LLM-ASSISTED ONBOARDING VIA RETRIEVAL-AUGMENTED INTERACTIVE COMPUTATIONAL NOTEBOOKS

by
BERKE ODACI

Submitted to the Graduate School of Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Computer Science

Sabancı University
June 2025

ABSTRACT

LLM-ASSISTED ONBOARDING VIA RETRIEVAL-AUGMENTED INTERACTIVE COMPUTATIONAL NOTEBOOKS

BERKE ODACI

COMPUTER SCIENCE & ENGINEERING M.Sc. THESIS, MAY 2025

Thesis Supervisor: Prof. SELIM BALCISOY

Keywords: Large Language Models, Interactive Computational Notebooks, Onboarding Support, Code Comprehension, Visual Analytics

Recent advancements in large language models (LLMs) have significantly improved their ability to understand programming workflows and generate functional code. While these models are widely used for code-related tasks such as generation and completion, they often fall short in providing sufficient explanation or contextual understanding, both of which are essential for effectively working with existing projects.

This challenge is particularly evident in Visual Analytics workflows, where interactive computational notebooks (e.g., Jupyter Notebooks) are commonly used to prototype and document complex visualizations, data transformations, and machine learning pipelines. These notebooks are accessed not only by developers but also by domain experts such as economists, analysts, or researchers who interact with the outputs, interpret the findings, or request changes. For both groups, onboarding into an unfamiliar project can be time-consuming and error-prone due to missing documentation, implicit logic, and the complexity of the code-output relationship.

To address this, we present a tool that supports the onboarding process by leveraging LLMs to analyze, explain, and edit interactive computational notebooks. The system parses the notebook into a directed graph of cells, generates natural language explanations for each cell, and stores them in a retrieval-augmented vector store. Users interact with the notebook through a web-based interface, where they can ask natural language questions, select specific cells for focused explanations, and even request code modifications, all with the ability to revert changes if needed.

We evaluate the tool with both software developers and domain experts through a mixed-method study, including task-based interactions and post-task surveys. Results show that the tool improves users' understanding of unfamiliar notebooks, increases their confidence in continuing the project, and is highly valued as a future onboarding aid. The tool demonstrates the potential of LLMs to bridge the gap between code and interpretation in data-driven environments, supporting more efficient collaboration and knowledge transfer across roles.

ÖZET

BÜYÜK DİL MODELLERİ İLE DESTEKLENEN ETKİLEŞİMLİ HESAPLAMA DEFTERLERİNDE PROJEYE UYUM SÜRECİ

BERKE ODACI

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, MAYIS 2025

Tez Danışmanı: Prof. Dr. SELIM BALCISOY

Anahtar Kelimeler: Büyük Dil Modelleri, Etkileşimli Hesaplama Defterleri, Uyum Süreci , Kod Anlayışı, Görsel Analitik

Özet

Büyük dil modelleri (LLM'ler), son yıllarda programlama süreçlerini anlamada ve işlevsel kod üretmede önemli ilerlemeler kaydetmiştir. Kod üretimi ve tamamlama gibi görevlerde yaygın olarak kullanılan bu modeller, çoğu zaman yeterli açıklama veya bağlamsal bilgi sunma konusunda yetersiz kalmaktadır. Oysa mevcut projelerle etkili bir şekilde çalışabilmek için hem geliştiriciler hem de konu uzmanları açısından bu tür açıklayıcı destek büyük önem taşımaktadır.

Bu ihtiyaç özellikle, veri görselleştirme ve analiz süreçlerinin yoğun olarak yürütüldüğü Görsel Analiz alanında kendini göstermektedir. Bu alanda, interaktif hesaplama defterleri (örneğin Jupyter Notebook) genellikle karmaşık görselleştirmeler, veri dönüşümleri ve makine öğrenmesi modellerinin prototiplenmesi için kullanılmaktadır. Söz konusu defterler yalnızca geliştiriciler değil; aynı zamanda çıktılarla doğrudan etkileşime giren, sonuçları yorumlayan veya değişiklik talep eden ekonomi uzmanları, veri analistleri ya da araştırmacılar gibi alan uzmanları tarafından da kullanılmaktadır. Yetersiz dokümantasyon ve örtük mantık

nedeniyle, bu tür defterlere adapte olmak her iki grup için de zaman alıcı ve hata yapmaya açık olabilir.

Bu çalışmada, projeye uyum sürecini kolaylaştırmak amacıyla, LLM'lerden yararlanarak interaktif hesaplama defterlerini analiz eden, açıklayan ve düzenleyebilen bir araç sunuyoruz. Sistem, defteri yönlü bir hücre grafiğine ayırmakta, her hücre için doğal dilde açıklamalar üretmekte ve bu içerikleri vektör tabanlı bir bilgi tabanında saklamaktadır. Kullanıcılar, web tabanlı bir arayüz üzerinden doğal dilde sorular sorabilir, belirli hücreleri seçerek odaklı açıklamalar alabilir ve kod üzerinde değişiklik talebinde bulunabilir; ayrıca yapılan değişiklikler gerekirse geri alınabilir.

Sistemi, yazılım geliştiriciler ve alan uzmanlarıyla gerçekleştirdiğimiz karma yöntemli bir değerlendirme ile test ettik. Görev tabanlı etkileşimler ve ardından yapılan anketler aracılığıyla elde edilen sonuçlar, aracın kullanıcıların defteri anlamasını kolaylaştırdığını, projeye devam etme konusundaki güvenlerini artırdığını ve gelecekte yeniden kullanılmak isteneceğini ortaya koymuştur. Bu çalışma, LLM'lerin kod ve yorum arasında köprü kurarak daha etkili işbirliği ve bilgi aktarımı sağlama potansiyelini ortaya koymaktadır.

ACKNOWLEDGEMENTS

It is a genuine pleasure to express my deepest gratitude to my mentor and thesis supervisor, Prof. Selim Saffet Balcisoy, for his continuous support, patience, and guidance throughout the past five years. His insights and encouragement have been invaluable to both my academic and personal development.

I would also like to sincerely thank all my friends and participants who contributed to the user studies and supported my research. Special thanks go to the members of the Behavioral Analytics & Visualization Lab (BAVLAB) for providing a collaborative and inspiring environment throughout this journey.

Finally, I am profoundly grateful to my family for their unwavering support, encouragement, and belief in me throughout my entire life. This thesis would not have been possible without them.

To my beloved family...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

In data-driven research and development environments, the ability to effectively onboard new team members is essential to sustaining productivity and collaboration. Onboarding typically involves understanding existing workflows, data pipelines, and decisions embedded in code artifacts. Because data science is often described as a multidisciplinary "team sport" (Kim, Zimmermann, DeLine & Begel, 2016; Wang, Mittal, Brooks & Oney, 2019), these artifacts are rarely the product of a single individual. Instead, they emerge from collaboration between data scientists, engineers, domain experts, and operations staff, each bringing different expertise, assumptions, and coding practices. Muller, Lange, Wang, Piorkowski, Tsay, Liao, Dugan & Erickson capture this diversity through the term "data science worker," reflecting the wide range of roles and responsibilities in modern data projects (Muller et al., 2019). While this diversity strengthens analytical capability, it also introduces barriers to shared understanding. Onboarding becomes especially difficult when contributors must interpret unfamiliar logic, undocumented decisions, or project-specific conventions without access to the original author's rationale.

In many modern projects, these artifacts are captured within interactive computational notebooks, which serve as dynamic, executable records of analysis. These notebooks often combine data preprocessing, modeling, visualization, and interpretation in a single interface (Rule, Tabard & Hollan, 2018). However, each project also introduces its own set of tools, conventions, and structural inconsistencies in the code, which can make onboarding overwhelming for newcomers, especially when no systematic support is provided Dagenais, Ossher, Bellamy, Robillard & de Vries (2010); Ju, Sajnani, Kelly & Herzig (2021); Matturro, Barrella & Benitez (2017). Despite their flexibility and expressiveness, such notebooks are frequently undocumented, inconsistently structured, or difficult to interpret without prior exposure to the author's reasoning (Pimentel, Murta, Braganholo & Freire, 2019).

This challenge is particularly pronounced in the context of Visual Analytics, where interactive computational notebooks are used not only to run code but also to prototype and communicate rich data visualizations, analytic workflows, and even

domain-specific insights. As notebook content grows in complexity through the layering of code, outputs, visual components, and interactive logic, the difficulty of understanding these artifacts increases. Wang, Wang, Drozdal, Muller, Park, Weisz, Liu, Wu & Dugan demonstrate that notebook users often neglect documentation during rapid iteration, leading to narrative gaps that can hinder comprehension and collaboration Wang et al. (2022). These challenges affect both developers, who must understand and adapt code structures, and domain experts, who focus on interpreting outputs, validating results, or suggesting modifications. In both cases, onboarding into an unfamiliar notebook-based project becomes a time-consuming and error-prone process. Prior work suggests that developers spend as much as 58% of their total time on program comprehension activities (Xia, Bao, Lo, Xing, Hassan & Li, 2018), which, while not limited to onboarding, highlights how significant understanding existing code is to overall developer effort.

Recent advancements in large language models (LLMs) have introduced new possibilities for assisting with code-centric tasks. Models such as CodeBERT (Feng, Guo, Tang, Duan, Feng, Gong, Shou, Qin, Liu, Jiang & Zhou, 2020), CodeT5 (Wang, Wang, Joty & Hoi, 2021), and Code Llama (et al., 2024b), as well as general-purpose models like GPT-4 (OpenAI, 2023), have demonstrated impressive capabilities in code generation, summarization, and completion. However, many of these applications focus primarily on generating new code, often without sufficient explanation or contextualization. This limits their usefulness in onboarding scenarios where the primary goal is not to generate code, but to understand, validate, and modify existing workflows. Recent studies Leinonen, Denny, MacNeil, Sarsa, Bernstein, Kim, Tran & Hellas (2023); Nam, Macvean, Hellendoorn, Vasilescu & Myers (2024a) have shown that LLMs can aid comprehension by producing contextual explanations. These explanations help reduce cognitive load and improve understanding of unfamiliar codebases. This opens the possibility of using LLMs not only for code generation but also for supporting human reasoning and exploration.

In this thesis, I present an interactive tool designed to support the onboarding process for both developers and domain experts working with interactive computational notebooks in Visual Analytics projects. The tool leverages the capabilities of LLMs for explanation, context-aware question answering, and code editing. It operates by parsing a notebook into a directed graph of cells, extracting logical dependencies, and generating natural language explanations for each component. These are stored in a retrieval-augmented knowledge base that enables efficient semantic search and interaction. Users engage with the system through a chat interface, where they can ask questions, request modifications to code, or seek clarification on outputs. A revert mechanism is included to maintain safety and user control during editing

operations.

