree nodes by cell index as seen in the last stage of our notebook processing

pipeline in Figure 4.2. Because subheadings map to specific cells, filtering on a variable automatically inserts each sentence as a child of its corresponding subheading, dynamically extending the outline. To avoid information overload, the tree initially expands only the headings and subheadings that are parents to the in-line summary sentences, collapsing all other nodes. Phrases within each sentence are highlighted as the most salient code actions which are likewise linked to the exact cells of occurrence, ensuring that summaries appear in-line under their relevant headings. We use a one-to-many mapping approach—if a single sentence contains multiple key phrases, it can correspond to several distinct cells. Deselecting or removing a variable tag restores the tree to its original, unfiltered state. To optimize performance, the filtered tree with its in-line summaries is cached on the first query of a selected variable, so subsequent selections of the same variable render instantly. Unifying high-level overview and fine-grained detail in a single interactive view supports comprehensive notebook understanding by collapsing the gulf between both scopes, accelerating identification of relevant information beyond what either scope can provide alone.

4.8 Bidirectionality

By leveraging cell indexing, we support bidirectional integration [39, 40] between the tree outline and the notebook editor. This lightweight overlay of LEO’s functionality directly within the notebook minimizes the gap between summarizing and source code, providing click-to-navigate interactions that instantly highlight corresponding code segments and vice versa. By tightly coupling summaries with their source code, users can navigate the notebook with minimal scrolling [41] or searching.

4.8.1 Tree-to-Notebook

In the tree-to-notebook direction, children nodes act as the navigation anchor. Clicking any child node scrolls and centers the corresponding code cell in the VS Code editor. When a

tree node spans multiple cells, we default to the earliest cell by position index. Because of the one-to-many implementation for our in-line summary nodes, clicking the highlighted phrase within the summary sentence will directly navigate to the exact cell where the action occurred.

When the tree is unfiltered, this behavior applies to all subheading nodes. When the tree is filtered, this behavior applies to all in-line summary nodes, but only to subheading nodes without in-line summaries (i.e. irrelevant to the selected variable's provenance). Thus, the interactivity of parent nodes in our interface is only to expand or collapse their children rather than to navigate to source content. We choose to not trigger navigation for these parent nodes to prevent unintended jumps in the editor and reduce cognitive distraction.

4.8.2 Notebook-to-Tree

In the notebook-to-tree direction, clicking a code cell in the notebook editor will automatically expand and highlight its corresponding subheading, as well as scroll that node into center view. This behavior directs the user's attention to the source code's corresponding analysis step or operation while still preserving the tree's existing expanded nodes. We choose to not force collapse the previously expanded nodes to similarly prevent unintended navigation and minimize cognitive distraction as users may often be clicking cells for code editing rather than actual tree interaction.

Chapter 5

Evaluation

5.1 Interview Structure

PID	Job Role	Computational Notebook Experience
P1	Computer Science Master’s student	5 years
P2	Aeronautics and Astronautics PhD student	3 years
P3	Computer Science undergraduate student	2-3 years
P4	Civil and Environmental Engineering PhD student	2 years
P5	Computer Science PhD student	5-6 years

Table 5.1: Demographic information of user study participants.

To evaluate the usability of LEO, we conducted a first-use study with 5 representative users, including a mix of undergraduate, Master’s, and PhD students with prior experience in data science and machine learning. All participants regularly use computational notebooks in academic classes and research with experience ranging from 2-6 years. They represented disciplines including Computer Science, Aeronautics and Astronautics, and Civil and Environmental Engineering. Participants were recruited from classmates, colleagues, and friends within the MIT community. Each participant received a small token of appreciation for their time.

We conducted in-person, one-hour user sessions. Each participant used a provided laptop with VS Code preconfigured with LEO and the target notebooks. Sessions began with a

10-minute walkthrough of LEO’s core features. Participants then completed two timed tasks (20 minutes each), corresponding to our two primary use cases.