The system is implemented using GPT-4 and the LangChain framework (Chase, 2022; OpenAI, 2023), and is presented through a web-based interface that integrates code execution, chat-based interaction, and editable notebook content. A demonstration video showcasing the tool's functionality is provided as supplementary material.

To evaluate the effectiveness of this approach, I conduct a user study involving two participant groups: professional developers and domain experts. Each participant interacts with real-world interactive computational notebooks, both with and without the assistance of the tool. Their experiences are assessed through structured tasks and post-task questionnaires. The goal of this study is to measure improvements in comprehension, confidence, and task efficiency enabled by the tool.

The main contributions of this thesis are summarized as follows:

- A novel tool has been developed to support the onboarding process in Visual Analytics workflows by leveraging large language models (LLMs) to explain, summarize, and modify code in interactive computational notebooks.
- The tool is designed to assist both developers and domain experts during onboarding. In data science projects, collaboration often occurs between individuals with different areas of expertise, for example, software engineers and subject matter experts in finance, economics, or healthcare. This disparity in technical and domain-specific knowledge makes onboarding more difficult, as each side may lack full visibility into the other's work.
- By enabling users to ask natural language questions and receive contextualized explanations about the code, data, outputs, and visualizations, the tool helps bridge this gap. It allows users to understand and explore unfamiliar projects more efficiently, facilitating collaboration and reducing reliance on direct developer support.
- In addition to improving mutual understanding, the tool empowers domain experts to take a more active role in their workflows. Instead of waiting for developer intervention, experts can use the tool to modify notebook logic, update parameters, or request code-level changes through an LLM-assisted interface, accelerating the iteration cycle.
- A user study with developers and domain experts validates the usefulness of the system, showing that it improves comprehension, increases confidence, and shortens onboarding time when compared to working with notebooks alone.

Through these contributions, this thesis explores the practical potential of large language models in facilitating onboarding within Visual Analytics projects. It presents a system that helps users interpret code, data, and visual outputs more effectively and supports collaboration between developers and domain experts by enabling accessible, LLM-assisted interaction with notebook-based workflows.

The remainder of this thesis is organized as follows:

Chapter 2 provides a comprehensive review of the relevant literature. This includes prior work on onboarding challenges in software and data science workflows, as well as research on the use of large language models (LLMs) for code generation, editing, summarization, and explanation.

Chapter 3 describes the architecture and core components of the proposed tool. It covers the overall workflow, the graph-based structure used for cell context retrieval, and how LLM prompting is used to generate code, explanations, and summaries in response to user actions.

Chapter 4 outlines the design of the user study. It details the participant groups, notebook selection process, experimental setup, and questionnaire design used to evaluate the tool's onboarding support capabilities.

Chapter 5 presents the results and analysis of the user study. It includes task-specific outcomes, compares coder and field expert experiences, and discusses the implications of the findings in relation to tool usage and user expectations.

Chapter 6 discusses the limitations of the current system and outlines future directions for extending functionality, improving usability, and supporting broader deployment in real-world notebook environments.

Chapter 7 concludes the thesis by summarizing the main contributions and reflecting on the broader significance of using LLMs to improve onboarding in computational notebooks.

2. Related Work

Understanding existing code and analytic workflows remains a core challenge in both data science and software engineering Dong, Lou, Zhu, Sun, Li, Zhang & Hao (2022); Hoang, Kang, Lo & Lawall (2020); Li, Xu, Di, Wang, Li & Zheng (2024); Liu, Tang, Xia & Yang (2023). This issue becomes particularly acute in *Visual Analytics* settings, where interactive computational notebooks, such as Jupyter Notebooks, serve not only as execution environments but also as documentation and storytelling tools. These notebooks intertwine code, narrative text, and visual output into a single medium, which, while excellent for exploratory analysis and communication, often lacks structure and consistency. Consequently, they pose unique barriers for onboarding new contributors who must comprehend previous workflows, trace analytical reasoning, and safely modify existing content.

Unlike traditional software repositories, interactive notebooks are frequently written informally, without rigorous commenting or modularity Wang, Li & Zeller (2020). Execution order may not be linear, variable reuse is common, and cells may depend on invisible context from earlier executions Wang, Kuo, Li & Zeller (2021). These characteristics diminish the effectiveness of conventional software engineering onboarding strategies and demand new methods tailored to this hybrid format. Both developers and domain experts, such as data scientists, economists, or biomedical researchers, face difficulties in interpreting the logic and intent embedded within these documents, especially when metadata, dependencies, and rationale are either implicit or scattered.

A substantial body of research has explored techniques for improving comprehension of unfamiliar codebases, particularly in the context of software maintenance, reverse engineering, and collaborative development environments LaToza, Venolia & DeLine (2006); Robillard, Walker & Zimmermann (2010). These efforts emphasize the value of mental model alignment, scaffolding, and context-aware tooling to aid developers in reasoning about existing systems. Additionally, onboarding support has been studied in the context of open-source and industrial projects where high turnover, distributed teams, and incomplete documentation further complicate in-

tegration Steinmacher, Treude & Gerosa (2019). These studies highlight the social and technical challenges faced by newcomers, including information overload, lack of mentorship, and the difficulty of navigating undocumented workflows.

In parallel, the emergence of large language models (LLMs) has introduced new paradigms for code-related assistance. Trained on massive corpora of code and natural language, LLMs such as CodeBERT Feng et al. (2020), CodeT5 Wang et al. (2021), and GPT-4 OpenAI (2023) have demonstrated strong performance in tasks including code generation, explanation, translation, and summarization. These models not only support productivity-enhancing features such as auto-completion and code synthesis, but also enable novel forms of interaction, such as answering questions about existing code or suggesting refactoring strategies. Their ability to bridge natural and programming languages makes them particularly well-suited for onboarding scenarios, where new users may seek intuitive explanations and context without combing through extensive documentation.

Despite growing adoption of LLM-based tools, most focus on general-purpose development environments and overlook the idiosyncrasies of notebook-based analytics. Notebooks pose specific challenges (e.g., non-linear execution, weak encapsulation, and interleaved output) but also offer opportunities for richer interaction, such as dialog-based walkthroughs or in-context annotation of cells. As a result, there is a pressing need to understand how LLMs can be leveraged to support onboarding not just for code, but for entire analytical narratives embedded in computational notebooks.

This chapter surveys the literature along two intersecting dimensions: onboarding strategies (Section 2.1), and the capabilities of LLMs in code-related tasks (Section 2.2). Section 2.1 focuses on the challenges of onboarding both developers and domain experts in data-intensive workflows, with special attention to informal, hybrid artifacts like notebooks. Section 2.2 is further divided into three parts: code generation (Section 2.2.1), code explanation (Section 2.2.2), and code summarization (Section 2.2.3), capabilities that are central to our proposed tool. Together, these perspectives inform the design and evaluation of intelligent onboarding assistants tailored for computational notebooks.

2.1 Onboarding

Onboarding in software and data science workflows involves enabling newcomers, whether developers or domain experts, to quickly understand, navigate, and contribute to existing code and analytical pipelines.

The challenge is especially pronounced in interactive computational notebooks (e.g., Jupyter). Unlike traditional software, notebooks intertwine code, commentary, and visual output in a non-linear and often undocumented manner. This structure increases the cognitive load, obscures dependencies, and hinders the ability of newcomers to build accurate mental models of workflow. Static documentation or linear tutorials are insufficient in such settings.

Recent visual analytics research has leveraged LLMs to improve onboarding for complex systems. A notable example is **LEVA** (LLM-Enhanced Visual Analytics), which employs large language models to interpret visual designs and relationships in a visual analytics system and generates interactive, context-aware tutorials to support onboarding, exploration, and summarization phases Zhao, Zhang, Zhang, Zhao, Wang, Shao, Turkay & Chen (2025). LEVA demonstrates that mixed-initiative guidance, combining system knowledge, visual design metadata, and natural-language tutoring, can effectively reduce users' learning curve and improve task performance.

Nevertheless, prior systems like LEVA focus on the visual analytics domain rather than notebook-based code workflows. Tools such as EDAssistant embed code search and API recommendation features inside notebooks, aiding comprehension but lacking explicit structure or context tracing Li, Zhang, Leung, Sun & Zhao (2023). Other efforts, such as *Albireo*, offer visual representations of code dependencies via force-directed graphs, where each node corresponds to a notebook cell and edges capture execution or data dependencies Wenskovitch, Zhao, Carter, Cooper & North (2019). This structure assists orientation by exposing non-linear cell flows and shared context, helping users navigate the implicit logic embedded in notebooks. However, while Albireo supports structural understanding, the burden of interpreting semantic meaning and analytical intent still largely falls on the user.

In contrast, our work augments these approaches by constructing a **directed cell-dependency graph** across notebook cells, combined with LLM-driven explanation and summarization. This hybrid structural-semantic approach surfaces both the relevant code context and its narrative meaning, supporting newcomers in building holistic mental models of exploratory notebook workflows.

2.2 Large Language Models

Large Language Models (LLMs) have emerged as powerful tools for natural language understanding and generation, with capabilities that extend to a wide range of programming-related tasks. Trained on massive corpora that include both code and natural language, models such as GPT-4 OpenAI (2023), CodeBERT Feng et al. (2020), and CodeT5 Wang et al. (2021) have demonstrated remarkable performance on code generation, explanation, and summarization benchmarks.

These models operate on the principle of masked language modeling or autoregressive prediction, enabling them to generate syntactically correct and semantically meaningful code snippets, translate between programming languages, summarize source files, and answer natural-language questions about code. Such capabilities make LLMs attractive not only for automation and productivity, but also for human-centered tasks like teaching, pair programming, and onboarding.

While the commercial integration of LLMs in tools such as GitHub Copilot has focused on enhancing developer productivity through auto-completion and synthesis, academic research is increasingly investigating their potential for deeper support in program comprehension. Particularly relevant is their ability to engage in multi-turn dialogue, provide context-aware feedback, and explain complex code behavior, features that are well-aligned with the needs of newcomers entering unfamiliar codebases or workflows.

In the context of this thesis, LLMs are leveraged to assist onboarding within interactive computational notebooks by supporting three complementary functions. First, they can generate code based on user input, either by editing an existing cell or creating a new one, facilitating iterative development and correction. Second, they provide explanations not only for code fragments, but also for the outputs produced by code execution, helping users interpret visualizations, statistical results, or intermediate variables. Third, they assist in summarizing the overall analytical workflow, including describing the structure and semantics of the datasets used in the project. These functionalities aim to reduce the cognitive load on newcomers and support a more intuitive and guided exploration of notebook-based analyses. Each of these capabilities is discussed in detail in the following subsections.

2.2.1 Code Generation by LLMs

Code generation remains one of the most prominent and well-researched capabilities of Large Language Models (LLMs). By transforming natural language descriptions

into executable code, LLMs have significantly lowered the barrier to programming and accelerated development workflows across a variety of platforms, including IDEs, version control systems, and interactive computational notebooks.

Model Advances: Early models such as CodeBERT Feng et al. (2020) and CodeT5 Wang et al. (2021) laid the groundwork for bidirectional understanding and generation tasks. More recent instruction-tuned models, including GPT-4 OpenAI (2023), LLaMA 3 et al. (2024a), and Code LLaMA et al. (2024b), demonstrate improved zero-shot and few-shot performance on natural language to code tasks across benchmarks like HumanEval, MBPP, and CodeXGLUE Lu, Guo, Ren, Huang, Svyatkovskiy, Blanco, Clement, Drain, Jiang, Tang, Li, Zhou, Shou, Zhou, Tufano, Gong, Zhou, Duan, Sundaresan, Deng, Fu & Liu (2021). Notably, Code LLaMA is specialized for code tasks, trained on permissively licensed source code, and supports multiple programming languages. LLaMA 3 continues this trajectory by integrating massive pretraining with strong instruction tuning, narrowing the gap between open and proprietary models.

Notebook-Oriented Generation: While many models are designed for general-purpose code generation, JuPyT5 Chandel, Clement, Serrato & Sundaresan (2022) specifically addresses generation in Jupyter notebook environments. It is trained on notebook-style data to produce coherent code cells that reflect real-world data science workflows, such as cell-based structure, context sensitivity, and informal code-comment mixing.

Evaluation Metrics: Performance is often measured using pass@k, semantic correctness, and functional equivalence. Liu et al. highlight the limitations of traditional evaluation approaches by introducing EvalPlus, a framework that generates comprehensive test suites to assess the robustness of LLM-generated code Liu, Xia, Wang & Zhang (2023). Their findings show that augmenting benchmarks with additional tests can significantly reduce pass rates, by up to 28.9%, revealing that many generations initially considered correct fail under stricter validation. This underscores the need for more rigorous, execution-based evaluation when deploying code generation models in practical settings.

Human-in-the-Loop Refinement: Despite impressive results, LLM-generated code can still exhibit logical errors, incomplete reasoning, or improper API usage. Interactive, multi-turn systems such as TiCoder Fakhoury, Naik, Sakkas, Chakraborty & Lahiri (2024) address these limitations by incorporating feedback loops, allowing users to iteratively refine and validate code with the model's assistance. This approach is particularly valuable in notebook environments, where code is often written incrementally and context evolves over time.

In summary, code generation by LLMs is a rapidly advancing field. For notebook-based onboarding, we leverage these advances to allow users to request new cell generation or cell editing via natural language. This interaction pattern, grounded in real-time context, supports incremental development and lowers the cognitive burden associated with writing boilerplate or exploring new libraries.

2.2.2 Code Explanation by LLMs

As analysts, developers, or domain experts engage with unfamiliar notebooks, understanding what code does and why is essential. Traditional forms of documentation, such as inline comments or docstrings are often incomplete, outdated, or entirely absent. LLMs offer a compelling alternative by generating natural-language explanations that clarify the intent, structure, and behavior of code.

Effectiveness in Educational Contexts: LLMs have been evaluated as explanation generators in academic settings, often outperforming novice-authored explanations in clarity and usefulness. Leinonen et al. compared ChatGPT-generated explanations with those written by students in introductory programming assignments and found that LLM-generated content was consistently rated as clearer and more accurate by external evaluators Leinonen et al. (2023).

Flexible Level of Detail: LLMs can generate explanations at different levels of detail, ranging from short summaries to step-by-step, line-by-line explanations, depending on what the user needs. For example, Xiao, Hou & Stamper found that beginners learned better when they were given multiple types of hints, such as general advice along with detailed code comments Xiao et al. (2024). Similarly, systems like LLM-powered e-books provide several kinds of explanations for the same code, including individual line descriptions, key concepts, and overall summaries MacNeil, Tran, Hellas, Kim, Sarsa, Denny, Bernstein & Leinonen (2023). This kind of flexibility is especially useful in computational notebooks, where users move between writing code, viewing results, and reading or writing explanatory text. Adapting the level of explanation to match this workflow helps make onboarding smoother and more intuitive.

Context-Aware Understanding: Modern interfaces that integrate LLMs, such as conversational IDE assistants, allow users to query the rationale behind specific implementation choices, identify causes of errors, or request alternative formulations. Nam, Macvean, Hellendoorn, Vasilescu & Myers demonstrate how LLMs can be

embedded in development environments to provide real-time, context-aware code understanding support Nam et al. (2024b).

In our system, LLMs are used not only to explain code syntax or logic, but also to interpret outputs, such as data tables, figures, or textual results, produced by notebook cells. This extends the explanatory capability from "what the code does" to "what the result means," enabling a more comprehensive onboarding experience.

2.2.3 Code Summarization by LLMs

Code summarization involves generating short, natural-language explanations of code segments, helping users quickly understand what the code does without inspecting each line. This feature is especially helpful in computational notebooks, where narrative and code coexist.

Statement-level summarization: Zhu, Miao, Xu, Zhu & Sun conduct a comprehensive evaluation of LLMs at the statement level and find that models like GPT-4 and CodeLlama outperform prior baselines, demonstrating that large models can reliably summarize even fine-grained code units under well-tuned prompting Zhu et al. (2024).

Structured hybrid context: Zhou, Li, Yu, Fan, Yang & Huang introduce StructCodeSum, a technique that combines local snippet context with global code structure (such as inter-function references and control/data flow) to produce more informative and coherent summaries Zhou et al. (2024).

Project-specific adaptation: Ahmed & Devanbu demonstrate that few-shot tuning on project-specific code examples significantly improves summarization quality, enabling better alignment with domain conventions and vocabulary Ahmed & Devanbu (2023).

Remaining challenges: Even with these advances, code summarization still suffers from issues like hallucinated details, misalignment with user priorities, and dependency on labeled examples. These problems are particularly acute in notebooks, where analysis is exploratory, data-driven, and often lacks formal structure.

Our system integrates code summarization at two scales: (a) notebook-level summaries that outline the overall workflow and data usage, and (b) grouped-cell summaries that describe the intents of related sections. These summaries leverage both

local snippet context and project-wide cues to help newcomers build clear mental models of notebook-driven analysis.

3. ONBOARDING SYSTEM

3.1 System Overview

Our system is designed to support onboarding in existing interactive computational notebooks, such as Jupyter notebooks, by leveraging large language models (LLMs) in a retrieval-augmented generation (RAG) framework Lewis, Perez, Piktus, Petroni, Karpukhin, Goyal, Küttler, Lewis, Yih, Rocktäschel, Riedel & Kiela (2020). The goal is to assist both developers and domain experts in understanding, modifying, and extending prior analyses with minimal friction.

The workflow begins with the system ingesting an already existing notebook. Optionally, the user may supply a separate context file, which can contain any additional background information, documentation, or domain-specific knowledge. While this context file can enhance LLM performance in some cases, it is not required for the system to function.

As part of the upload process, the system automatically sends each code cell to the LLM to generate a textual explanation or summary. These cell-level context snippets are embedded and stored in a vector database. This improves the quality of later retrieval operations by enriching the semantic search space, especially in cases where the user does not click a specific cell but instead asks general questions about the notebook. By ensuring that each cell has pre-generated, LLM-authored context, the system increases the likelihood of retrieving the correct portion of the notebook during RAG-based interaction.

To model the structure and flow of the notebook, the system constructs a directed graph where each node corresponds to a code cell. Edges represent semantic or execution dependencies, such as variable definitions, function usage, or import statements, across cells. This graph allows the system to retrieve relevant context when

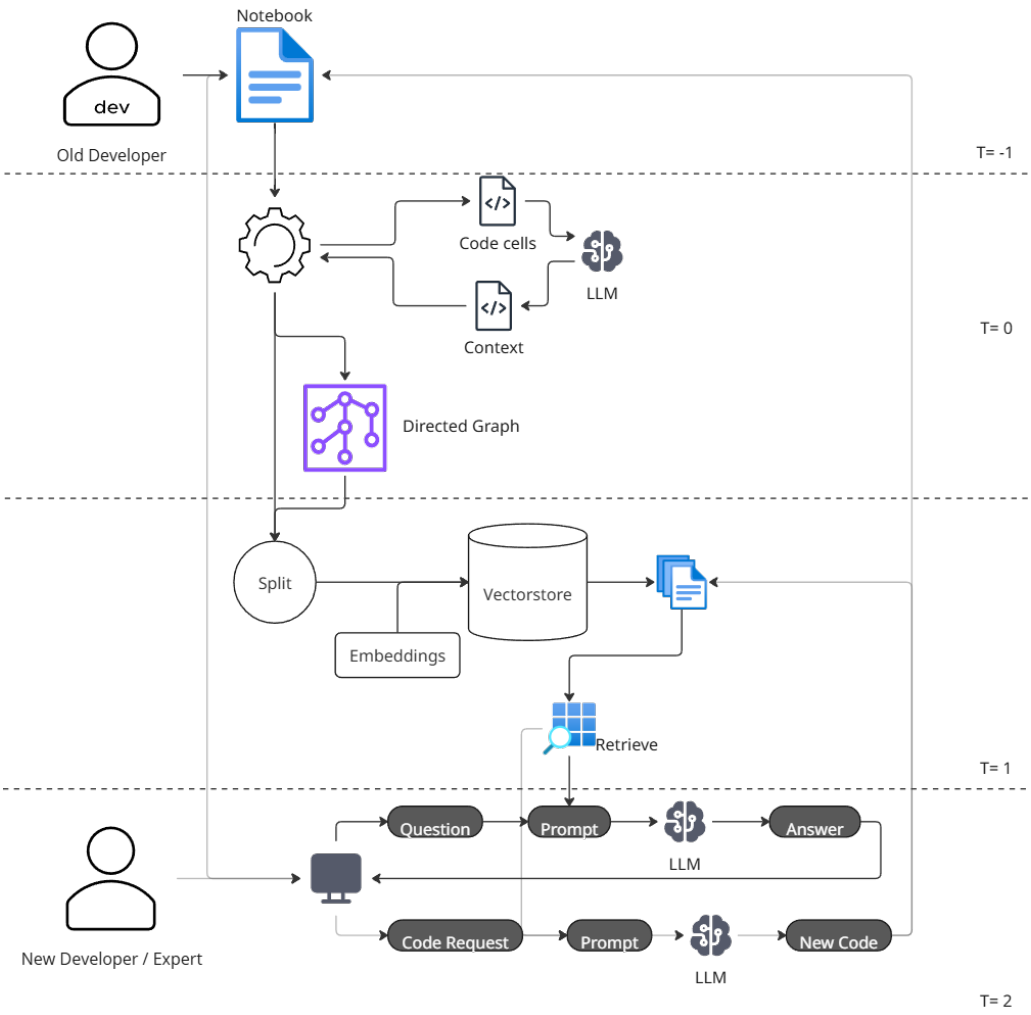

Figure 3.1 Overview of the system's workflow for LLM-assisted onboarding in computational notebooks.

user queries reference a specific part of the notebook.