For Task 1, we asked participants to conduct a Quality Assurance (QA) review on a coworker’s EDA. Participants examined a poorly documented notebook performing topic modeling to compare historical newspaper corpora [42]. We selected this example to evaluate LEO’s ability to accelerate comprehension with ambiguous variable names and minimal markdown or comments.

For Task 2, participants were asked to revisit a self-authored notebook that they had not opened for at least three months, identifying their most interesting insight in preparation for extending the analysis. This use case probed LEO’s effectiveness in the case where an analyst was once familiar with the code but has since forgotten implementation details. These notebooks, ranging from 15 to 400 cells, covered EDAs and modeling scripts and tutorials. Before each task, participants rated their familiarity with the notebook on a five-point Likert scale ($\mu = 2.6$), typically recalling high-level objectives but not the specific code details. During both tasks, we encouraged users to vocalize their thoughts via a think-aloud protocol.

Finally, we concluded with a 10-minute semi-structured interview to gather qualitative feedback on LEO’s summarization and navigation features.

5.2 Results

Participants quickly grasped how being presented with the notebook summarization at the beginning of the task aided in navigating notebook content.

5.2.1 Supporting Code Comprehension in Unfamiliar Notebooks

All five participants (P1-P5) completed both tasks within the allotted 20-minute windows. In Task 1, everyone accurately identified that the notebook conducted topic modeling on two newspaper text corpora. In Task 2, all participants identified their most important

insight. This demonstrates LEO’s effectiveness for both users to understand new code and re-familiarize themselves with previously authored code.

Participants shared the sentiment that LEO helped them “dig themselves out of holes when they get lost in the details” and expressed interest in adopting the tool in the future for enhanced notebook comprehension. Several noted situations in which they routinely examine classmates’ or colleagues’ notebooks and were excited about LEO’s ability to streamline this process. For the second use case, P3 said that she could envision using LEO to “write her handoff document” as she has numerous notebooks that she needs to return to and summarize for colleagues taking over her work.

Interestingly, participants’ interaction behavior diverged across the two tasks, reflecting LEO’s distinct contributions in different contexts. In general, participants indicated that they depended more on the tool in Task 1 for gradual task completion. During Task 1, participants relied more on the hierarchical tree to obtain a high-level overview of the notebook and only investigated a small number of variable summaries later in the session to validate their understanding. In contrast, in Task 2, participants already had a high-level understanding of their notebook and felt more comfortable immediately diving into variable-level summaries immediately. Moreover, multiple participants (P1-P3) reported that they only depended on LEO until a single summary prompted an “aha” moment. For example, in our debrief, P3 mentioned “I forgot about [this insight] until I saw it in the tree,” at which point she instantly recalled the notebook’s key finding. After that discovery, she no longer needed much support from the tool. This behavior underscores the value of integrating both notebook-level and variable-level summaries. Depending on the use case, analysts may derive the greatest benefit from either scope first—and that scaffolding provided by LEO’s summary can dramatically accelerate the remainder of their exploration.

All but one participant found LEO beneficial for the Quality Assurance task. While P1 thought it was helpful for understanding completely new code, she questioned its utility in the context of QA, noting that LEO does not automatically detect faulty code and code

errors, requiring her to still conduct manual verification.

5.2.2 Multi-Scope Summarization for Targeted Investigation

Participants really valued the level of detail provided by LEO’s in-line variable summaries. For example, P3 pointed out that it was impressive that the summary “sliced [this variable] on the first column”. Likewise, P4 appreciated receiving details such as “dropping the fourth column.” These comments support the necessity of multi-scope summarization. By filtering on individual variables, analysts can surface the fine-grained information required for deeper code comprehension.

5.2.3 Complementary User Flow Paths

One common behavior observed was that participants would immediately begin exploration with the hierarchical tree outline, interleaving interactions between the tree, variable tags, and code cells. Typically, users would expand a heading, click a subheading to navigate directly to its corresponding cell, and treat the label as a concise brief before examining the code in detail. For example, on start, P1 and P3-P4 followed a systematic workflow consisting of opening a heading, clicking the first subheading to jump to its cell, review the code, and then repeat this node-cell examination sequentially. P5 went a step further by using the next subheading as an implicit boundary marker, helping her identify the functional “borders” of each code block before beginning to make sense of the code on her own.