The notebook is then segmented into smaller parts and indexed into a vector store. This enables the use of a RAG pipeline, where semantically relevant notebook fragments are retrieved to construct context-aware prompts for the LLM.

Once processing is complete, the interactive notebook is presented to the user. At this stage, the system supports several forms of interaction. Users can:

- Ask natural language questions about the notebook.
- Request code generation based on prompts.
- Click on specific cells to get targeted explanations or clarifications.

When a user selects a particular cell, the LLM responds using context specifically tailored to that cell. This functionality is especially useful for cells that contain either complex logic or difficult-to-interpret output (e.g., long tables or intricate plots).

From this point onward, the system becomes an active assistant during the onboarding process. Whether the user is a software developer or a domain expert, the LLM is capable of answering questions, proposing code modifications, and offering explanations across a wide range of technical and analytical concerns.

In addition to LLM-based assistance, users retain full interactivity with the notebook interface. They can edit text and code, insert new cells, execute computations, and explore outputs freely. If a user is dissatisfied with a code modification made by the LLM, they may undo the change on a per-cell basis via a dedicated "redo" functionality, preserving control and reversibility during experimentation.

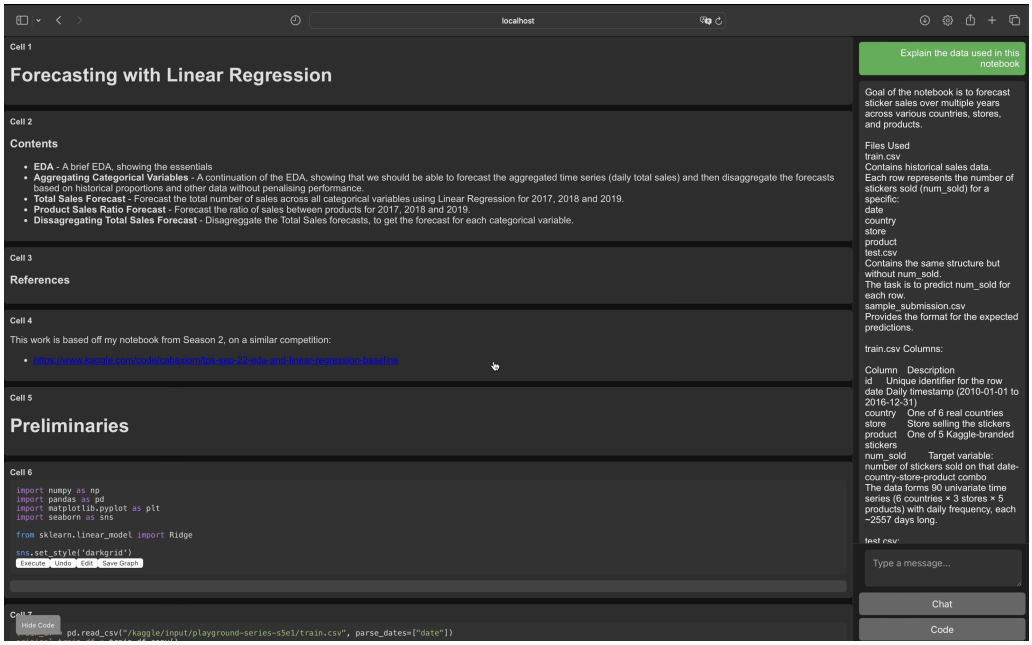

Figure 3.2 Front-end interface of the onboarding assistant. Users can interact with the notebook, query the LLM, and view context-aware responses.

3.2 Dependency Graph Construction

To capture the flow of logic and dependencies in a computational notebook, we construct a directed graph $G = (V, E)$, where each node $v_i \in V$ represents a code cell in the notebook, and each directed edge $(v_i, v_j) \in E$ indicates that cell v_j depends on cell v_i.

Unlike linear scripts, interactive notebooks such as Jupyter support non-linear execution, frequent redefinition of variables, and fragmented logic across cells. As such, inferring context requires a structural representation that captures both implicit and explicit dependencies between cells.

Graph Nodes: Each code cell in the notebook is assigned a unique index based on its position in the document. Let $C = [c_1, c_2, \ldots, c_n]$ denote the ordered list of code cells. Each cell c_i corresponds to a node $v_i \in V$ in the graph.

Edge Construction Rules: We define the edge set E using the following rules:

1.1 **Sequential Flow Rule:** For all $i \in \{1, \ldots, n-1\}$, we add an edge (v_i, v_{i+1}). This captures the default sequential flow of the notebook as written.

1.2 **Symbol Dependency Rule:** Let $\mathcal{D}_i$ denote the set of symbols (e.g., functions, classes, variables) defined in cell c_i, and let $\mathcal{U}_j$ denote the set of symbols used in cell c_j. For all $i, j \in \{1, \ldots, n\}$, if $\mathcal{D}_i \cap \mathcal{U}_j \neq \emptyset$, we add a directed edge (v_i, v_j). This ensures that semantic dependencies (e.g., a function defined in one cell and used in another) are captured explicitly.

The resulting graph may contain multiple edges between non-adjacent cells and captures both temporal and logical relationships.

Symbol Extraction and Analysis: To identify $\mathcal{D}_i$ and $\mathcal{U}_j$, we statically analyze each code cell by parsing its abstract syntax tree (AST). For each cell:

- Definitions ($\mathcal{D}_i$) include function names, class names, and top-level variable assignments.
- Usages ($\mathcal{U}_j$) include any referenced identifiers not locally defined within the same cell.

We maintain a global symbol table indexed by cell, enabling efficient lookup of where each symbol was first defined. When a symbol is defined in multiple cells, we treat the most recent definition, based on the topological sort of the graph, as authoritative. This is implemented by traversing the graph in topological order and updating the symbol table whenever a redefinition is encountered. This way, symbol resolution always reflects the most recent version visible from earlier execution paths.

Use in Context Retrieval: When a user selects a specific cell c_k, the system queries the graph to retrieve all upstream cells that provide relevant context. This is done by computing the transitive closure in the graph to identify nodes from which there exists a directed path to v_k:

$$\text{Context}(c_k) = \{c_i \in C \mid \text{there exists a path } v_i \rightsquigarrow v_k \text{ in } G\}$$

The content of these cells is retrieved and used to construct an LLM prompt. This ensures that the model has access to all necessary symbol definitions and execution context when generating explanations or answering user queries for the selected cell.

Handling Non-Code Cells: While markdown or comment-only cells do not define or use executable symbols and are thus not connected via semantic edges in the directed graph, they are still included in the RAG system. This is because such cells may contain explanations, observations, or interpretations written by the original author, which are valuable for onboarding. These cells are embedded and indexed in the vector store alongside code cells, enabling them to be retrieved during semantic search if relevant to the user's query.

3.3 LLM Integration

The core of our system's intelligence is provided by a large language model (LLM), which supports onboarding tasks such as answering user questions, explaining code cells, and generating or editing code. To ensure relevance and accuracy, the LLM is not used in isolation; rather, it is integrated into a retrieval-augmented generation (RAG) framework and interacts closely with the cell dependency graph.

Prompt Construction: All interactions with the LLM involve the construction of a task-specific prompt, which includes:

- A user query or instruction (e.g., "What does this cell do?", "Fix this error", "Add a function to filter the dataset.")
- Context retrieved from either the dependency graph or vector store, and, if a cell is selected, the contents of that specific cell and its upstream dependencies.
- A system prompt that defines the model's role (e.g., "You are an assistant helping a user understand and edit this data science notebook.")

Context Retrieval Paths: There are two distinct pathways for retrieving context:

2.1 **Graph-based retrieval:** If the user has clicked on a specific cell, the system uses the dependency graph to gather all upstream cells that define symbols used in the selected cell. The content of the selected cell itself is also included to ensure local context is preserved.

2.2 **RAG-based retrieval:** For general questions not tied to a specific cell, relevant notebook fragments are retrieved from the vector store using dense embedding similarity to the user's query.

The retrieved context is inserted into the prompt in a structured format to give the LLM a complete and coherent view of the relevant portion of the notebook.

Prompt Format: Each LLM call uses a structured prompt composed of a system message, relevant notebook context, and the user query. If a user clicks on a specific cell before asking a question, the selected cell and its upstream dependencies (as determined by the dependency graph) are included in the context. If no cell is selected, context is retrieved via similarity search from the vector store.

A representative prompt structure looks like this: [System message] You are a helpful assistant for understanding and editing code in a data science notebook.

[Context] Cell 2 (Upstream) def load$_data$(...) : ...

Cell 5 (Selected) result = process(data)

[User question] What does the result variable represent, and why is process() used here?

This format ensures that the LLM is grounded in the relevant notebook context, whether the question pertains to a specific cell or a general workflow topic. It also enables follow-up interactions by preserving conversational coherence.

Prompt Design Process: Prompt formats used in the system were developed iteratively through empirical testing. During the tool's development, we continuously evaluated how the LLM responded to different prompt structures. When we observed incorrect, or vague outputs, we revised the system message, formatting, or context ordering to improve accuracy and relevance. This refinement process continued until we were satisfied with the tool's behavior in diverse scenarios. The user study was conducted using this final version of the prompt format, ensuring that results reflect the system's best-performing configuration.