All but one participant (P2-P5) adopted a systematic approach to variable selection as opposed to random sampling. Although LEO was designed to summarize user-defined variables, several participants (P2, P3) also queried external library functions, and others (P4, P5) investigated unfamiliar models. These behaviors suggest that LEO’s summarization scope could be broadened to include other types of code constructs.

Persisting variables to the pane acted as an important refresher to ground users’ task-driven exploration in actual code elements. Surprisingly, participants most often began to summarize

variables via the hover command rather than by selecting from the pane’s pre-populated most frequently used variables. Of the three available entry points, the search bar was the least utilized (P2, P3). In post-task debriefs, we presented a hypothetical scenario to retrieve a variable whose purpose was remembered but name was not with LEO. Most participants reported that they would begin with the search bar to explore all possible variables or filter by educated guesses. However, this strategy was less observed in practice. These results underscore the importance of offering multiple, complementary paths for variable querying as different users naturally gravitate toward different interaction techniques when navigating complex notebooks.

5.2.4 Clarification of Poorly Documented Code

Several participants noted their own notebooks suffered from sparse documentation and inconsistent naming conventions. In two notable cases, LEO’s summaries both validated and challenged these practices. P2 and P3 were impressed when the tool accurately summarized the purpose of a poorly named variable. On the other hand, P5 discovered through the summary that the name she gave a variable misaligned with its actual usage, prompting her to consider stricter naming in future code. These interactions not only facilitated comprehension, but also encouraged participants to reflect critically on their own coding practices.

Users also reported LEO accelerated their code comprehension and notebook navigation by reducing the cognitive burden of jumping around cells and having to context-switch between code and sensemaking. For example, many participants described the tool as crucial for clarifying undescriptive variables. The notebook primarily worked with two datasets—wz and sz—which offer no semantic information. P3 said that, without LEO, she would “just have to scroll through the entire [notebook] and check where [a variable] was initialized, where it was perhaps modified...”. Similarly, P2 remarked she would have had to “check every cell to get a sense of what the variable was doing”, but still doubted that she would have “gained context into what the dataset actually represented”. The only indication in the code

that these variables referred to historical newspaper corpora appeared in a few plot legends labeled “newspaper,” a breadcrumb easy to overlook without detailed variable summaries.

5.2.5 Interaction Behavior to Record Sensemaking

Because participants were already familiar with their own notebooks, some (P4, P5) bypassed a full expansion of every heading and directly jumped to the sections they remembered as most relevant, skipping steps like “Data Import.” P4 was even more intentional about only focusing on the most relevant tree headings. She collapsed headings after reviewing their associated cells if she found them not pertinent to the task at hand, effectively using the expand/collapse functionality to mark which sections to return to later. This use of expansion and collapsing as an impromptu bookmarking mechanism mirrors P3’s appreciation for persistent variable querying, reaffirming the value of integrating explicit annotation features within LEO’s interface in the future.

Participants also found the persistent caching of their queried variables and corresponding summaries helpful. P3 described the variables pane as a “great refresher” of her investigative process, signaling its role in tracking and contextualizing analysis steps. This feedback points to future work extending LEO with features that also document and externalize the analyst’s evolving thought process throughout the sensemaking workflow.

5.2.6 Code Outputs to Recall Insights

Participants enjoyed exploring their own notebooks with LEO as it was able to resurface forgotten details. For example, P5 remarked that it was “good that the summary matched what she thought occurred in the notebook, but also [code] she didn’t realize” had happened.

They often lamented that they could not remember what a plot was displaying in their notebook. In these cases, they reported that LEO clarified the purpose of certain forgotten visualizations in the context of all code cells. In particular, P4 and P5 credited the tool with revealing the data transformations driving key plots, which directly led them to their most

significant insights. When referring to two plots in her code, P4 found it amusing saying, “I did not realize that the one variable is the same in both plots.” She followed this statement with “Honestly, I’m not sure if I would have discovered this because, I don’t know. For some reason I thought they were different things.” By summarizing variables underpinning visual outputs, LEO streamlines code comprehension and facilitates. By summarizing the variables underpinning visual outputs, LEO streamlines code comprehension and facilitates the retrieval of insights that analysts had previously inferred but later forgotten.