LLM Model Selection: For this project, we selected the ChatGPT-4 API as the

LLM backend. Although models fine-tuned for specific tasks (e.g., code completion or summarization) can offer strong performance in narrow domains, our system requires a broader skill set. The LLM must not only reason about code, but also:

- Understand general user language, both technical and non-technical.
- Explain datasets, modeling steps, and analytical goals.
- Interpret narrative comments and intentions left by previous developers.
- Comprehend user requests with limited or ambiguous phrasing.

These requirements make a general-purpose, instruction-tuned model like ChatGPT-4 more suitable than narrower, code-only models. Its strong performance across both conversational and technical tasks allows it to support diverse users, including both programmers and domain experts, through natural, adaptive interactions.

Task Types: The LLM supports multiple user-facing operations:

- **Code Explanation:** Describing the logic, purpose, and output of a selected code cell.
- **Code Generation:** Producing new code based on a user instruction or modifying an existing cell.
- **Workflow Questions:** Answering high-level questions about the dataset, modeling choices, or analysis steps.

All prompts are passed to the model through an API interface that is decoupled from the notebook front end, allowing for easy future adaptation to alternative models.

3.4 User Interactions and Features

The system is designed to support intuitive, seamless interaction between users and the computational notebook environment. It accommodates both code-centric and non-technical users by enabling natural language interaction with the notebook and providing real-time assistance through LLM-powered features. The interface remains consistent with the traditional notebook paradigm while augmenting it with onboarding-focused capabilities.

Notebook Display and Interaction: Upon loading, the system presents the full notebook in an interactive web-based interface (see Figure 3.2). Users can read, scroll, and freely navigate across all cells, code and markdown alike. Cells retain full interactivity: users can modify code or text, insert new cells, delete or rearrange existing ones, and execute any part of the notebook just as they would in a standard Jupyter environment.

LLM Querying: At any point, users may enter natural language queries through a dedicated input field. The query can be general (e.g., "What does this notebook do?") or targeted (e.g., "Explain what this loop is doing."). If a cell is selected prior to asking a question, the system incorporates that cell and its upstream dependencies into the LLM prompt, enabling localized explanations.

Cell-Based Explanation: Clicking on a specific cell enables cell-focused assistance. In this mode, the LLM answers questions specifically in the context of that cell. This is particularly helpful when the selected cell contains:

- Complex control structures (e.g., nested loops or conditionals),
- Abstracted function calls,
- Output-heavy results such as data tables or visualizations.

This feature supports both technical and non-technical users in interpreting unfamiliar or opaque analysis logic.

Code Generation and Editing: Users may ask the LLM to modify existing cells or generate entirely new ones. For example, a prompt like "Add a function to normalize the values in this column" will result in either a modified cell or an inserted cell with the requested code. The system highlights LLM-generated changes and associates them with an undo/redo interface for traceability and control.

The system does not rely on precomputed or static explanations. Instead, all user questions are dynamically forwarded to the LLM at runtime. This decision was made to support a wide range of user query styles and natural language phrasing. Since different users (e.g., coders vs. field experts) may use different terminology, dynamically generating responses ensures the system adapts in real time to the intent behind each question. Every response is freshly generated by the LLM using current context retrieved from the notebook.

Redo Functionality: Every LLM-driven code edit is reversible. A "redo" button is associated with each cell that has been modified by the LLM, allowing users to revert changes easily. This functionality is implemented through local versioning

rather than by re-querying the LLM. Each time a cell is modified, its previous state is stored, enabling instant rollback without incurring additional latency or API cost. This is critical for iterative development and promotes experimentation without risk of losing the original state.

Manual Exploration: LLM assistance is optional. Users can fully explore, edit, and run the notebook on their own. The system does not interfere with manual workflows, allowing users to alternate between automated guidance and hands-on exploration as needed.

Mixed-Expertise Support: The design intentionally accommodates both software developers and field specialists (e.g., data analysts, researchers). By combining notebook-native interaction with LLM-based contextual support, the system reduces onboarding friction across a wide range of technical backgrounds.

4. USER STUDY

To evaluate the effectiveness of our LLM-assisted onboarding tool, we conducted a user study involving participants with varying technical backgrounds. The goal of the study was to assess how well the system supports users in understanding, navigating, and continuing work on existing computational notebooks, particularly those with complex workflows and limited documentation.

The study was designed around realistic notebook scenarios and structured user tasks, followed by a questionnaire to collect both quantitative ratings and qualitative feedback. Participants were exposed to different datasets and code environments, and we measured their perceptions of code clarity, data understanding, and overall usability, both with and without assistance from the tool.

In the following sections, we describe the test datasets used, the participant groups involved, and the structure of the evaluation experiment.

4.1 User Groups

To understand how the tool performs across different types of users, we divided study participants into two main groups based on their technical backgrounds:

Coders: This group consisted of participants with prior experience in programming, particularly in Python and working with Jupyter notebooks. These users were expected to be familiar with typical data science workflows, including data loading, transformation, modeling, and interpretation. They represent software developers, data scientists, and technically-oriented analysts who are likely to engage with notebooks from a code-centric perspective.

Field Experts: This group included domain specialists (e.g., public health re-

searchers, urban planners, economists) with little to no programming background. These users typically interact with notebook content through its outputs, narrative elements, and visualizations. Their perspective is primarily analytical or decision-focused rather than implementation-driven.

All participants were either actively working in their respective fields or were enrolled in graduate-level academic programs. This ensured that each user had sufficient background knowledge to engage meaningfully with the content and provide relevant feedback based on their expertise.

Including both groups allowed us to evaluate how well the system supports onboarding across a wide range of user profiles. This distinction was essential, as onboarding in computational notebooks often involves both code comprehension and domain-specific reasoning, challenges that manifest differently for coders and field experts.

4.2 Test Data

To evaluate the tool's performance in realistic scenarios, we selected three types of notebook-based tasks. These test cases were chosen to reflect different levels of complexity, types of workflows, and degrees of documentation quality.

Paper Dataset: The first and most complex test was built using the notebook and dataset from the study *"One City, Two Tales: Using Mobility Networks to Understand Neighborhood Resilience and Fragility during the COVID-19 Pandemic"* Boz et al. (2024). This notebook analyzes human mobility in New York City using network metrics such as betweenness centrality and ego-network dissimilarity. It includes dense analytical content, numerous figures, and domain-specific narrative. One of the key figures used in our study (see Figure 4.1) presents the socioeconomic distribution of neighborhoods in the top and bottom mobility dissimilarity quartiles during the first wave of the pandemic.

The study constructs weekly mobility networks between Census Block Groups (CBGs) by aggregating SafeGraph POI visit data between January 2019 and December 2020. These mobility patterns are represented as directed graphs, capturing flows between neighborhoods. Neighborhood-level adaptability is then modeled as a function of mobility change, with higher adaptability indicating greater compliance with shelter-in-place policies.

The analysis incorporates both socioeconomic variables (such as income, race, and education) and geographic characteristics (such as local amenity access) to understand how different communities responded to the pandemic. A key insight from the paper is that highly adaptable neighborhoods were not only socioeconomically advantaged but also had better access to local services, while low-adaptability neighborhoods lacked clear predictors.

This dataset was selected to simulate a high-barrier, research-grade notebook that poses challenges for both coders and field experts due to its multilayered preprocessing, complex visualizations, and limited inline documentation.

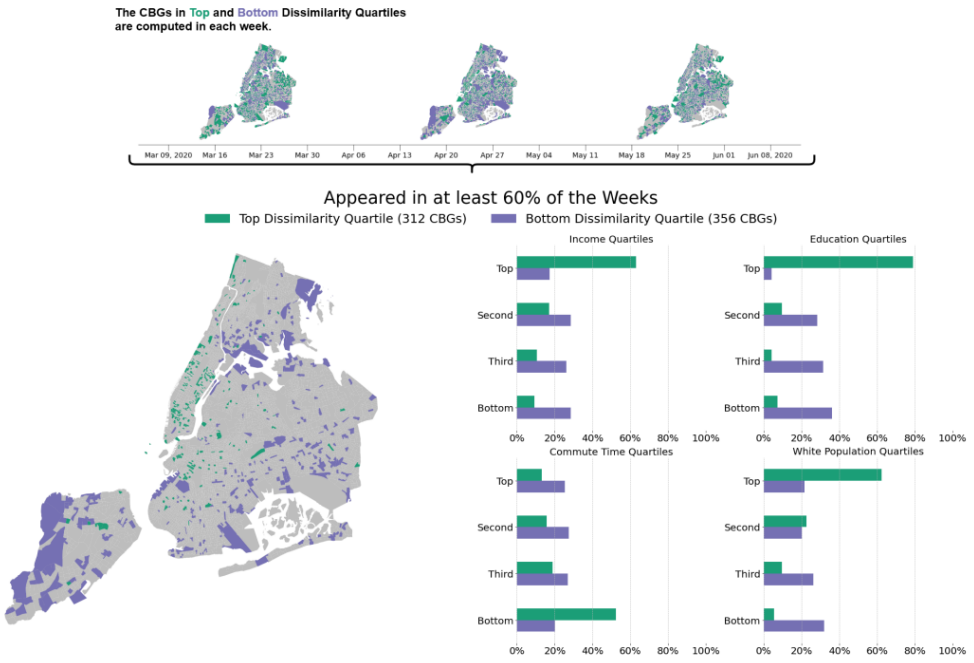

Figure 4.1 The socioeconomic distribution of Census Block Groups (CBGs) that changed their mobility patterns the most and the least during the first wave of the COVID-19 pandemic, appearing in the top and bottom dissimilarity quartiles in at least 60% of the weeks analyzed. CBGs with the most change are primarily located in the financial center of NYC. Significant socioeconomic patterns are visible for the top quartile group, while the bottom group shows no clear profile Boz et al. (2024).

Kaggle Notebooks: To evaluate the tool in more typical, mid-complexity settings, we selected two different data science notebooks from Kaggle. While these notebooks are not as detailed as the academic paper, they reflect real-world workflows commonly found in applied machine learning contexts. We ensured that both notebooks included rich datasets, meaningful visualizations, and implementations of machine learning, deep learning, or AI models. This allowed us to test the system's ability to handle a wide range of notebook elements, from exploratory data analysis to model

training and interpretation. The inclusion of both visual and model-rich content made these notebooks ideal for assessing how well the tool supports multi-faceted analytical tasks.

Unit Tests: We also created a series of isolated, focused prompts to test individual system capabilities, such as code generation, output explanation, and summarization. These unit tests provided a controlled way to measure how well the tool performs on narrowly defined onboarding tasks, independent of broader notebook context.