5.2.7 Trust of AI Summarization

P1 and P2 reported that the LLM-generated top-level headings were occasionally too broad, leading them to rely more heavily on variable-level summaries than on the tree outline. This feedback signals the need for further investigation into the effectiveness of our two-level heading hierarchy in capturing diverse sensemaking strategies.

A couple participants (P1, P2) initially expressed skepticism for applying AI-driven summarization to their code. In particular, P2 noted that her own variable naming conventions are often vague or misleading, creating doubt in an LLM’s ability to accurately infer each variable’s purpose and concerns over potential hallucinations. To validate LEO’s reliability, she actually began Task 2 by selectively querying a few critical variables and comparing the generated summaries against her recollection. Upon observing a high degree of accuracy, she reported, “I was more inclined to trust the tool.” This behavior not only illustrates common hesitancy around Generative AI in data science [29, 30], but also affirms one of LEO’s applications to support analysts working with poorly documented notebooks and inconsistent naming schemes.

Chapter 6

Discussion and Future Work

In this section, we reflect on the limitations of the system and synthesize potential directions for future research to extend LEO’s applicability and utility.

6.1 Annotation

LEO is successful at summarizing notebook code across multiple scope, and our user studies confirmed that this significantly accelerates the first step in sensemaking–code comprehension. However, later stages of sensemaking require analysts to externalize and record their emerging insights [6]. In our formative studies (see “Interaction Issues” in Table 4.1), one participant lamented that traditional notebooks lack mechanisms for linking notes directly to variables or cells, as markdown is still bound to the linear structure.

Although LEO does not currently support in-house annotation, several study participants surprisingly utilized its persistent tagging and collapsible headings to preserve and revisit their observations. These behaviors signal a clear need for lightweight annotation features.

To address these gaps, future versions of LEO could include text highlighting, note-taking boxes, in-tool editing, and persistent tree-state caching. Highlighting text and note-taking boxes allow analysts to mark and comment on particular code sections and summaries with questions, remarks, and insights to revisit as they navigate the notebook. Caching tree states

enables analysts to save and restore specific expanded/collapsed views.

By integrating these annotation techniques, LEO would support analysts in capturing their insights as they navigate complex workflows, facilitate collaborative review by embedding shared comments, and better streamline the complete sensemaking process.

6.2 Expanding Targets of Summarization

Additionally, future iterations of LEO should broaden the content for summarization beyond just code to include both outputs and markdown. Informed by our formative studies, we deliberately omitted these targets and chose to focus only on code because we observed that users can typically understand well-documented notebooks without assistance. However, our subsequent user studies on the current state of the tool revealed usage scenarios where reintegrating outputs and markdown into the processing pipeline could be useful. For instance, one participant wanted to compare LEO’s automatically generated outline with her existing markdown headings, suggesting that synchronizing the two hierarchies could make the tool valuable even in well documented environments. Several participants also mentioned that lightweight summaries of cell outputs would be helpful. Rather than adding more text, LEO could display simple visual cues, such as badges or tooltips indicating the count of visualizations or dataframes within each section to further guide notebook navigation. LEO’s current design combining variable summaries and native notebook outputs already helped users grasp dataset contents in Task 1 and surface their most significant insights in Task 2. By expanding LEO’s pipeline to process outputs and markdown alongside code, the system would generate a more holistic overview of notebook content, further aiding notebook comprehension, insight discovery, and EDA collaboration. Ultimately, extending LEO’s targets of summarization to encompass code, outputs, and markdown across multiple scopes will enhance its utility across the full range of computational notebooks and EDA tasks.

Appendix A

GPT Prompting

We prompt OpenAI's `gpt-4o-2024-11-20` model.

A.1 Tree Outline Prompt

Below is the exact prompt used to elicit the tree summary discussed in Section [4.5.2](#):

```
You're given a JSON array of notebook code cells. Produce a JSON output with this structure:
```

```
{
  narrative: string,           // one-sentence summary of the notebook's overall purpose
  groups: [
    {
      name: string,           // broad functional group label
      subgroups: [
        {
          name: string,       // more specific subgroup label
          cells: number[]    // array of cell ids
        }
      ]
    }
  ]
}
```

Rules:

1. Use as much context as possible in the code to name each group and subgroup.

2. **Every single cell id number must appear exactly once** in one and only one subgroup's 'cells' array.
 - Do not omit any cell id.
 - **Every id number from 0 to `codeCells.length - 1`**, inclusive must be included in the output.
 - No cell ID may be skipped, missing, duplicated, or invented.
 - Do not repeat a cell number in more than one place.
3. The order of cells in each subgroup can be ascending or based on logical flow.

Here is the input JSON. Label the cells by their 'id':

```


${codeCells
  .map(
    (cell, i) =>
      'Block ${cell.id}:\n${cell.source.join('\n')}'
  )
  .join('\n\n')}


```

Listing A.1: Full GPT-4o prompt used for generating the tree outline.

A.2 Variable Textual Summary Prompt

This is the variable summary prompt from Section 4.6.2:

```

You are an expert at writing concise, factual variable summaries.

Output
Return only the summary as plain text, with one sentence per line. Do not include any
  ↔ explanations, bullet points, or extra commentary.

Structure
1. Overview (1 sentence): A short, high-level statement of what happens to {variable}.
2. Details:
   - Each sentence must describe discrete action or functionality on {variable}.
   - Annotate exactly one cell per sentence using the syntax:
     '{<phrase>}[cell N]' with N the associated cell id.
   - Use the first relevant cell id number if multiple apply.
   - Keep sentences concise and strictly factual (no this notebook explores).
   - Use contractions like "It's" instead of "It is".

Example
This variable is a dataframe describing customer churn rates.

```

An `{"initial exploratory analysis"}[cell 2]` of the customer's spending patterns and corresponding `↔ segments`.

The data undergoes `{"log transformation of numeric features"}[cell 8]`.

This is followed by `{"one-hot encoding of categorical variables"}[cell 9]`.

A `{"random forest classifier"}[cell 15]` identifies key predictive features.

These inform feature selection for the final `{"XGBoost model"}[cell 18]`.

Write the summary for variable `{variable}`:

Notebook code:

```
{codeCells
  .map(
    (cell, i) =>
      `Block ${cell.id}: \n${cell.source.join('\n')}`
  )
  .join('\n\n')}
```

Listing A.2: Full GPT-4o prompt used for generating the variable textual summaries.

References

- [1] K. Wongsuphasawat, Y. Liu, and J. Heer. *Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study*. <https://arxiv.org/abs/1911.00568>. 2019.
- [2] M. Li Vigni, C. Durante, and M. Cocchi. *Exploratory Data Analysis*. 1st. Elsevier, 2013. DOI: [10.1016/B978-0-444-59528-7.00003-X](https://doi.org/10.1016/B978-0-444-59528-7.00003-X).
- [3] M. B. Kery, M. Radensky, M. Mayank, B. E. John, and B. A. Myers. “The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool”. In: *CHI '18: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, pp. 1–11. DOI: [10.1145/3173574.3173748](https://doi.org/10.1145/3173574.3173748).
- [4] M. B. Kery and B. A. Myers. “Interactions for Untangling Messy History in a Computational Notebook”. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 147–155. DOI: [10.1109/VLHCC.2018.8506576](https://doi.org/10.1109/VLHCC.2018.8506576).
- [5] A. C. Rule. “Design and Use of Computational Notebooks”. Ph.D. dissertation. University of California, San Diego, 2018.
- [6] S. Chattopadhyay, Z. Feng, E. Arteaga, A. Au, G. Ramos, T. Barik, and A. Sarma. *Make It Make Sense! Understanding and Facilitating Sensemaking in Computational Notebooks*. <https://arxiv.org/abs/2312.11431>. 2023.
- [7] L. Battle and J. Heer. “Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau”. In: *Eurographics Conference on*