By combining these three types of data sources, we ensured the evaluation covers a range of difficulty levels and use cases reflective of real-world notebook usage.

4.3 Experiment Setup and Questionnaire

The user study was structured around three experimental conditions, each targeting different evaluation goals. All participants completed tasks within each condition, allowing us to measure both their standalone performance and their interaction with the tool.

1. Kaggle Dataset Condition (A/B Test): Participants were randomly assigned one of the two Kaggle notebooks (A or B). In the first round, they completed a set of tasks without the help of the tool. In the second round, they switched to the other notebook and used the tool to complete a similar set of tasks on a different dataset and code structure. While task formats were comparable to enable consistent evaluation, content differences helped mitigate learning effects between rounds. After each round, they completed a questionnaire. This A/B design enabled us to compare performance and satisfaction before and after tool use.

2. Paper Dataset Condition: Participants were provided with the notebook based on the COVID-19 mobility network study Boz et al. (2024). They were asked to explore the notebook and use the tool to ask questions, request explanations, and engage with outputs. Afterward, they completed a questionnaire assessing their understanding and experience. This condition was designed to evaluate the tool's performance in highly complex, domain-specific analytical contexts.

3. Unit Test Condition: In this setup, participants interacted with the tool on focused, single-function tasks. Examples included generating a missing func-

tion, summarizing a complex cell, or explaining a chart. Each test was followed by targeted questions to assess how well the system fulfilled the requested task.

Questionnaire Design: Participants were asked to rate their experience on a 1–10 scale across the following dimensions (asked after each notebook or task):

- How easy was it to understand the data used in the notebook?
- How easy was it to understand the code of the notebook?
- Was the explanation in the notebook sufficient?
- How confident do you feel in continuing to work on the project?

After using the tool, participants were also asked:

- How well did the LLM explain the notebook content?
- Would you consider using this tool in the future?

For unit test tasks:

- How well did the system perform code generation?
- How helpful was the summary?
- How clear was the explanation?

An optional open-ended text field was also included to allow participants to leave qualitative feedback, including suggestions, praise, or critical remarks.

This experimental design allowed us to measure user satisfaction and perceived utility across varied scenarios and user types, creating a comprehensive foundation for the evaluation that follows.

Open-Ended Feedback Collection: In addition to the structured 1–10 scale questionnaire, participants were also provided with an open-ended feedback form at the end of the study. This form invited them to elaborate on their ratings, including what aspects of the tool they found most or least helpful and why. These free-text responses offered valuable qualitative insights. For example, several coders explained their low ratings for code generation by noting that they preferred to write code themselves for stylistic consistency and long-term maintainability. This qualitative feedback helped contextualize the numeric scores and informed our interpretation of user preferences.

Answer Validation and Monitoring: Given the known risk of hallucinations in large language models, all LLM-generated responses during the user study were

monitored for correctness. The study was conducted in person, with the same local computer setup for all participants. I was present during each session, actively observing the interaction between participants and the system. For both Kaggle notebooks, I had fully reviewed and tested the notebooks beforehand to understand their logic and verify key outputs. For the paper dataset, I studied the full publication and further contacted the original author to clarify any uncertainties. This allowed me to judge the correctness of the LLM's answers during the study and ensure that any hallucinated or misleading responses could be identified.

5. RESULTS & DISCUSSION

The user study involved a total of 25 participants, comprising 15 coders and 10 field experts. Among the coders, 9 were graduate students and 6 were currently working professionals. The field expert group included 7 active professionals and 3 graduate students. This distribution ensured a diverse participant pool, with real-world experience and academic expertise both represented across technical and non-technical user groups.

All participants completed all experimental conditions, as described in Chapter 4, which included interacting with academic paper-based notebooks, Kaggle notebooks, and isolated unit test tasks. Each condition was followed by a structured questionnaire and, in some cases, timed evaluations.

This chapter presents the results of the user study in detail. In Section 5.1, we summarize the evaluation metrics and structure of the questionnaire. Section 5.3 discusses the results for the paper dataset, highlighting how users navigated complex visuals and domain logic. Section 5.2 compares notebook performance with and without the tool in the Kaggle condition. Section 5.4 presents findings from the unit test evaluations. Section 5.5 analyzes written user feedback, and concludes with key insights from the study.

5.1 Overview of Evaluation Metrics

To evaluate the effectiveness of our onboarding tool, participants completed structured questionnaires following each task across three test conditions: Kaggle notebooks (A/B test), the academic paper-based notebook, and isolated unit test scenarios. All 25 participants completed all three tasks in the same order, Kaggle first, followed by the paper notebook, and then unit tests. Breaks were provided between

each stage to reduce fatigue and ensure thoughtful feedback.

All participants, regardless of background (coders or field experts), answered the same set of evaluation questions. The questionnaires were designed to measure perceived clarity, usefulness, and tool effectiveness, both with and without the assistance of the system. No task completion time was recorded, as participants were allowed to proceed at their own pace to prioritize depth of engagement over speed.

Responses were collected on a 1–10 Likert scale, where 1 represented the most negative response (e.g., very difficult, very poor) and 10 the most positive (e.g., very easy, very helpful).

General comprehension questions (used across all notebook tasks):

- **Data Understanding:** How easy was it to understand the dataset used in the notebook?
- **Code Comprehension:** How easy was it to understand the code and logic in the notebook?
- **Explanation Sufficiency:** Was the explanation (from markdown cells or the tool) sufficient to follow the analysis?
- **Project Confidence:** How confident do you feel in continuing work on this project?

These were rated separately for tasks completed with and without tool support, particularly in the Kaggle A/B test.

Tool-specific questions (only asked after tool-assisted tasks):

- **LLM Explanation Quality:** How well did the LLM explain the notebook's content?
- **Future Use Intent:** Would you consider using this tool in the future?

Unit test evaluation (feature-focused): In the final task, participants interacted with the tool in isolated test scenarios, each targeting a specific function. They were asked to rate the following:

- **Code Generation:** Quality and usefulness of the code generated by the LLM.
- **Summary Clarity:** How well the tool summarized the content of a code cell or analytical block.

- **Explanation Accuracy:** Clarity and correctness of explanations given for code or outputs.

Each questionnaire also included an optional text box for open-ended feedback. These responses were reviewed to identify qualitative insights that supplement the numerical results, including recurring praise, usability issues, and feature suggestions.

The following sections report the results per test condition, followed by a cross-group comparison and synthesis of user feedback.

5.2 Kaggle Notebook Comparison (A/B Test)

The Kaggle notebook condition was used to evaluate how the tool supports users in typical data science workflows involving visualizations and machine learning models. Each participant completed a task using one notebook without the tool, followed by a similar task using a second notebook with the tool. This A/B structure enabled within-subject comparisons.

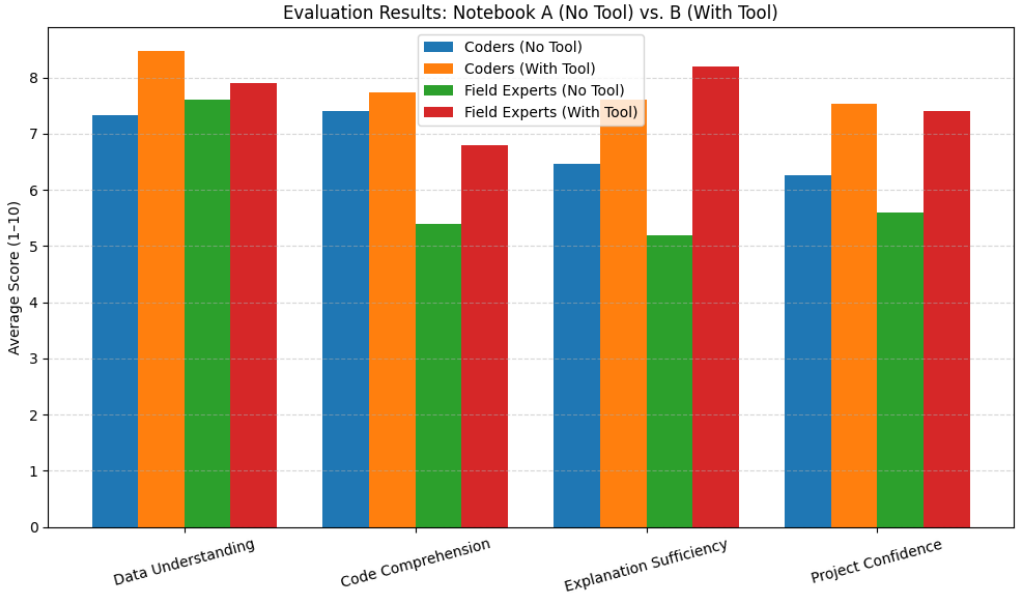

Figure 5.1 Average scores for core evaluation metrics across user groups, before and after using the tool.

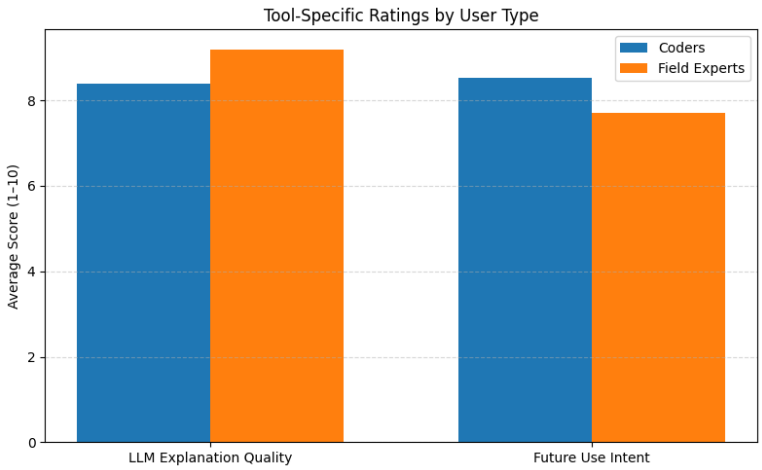

Figure 5.2 Average LLM-specific ratings after tool-assisted Kaggle tasks: explanation quality and future use intent.