- Visualization (EuroVis)*. Eurographics, 2019. DOI: [10.1111/cgf.13678](https://doi.org/10.1111/cgf.13678). URL: <https://doi.org/10.1111/cgf.13678>.
- [8] D. Koop and J. Patel. “Dataflow Notebooks: Encoding and Tracking Dependencies of Cells”. In: *TaPP’17: Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*. USENIX, 2017, p. 17. DOI: [10.5555/3183865.3183888](https://dl.acm.org/doi/10.5555/3183865.3183888). URL: <https://dl.acm.org/doi/10.5555/3183865.3183888>.
- [9] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. “Vis-Trails: Visualization Meets Data Management”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD ’06)*. 2006, pp. 745–747. DOI: [10.1145/1142473.1142574](https://dl.acm.org/doi/10.1145/1142473.1142574). URL: <https://dl.acm.org/doi/10.1145/1142473.1142574>.
- [10] X. Pu, S. Kross, J. M. Hofman, and D. G. Goldstein. “Datamations: Animated Explanations of Data Analysis Pipelines”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI ’21)*. ACM, 2021, pp. 1–14. DOI: [10.1145/3411764.3445063](https://dl.acm.org/doi/pdf/10.1145/3411764.3445063). URL: <https://dl.acm.org/doi/pdf/10.1145/3411764.3445063>.
- [11] D. Ramasamy, C. Sarasua, A. Bacchelli, and A. Bernstein. *Visualising Data Science Workflows to Support Third-Party Notebook Comprehension: An Empirical Study*. <https://link.springer.com/article/10.1007/s10664-023-10289-9>. 2023.
- [12] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg. “Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 1–12. DOI: [10.1109/TVCG.2017.2744878](https://doi.org/10.1109/TVCG.2017.2744878).
- [13] K. Eckelt, K. Gadhane, A. Lex, and M. Streit. “Loops: Leveraging Provenance and Visualization to Support Exploratory Data Analysis in Notebooks”. In: *IEEE Transactions on Visualization and Computer Graphics* 31.1 (2025), pp. 1213–1223. DOI: [10.1109/tvcg.2024.3456186](https://ieeexplore.ieee.org/document/10689475/). URL: <https://ieeexplore.ieee.org/document/10689475/>.

- [14] Y. Liu, T. Althoff, and J. Heer. “Paths Explored, Paths Omitted, Paths Obscured: Decision Points & Selective Reporting in End-to-End Data Analysis”. In: *CHI '20: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020. DOI: [10.1145/3313831.3376533](https://doi.org/10.1145/3313831.3376533). URL: <https://dl.acm.org/doi/abs/10.1145/3313831.3376533>.
- [15] Y. Liu, A. Kale, T. Althoff, and J. Heer. “Boba: Authoring and Visualizing Multiverse Analyses”. In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (2021), pp. 1753–1763. DOI: [10.1109/TVCG.2020.3028985](https://doi.org/10.1109/TVCG.2020.3028985). URL: <https://ieeexplore.ieee.org/document/9216579>.
- [16] A. Rule, I. Drosos, A. Tabard, and J. D. Hollan. “Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding”. In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (2018). DOI: [10.1145/3274419](https://doi.org/10.1145/3274419). URL: <https://doi.org/10.1145/3274419>.
- [17] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik. “What’s Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities”. In: *CHI '20: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 2020, pp. 1–12. DOI: [10.1145/3313831.3376729](https://doi.org/10.1145/3313831.3376729). URL: <https://doi.org/10.1145/3313831.3376729>.
- [18] W. Epperson, V. Gorantla, D. Moritz, and A. Perer. “Dead or Alive: Continuous Data Profiling for Interactive Data Science”. In: *IEEE Transactions on Visualization and Computer Graphics* 30.1 (2021), pp. 197–207. DOI: [10.1109/TVCG.2023.3327367](https://doi.org/10.1109/TVCG.2023.3327367). URL: <https://dl.acm.org/doi/10.1109/TVCG.2023.3327367>.
- [19] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine. “Managing Messes in Computational Notebooks”. In: *CHI '19: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 2019, pp. 1–12. DOI: [10.1145/3290605.3300500](https://doi.org/10.1145/3290605.3300500). URL: <https://dl.acm.org/doi/10.1145/3290605.3300500>.