Coders: Among coder participants, the tool led to consistent improvements across all evaluation metrics. The most notable gains were observed in:

- **Data Understanding:** increased from 7.33 to 8.47 (+1.13)
- **Project Confidence:** increased from 6.27 to 7.53 (+1.27)
- **Explanation Sufficiency:** improved from 6.47 to 7.60 (+1.13)

Coders also rated the tool highly for LLM explanation quality (mean: 8.4) and future use intent (mean: 8.53), suggesting a strong positive reception. These users reported that the tool helped clarify preprocessing logic, reduced lookup time, and served as a useful checkpoint for confirming interpretations.

Field Experts: For field experts, the tool had an even more pronounced impact. The largest improvements were in:

- **Explanation Sufficiency:** increased from 5.20 to 8.20 (+3.00)
- **Code Comprehension:** increased from 5.40 to 6.80 (+1.40)
- **Project Confidence:** increased from 5.60 to 7.40 (+1.80)

LLM explanation quality was rated extremely high (mean: 9.2), indicating that the tool effectively bridged the gap between narrative content and technical implementation. Several participants from this group noted that they would not have been able to complete the second notebook without the tool.

Visual Summary: Figure 5.1 shows the average ratings across four core dimensions (data understanding, code comprehension, explanation sufficiency, and project

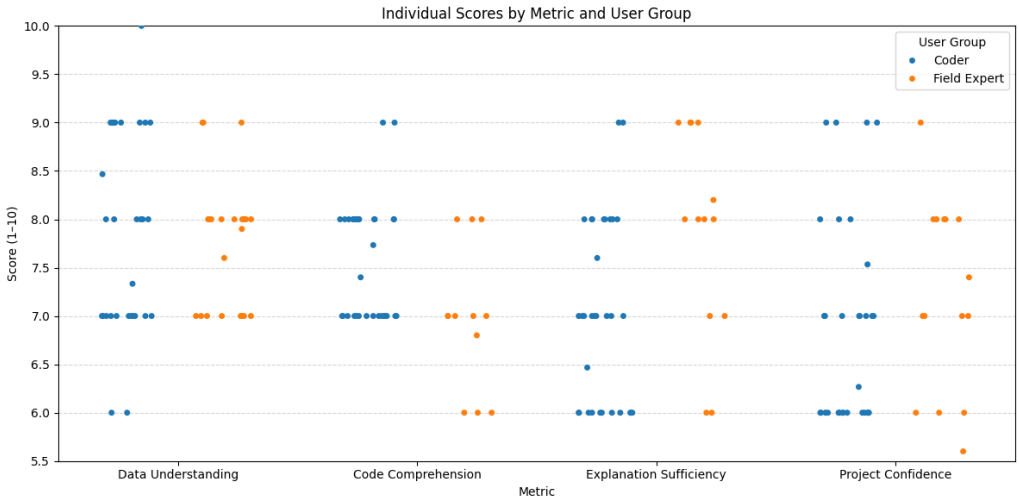

Figure 5.3 Individual participant scores across metrics, separated by user group and condition.

confidence), comparing performance with and without the tool for both user groups. Figure 5.2 summarizes the LLM-specific evaluation results. Figure 5.3 presents all individual responses, highlighting both consistency in ratings and the tool's impact.

Interpretation: The A/B test results indicate that the onboarding tool had a clear positive effect across all measured dimensions. Field experts experienced the largest relative improvement in explanation sufficiency and confidence, while coders benefited from faster comprehension and smoother onboarding into unfamiliar notebook logic. The tool's ability to support different levels of technical background through contextual explanations appears to be one of its most effective design features.

5.3 Paper Dataset Results

The academic paper-based notebook derived from the "One City, Two Tales" study Boz et al. (2024) was used to evaluate the system's ability to support onboarding in complex, domain-specific analytical environments. The notebook contained socioeconomic data, network-based mobility metrics, and dense visual outputs, making it significantly more difficult to interpret than typical data science notebooks.

Coder Performance: Coders reported a mixed experience in this condition. While they were generally able to follow the code execution flow, they struggled with

unfamiliar domain-specific visualizations and sparse documentation. Average scores were highest in *Code Comprehension* (7.27) and *Project Confidence* (7.53), indicating that their technical background helped them maintain orientation. However, lower scores in *Data Understanding* (5.20) and *Explanation Sufficiency* (4.93) suggest that the notebook's domain complexity limited full comprehension. Despite this, LLM explanation quality (8.40) and future use intent (8.87) were both rated highly.

Field Expert Performance: Field experts found this task more challenging overall. They scored relatively higher in *Data Understanding* (6.70) than coders, but considerably lower in *Code Comprehension* (4.10) and *Explanation Sufficiency* (3.60), reflecting their difficulty interpreting Python code without formal training. Still, they rated LLM explanation quality at 9.20 and future use intent at 9.30, suggesting that the tool provided substantial support in bridging the technical-language gap.

Quantitative Summary: Figure 5.4 shows average scores across both user groups for each evaluation criterion. Figure 5.5 displays individual responses, showing a consistent pattern of higher satisfaction with LLM assistance, particularly among field experts.

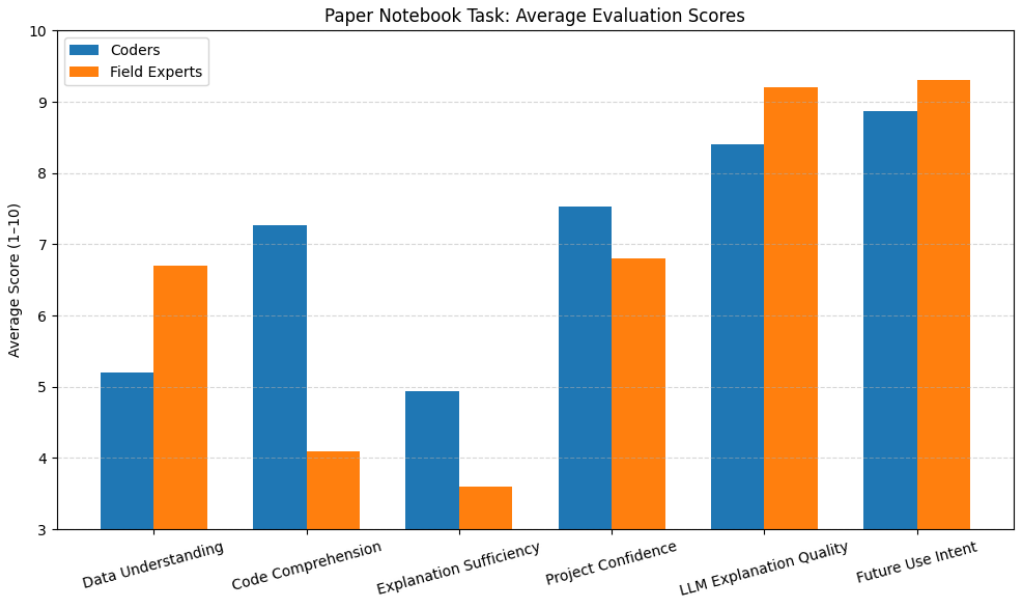

Figure 5.4 Average scores across evaluation metrics for coders and field experts in the paper dataset condition.

Interpretation: This condition demonstrated the tool's ability to assist users working with highly technical and under-documented content. Coders leveraged the tool for clarification and efficiency, while field experts depended on it to make the

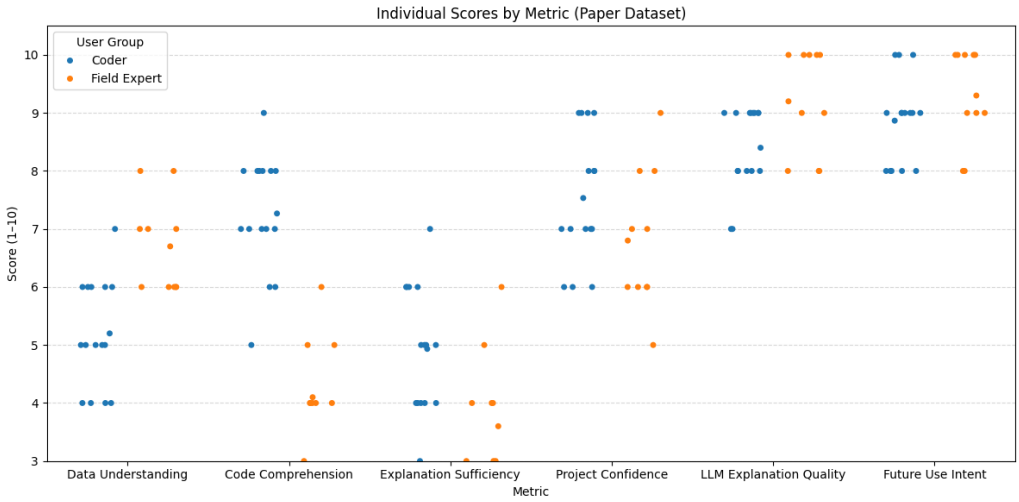

Figure 5.5 Individual participant responses across all evaluation metrics in the paper dataset condition.

notebook comprehensible at all. These results highlight the value of context-aware LLM support in research-grade analytics, where onboarding barriers are particularly steep.

5.4 Unit Test Evaluation

The final task evaluated the tool in focused, context-independent scenarios that tested individual capabilities: code generation, code summarization, and explanation. These unit tests allowed participants to assess the tool's core features in isolation, without requiring familiarity with a full notebook workflow.

Coder Performance: Coders responded positively to the summary and explanation capabilities of the tool, giving average scores of 8.07 and 8.47 respectively. However, the average score for *Code Generation* was considerably lower at 2.93. Feedback from this group revealed that the low rating was not due to inaccuracy, but rather due to preference: many coders expressed that they preferred to write code themselves. They noted that small edits were faster to perform manually than waiting for the LLM and verifying its output. Others emphasized that LLM-generated code often did not align with their personal coding style, including variable naming, abstraction preferences, and logical structuring. Several participants mentioned concerns about long-term readability and maintainability, stating that

relying on AI-generated code might lead to disorganized or inconsistent codebases.

Field Expert Performance: Field experts, in contrast, rated all three capabilities highly. Their average scores were 7.50 for *Code Generation*, 7.40 for *Summary*, and 8.90 for *Explanation*. Participants in this group frequently commented that receiving immediate answers and direct code modifications from the LLM saved time and reduced cognitive overhead. For many, the tool helped them engage with the notebook in ways that would otherwise have been inaccessible due to limited programming experience.

Comparison Table: Table 5.1 summarizes the average scores across user groups for each unit test capability.

Table 5.1 Average unit test scores by user group.