- [20] Y. Wu, J. M. Hellerstein, and A. Satyanarayan. “B2: Bridging Code and Interactive Visualization in Computational Notebooks”. In: *UIST '20: Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, 2020, pp. 152–165. DOI: [10.1145/3379337.3415851](https://doi.org/10.1145/3379337.3415851). URL: <https://dl.acm.org/doi/10.1145/3379337.3415851>.
- [21] K. Gadhane, Z. Cutler, and A. Lex. “Persist: Persistent and Reusable Interactions in Computational Notebooks”. In: *Computer Graphics Forum (EuroVis 2024)* 43.3 (2024). DOI: [10.1111/cgf.15092](https://doi.org/10.1111/cgf.15092). URL: <https://doi.org/10.1111/cgf.15092>.
- [22] A. Wang, D. Wang, X. Liu, and L. Wu. “Graph-Augmented Code Summarization in Computational Notebooks”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21) Demonstrations Track*. 2021. DOI: [10.24963/ijcai.2021/717](https://www.ijcai.org/proceedings/2021/0717.pdf). URL: <https://www.ijcai.org/proceedings/2021/0717.pdf>.
- [23] Y. Ouyang, L. Shen, Y. Wang, and Q. Li. “NotePlayer: Engaging Computational Notebooks for Dynamic Presentation of Analytical Processes”. In: *UIST '24: Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* 9 (2024), pp. 1–20. DOI: [10.1145/3654777.3676410](https://doi.org/10.1145/3654777.3676410). URL: <https://doi.org/10.1145/3654777.3676410>.
- [24] D. Kang, T. Ho, N. Marquardt, B. Mutlu, and A. Bianchi. “ToonNote: Improving Communication in Computational Notebooks Using Interactive Data Comics”. In: *CHI '21: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, 2021, 727:1–727:14. DOI: [10.1145/3411764.3445434](https://doi.org/10.1145/3411764.3445434). URL: <https://doi.org/10.1145/3411764.3445434>.
- [25] H. Li, L. Ying, H. Zhang, Y. Wu, H. Qu, and Y. Wang. “Notable: On-the-fly Assistant for Data Storytelling in Computational Notebooks”. In: *CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, 2023, 173:1–173:16. DOI: [10.1145/3544548.3580965](https://doi.org/10.1145/3544548.3580965). URL: <https://doi.org/10.1145/3544548.3580965>.

- [26] A. Y. Wang, Z. Wu, C. Brooks, and S. Oney. “Callisto: Capturing the “Why” by Connecting Conversations with Computational Narratives”. In: *CHI '20: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 2020, pp. 1–13. DOI: [10.1145/3313831.3376740](https://doi.org/10.1145/3313831.3376740). URL: <https://doi.org/10.1145/3313831.3376740>.
- [27] Y. Lin, L. Yang, H. Li, H. Qu, and D. Moritz. *InterLink: Linking Text with Code and Output in Computational Notebooks*. <https://arxiv.org/abs/2502.16114>. 2025.
- [28] A. Rule, A. Tabard, and J. D. Hollan. “Exploration and Explanation in Computational Notebooks”. In: *CHI '18: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, 32:1–32:12. DOI: [10.1145/3173574.3173606](https://doi.org/10.1145/3173574.3173606). URL: <https://doi.org/10.1145/3173574.3173606>.
- [29] J. P. Inala, C. Wang, S. Drucker, G. Ramos, V. Dibia, N. Riche, D. Brown, D. Marshall, and J. Gao. *Data Analysis in the Era of Generative AI*. <https://arxiv.org/abs/2409.18475>. 2024.
- [30] D. Wang, J. D. Weisz, M. Muller, P. Ram, W. Geyer, C. Dugan, Y. Tausczik, H. Samulowitz, and A. Gray. “Human–AI Collaboration in Data Science: Exploring Data Scientists’ Perceptions of Automated AI”. In: *Proceedings of the ACM on Human–Computer Interaction* 3.CSCW (2019), pp. 1–24. DOI: [10.1145/3359313](https://doi.org/10.1145/3359313). URL: <https://dl.acm.org/doi/10.1145/3359313>.
- [31] L. Xie, C. Zheng, H. Xia, H. Qu, and Z. Chen. “WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization”. In: *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. ACM, 2024, 119:1–119:14. DOI: [10.1145/3654777.3676374](https://doi.org/10.1145/3654777.3676374). URL: <https://doi.org/10.1145/3654777.3676374>.
- [32] W. Willett, J. Heer, and M. Agrawala. “Scented Widgets: Improving Navigation Cues with Embedded Visualizations”. In: *IEEE Transactions on Visualization and Computer*

- Graphics* 13.6 (2007), pp. 1129–1136. DOI: [10.1109/TVCG.2007.70589](https://doi.org/10.1109/TVCG.2007.70589). URL: <https://doi.org/10.1109/TVCG.2007.70589>.
- [33] L. Studtmann, S. Aydin, and H. Lichter. “Histree: A Tree-Based Experiment History Tracking Tool for Jupyter Notebooks”. In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)* (2023). DOI: [10.1109/APSEC60848.2023.00040](https://doi.org/10.1109/APSEC60848.2023.00040). URL: <https://ieeexplore.ieee.org/document/10479422>.
- [34] J. Wagemans, J. H. Elder, M. Kubovy, S. E. Palmer, M. A. Peterson, M. Singh, and R. von der Heydt. “A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure-ground organization”. In: *Psychol Bull* 138.6 (Nov. 2012), pp. 1172–1217.
- [35] D. A. Norman and S. W. Draper. *User-Centered System Design: New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, 1986. ISBN: 0-89859-781-1.
- [36] G. A. Miller. “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information”. In: *Psychological Review* 63.2 (1956), pp. 81–97. DOI: [10.1037/h0043158](https://doi.org/10.1037/h0043158). URL: <https://psycnet.apa.org/record/1957-02914-001>.
- [37] Microsoft. *VS Code Hover*. <https://code.visualstudio.com/api/references/vscode-api#Hover>.
- [38] N. Shrestha, T. Barik, and C. Parnin. “Unravel: A Fluent Code Explorer for Data Wrangling”. In: *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. ACM, 2021, pp. 198–207. DOI: [10.1145/3472749.3474744](https://doi.org/10.1145/3472749.3474744). URL: <https://doi.org/10.1145/3472749.3474744>.
- [39] E. L. Hutchins, J. D. Hollan, and D. A. Norman. “Direct Manipulation Interfaces”. In: *Human-Computer Interaction* 1 (1985), pp. 311–338.
- [40] R. A. Becker and W. S. Cleveland. “Brushing Scatterplots”. In: *Technometrics* 29.2 (1987), pp. 127–142.

- [41] D. Wootton, A. R. Fox, E. Peck, and A. Satyanarayan. “Charting EDA: Characterizing Interactive Visualization Use in Computational Notebooks with a Mixed-Methods Formalism”. In: *IEEE Transactions on Visualization and Computer Graphics* 31.1 (2025), pp. 1191–1201. DOI: [10.1109/TVCG.2024.3456217](https://doi.org/10.1109/TVCG.2024.3456217). URL: <https://doi.org/10.1109/TVCG.2024.3456217>.
- [42] thomkir1. *EDA.ipynb: Exploratory Data Analysis for Topic Modelling of Historical Corpus Comparison*. <https://labs.onb.ac.at/gitlab/thomkir1/topic-modelling-for-historical-corpus-comparison/-/blob/acefee6778963123bfa0c417742ec9885734fc31/notebooks/EDA.ipynb>. 2025.