User Group	Code Generation	Summary	Explanation
Coders	2.93	8.07	8.47
Field Experts	7.50	7.40	8.90

Interpretation: These results highlight a divergence in expectations between the two user groups. Coders were less interested in AI-generated code due to speed, stylistic preferences, and concerns about long-term maintainability. On the other hand, field experts valued the immediacy and automation provided by the tool, particularly in tasks they would have struggled to perform manually. While improvements in generation quality may help address coder concerns, the tool's summarization and explanation capabilities already offer substantial onboarding value to users with diverse technical backgrounds.

5.5 Discussion

Table 5.2 summarizes average scores across all evaluation metrics, user groups, and task types. This unified view highlights key trends in how coders and field experts interacted with the tool across different contexts, and how their needs and expectations diverged.

Key Insights: The tool consistently improved participants' understanding, explanation satisfaction, and project confidence, especially among field experts. Gains

Table 5.2 Average scores by user group, task, and evaluation metric.

Metric	Kaggle (No)	Kaggle (With)	Paper	Unit Test
Coders				
Data Understanding	7.33	8.47	5.20	–
Code Comprehension	7.40	7.73	7.27	–
Explanation Sufficiency	6.47	7.60	4.93	–
Project Confidence	6.27	7.53	7.53	–
LLM Explanation Quality	–	8.40	8.40	–
Future Use Intent	–	8.53	8.87	–
Code Generation	–	–	–	2.93
Code Summary	–	–	–	8.07
Explanation Accuracy	–	–	–	8.47
Field Experts				
Data Understanding	7.60	7.90	6.70	–
Code Comprehension	5.40	6.80	4.10	–
Explanation Sufficiency	5.20	8.20	3.60	–
Project Confidence	5.60	7.40	6.80	–
LLM Explanation Quality	–	9.20	9.20	–
Future Use Intent	–	7.70	9.30	–
Code Generation	–	–	–	7.50
Code Summary	–	–	–	7.40
Explanation Accuracy	–	–	–	8.90

were most pronounced in the Kaggle A/B test, where participants had the opportunity to directly compare assisted and unassisted experiences. Field experts benefited most in explanation sufficiency (+3.0 improvement) and confidence, confirming the tool's utility in lowering onboarding barriers.

The paper dataset, while more complex, confirmed the same pattern: field experts relied heavily on the tool to understand outputs and connect code to real-world meaning. Coders, although more comfortable navigating the notebook, still benefited from LLM explanations to reduce the cognitive effort of working through dense visual or domain-specific content.

Code Generation Differences: In the unit test evaluation, coders rated code generation significantly lower than field experts. However, this reflected a practical stance rather than tool failure. Coders reported that they preferred to write code themselves due to efficiency, control over naming and logic structure, and long-term code quality. They expressed concern that relying on LLMs could lead to inconsistent or difficult-to-maintain codebases.

In contrast, field experts were generally satisfied with LLM-generated code, viewing it as a short-term solution or support mechanism while awaiting input from a de-

veloper. They emphasized the value of getting "something that works" quickly and described the tool as reducing dependency on technical collaborators for routine or exploratory tasks.

General Patterns: Summarization and explanation were rated highly by both groups, with particularly strong results among field experts. These two features appear to bridge technical skill gaps and support learning and handoff, two of the most important factors in real-world onboarding scenarios. Even in high-complexity cases like the paper dataset, users felt more equipped to interpret notebooks with the LLM's assistance.

Interestingly, coders also highlighted the value of these features, not for learning new content, but for navigating poorly documented notebooks. Several participants mentioned that the LLM effectively filled in for missing author guidance, such as explaining undocumented functions, describing cell-level intent, or summarizing sections in notebooks that lacked headers or clear structure. In this way, the LLM functioned as a contextual layer that made otherwise opaque notebooks searchable and more coherent.

Conclusion: The results demonstrate that an LLM-powered onboarding assistant can address real bottlenecks in both code-centric and analysis-centric workflows. While generative code support remains controversial among experienced programmers, explanation and summary capabilities already provide broad value and promote deeper engagement with unfamiliar projects.

6. LIMITATIONS & FUTURE WORK

While the proposed tool demonstrates strong potential for improving onboarding in interactive computational notebooks, several limitations remain that offer directions for future enhancement.

First, the system relies on GPT-4 via the ChatGPT API, which, while general-purpose and powerful, introduces cost, latency, and version-dependence issues. While GPT-4's versatility made it ideal for supporting both code and domain-level tasks, more specialized or locally hosted models could improve response speed, reduce cost, and increase reproducibility.

We briefly explored using separate LLMs for code generation and explanation, but found this approach impractical within our constraints. The code generation task often required deep understanding of user questions, prior outputs, and notebook structure, all of which demanded general-purpose language capabilities. Specialized models lacked this flexibility, and due to hardware limitations we were unable to experiment with larger or fine-tuned alternatives. Future work could investigate combining lightweight domain-specific models in a multi-agent setup for improved modularity and privacy.

Second, the directed graph used for context retrieval is based on relatively simple heuristics, such as sequential cell order and function usage detection. While this proved sufficient in many cases, it may fail in notebooks with highly nonlinear execution paths or complex variable dependencies. Incorporating static code analysis or runtime tracing could lead to more accurate contextualization.

Third, although the user study included participants from both technical and non-technical backgrounds, the sample size (25 participants) limits generalizability.

A key usability concern raised during testing was the low rating of code generation by experienced coders. This was not due to technical failure but rather philosophical and stylistic preferences, many coders preferred to retain direct control over their code for long-term maintainability. This highlights the need for customizable LLM

behavior, such as enforcing style conventions or scoping suggestions within stricter constraints.

Another practical limitation is the lack of automated dependency management. If the LLM generates code that requires a library not yet installed in the user's environment, installation must currently be done manually. In future iterations, the tool could proactively detect missing packages and offer automated installation to streamline usability.

For future work, we aim to enhance the context retrieval system with more advanced dependency tracking, explore multi-agent LLM architectures for task decomposition, and offer user-tunable parameters for LLM behavior (e.g., verbosity, technicality, coding style). We also plan to expand testing to larger, more heterogeneous user groups and integrate version control awareness to support onboarding in evolving notebooks.

Ultimately, this work opens the door to broader applications of LLMs in human-in-the-loop workflows and suggests that onboarding support can be meaningfully improved through natural language interaction, even in complex, mixed-media environments like Jupyter notebooks.

7. CONCLUSION

This thesis presented a tool designed to support onboarding in interactive computational notebooks using large language models. The system integrates program analysis, graph-based context retrieval, and natural language prompting to assist both coders and field experts in understanding, navigating, and contributing to existing notebooks.

Onboarding in computational notebooks is often hindered by the informal, nonlinear, and densely interwoven nature of code, outputs, and narrative explanations. Our tool addresses this by enabling users to ask questions, generate new code, and receive contextual explanations or summaries at any point in the notebook, using a directed graph to retrieve relevant context dynamically.

To evaluate the system, we conducted a user study involving 25 participants, including both programmers and non-programmers. The study demonstrated that the tool improves code comprehension, data understanding, and user confidence, especially for field experts with limited programming experience. While code generation was well received by non-programmers, experienced coders expressed reservations, preferring full control over implementation for reasons of maintainability and consistency.

The results suggest that LLM-based tools can meaningfully assist onboarding workflows in data science environments, particularly through summarization and explanation rather than full automation. The ability to retrieve context-aware answers and incrementally build understanding offers substantial value, especially in high-complexity or poorly documented notebooks.

This work contributes both a practical implementation and an empirical understanding of how LLMs can support interdisciplinary collaboration and knowledge transfer in notebook-based workflows. Future directions include deeper integration with runtime environments, improved dependency management, user-customized LLM behavior, and expanded testing in real-world team settings.

By bridging the gap between natural language and code, and between developers and domain specialists, this tool lays the groundwork for more accessible, intelligent, and collaborative notebook systems.

BIBLIOGRAPHY

Ahmed, T. & Devanbu, P. (2023). Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Boz, H. A., Bahrami, M., Balcisoy, S., Bozkaya, B., Mazar, N., Nichols, A., & Pentland, A. (2024). Investigating neighborhood adaptability using mobility networks: a case study of the covid-19 pandemic. *Humanities and Social Sciences Communications*, *11*(1).

Chandel, S., Clement, C. B., Serrato, G., & Sundaresan, N. (2022). Training and evaluating a jupyter notebook data science assistant.

Chase, H. (2022). LangChain.

Dagenais, B., Ossher, H., Bellamy, R. K. E., Robillard, M. P., & de Vries, J. P. (2010). Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (pp. 275–284)., New York, NY, USA. Association for Computing Machinery.

Dong, J., Lou, Y., Zhu, Q., Sun, Z., Li, Z., Zhang, W., & Hao, D. (2022). Fira: Fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (pp. 970–981)., New York, NY, USA. Association for Computing Machinery.

et al., A. G. (2024a). The llama 3 herd of models. For the full list of authors, refer to the original publication.

et al., B. R. (2024b). Code llama: Open foundation models for code. For the full list of authors, refer to the original publication.

Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., & Lahiri, S. K. (2024). Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, *50*(9), 2254–2268.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages.

Hoang, T., Kang, H. J., Lo, D., & Lawall, J. (2020). Cc2vec: distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (pp. 518–529)., New York, NY, USA. Association for Computing Machinery.

Ju, A., Sajnani, H., Kelly, S., & Herzig, K. (2021). A case study of onboarding in software teams: Tasks and strategies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, (pp. 613–623).

Kim, M., Zimmermann, T., DeLine, R., & Begel, A. (2016). The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (pp. 96–107)., New York, NY, USA. Association for Computing Machinery.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International*

Conference on Software Engineering, ICSE '06, (pp. 492–501)., New York, NY, USA. Association for Computing Machinery.

Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A., & Hellas, A. (2023). Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, (pp. 124–130)., New York, NY, USA. Association for Computing Machinery.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Li, C., Xu, Z., Di, P., Wang, D., Li, Z., & Zheng, Q. (2024). Understanding code changes practically with small-scale language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, (pp. 216–228)., New York, NY, USA. Association for Computing Machinery.

Li, X., Zhang, Y., Leung, J., Sun, C., & Zhao, J. (2023). Edassistant: Supporting exploratory data analysis in computational notebooks with in situ code search and recommendation. *ACM Trans. Interact. Intell. Syst.*, *13*(1).

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Liu, Z., Tang, Z., Xia, X., & Yang, X. (2023). Ccrep: Learning code change representations via pre-trained code model and query back. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, (pp. 17–29).

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., & Liu, S. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation.

MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., Bernstein, S., & Leinonen, J. (2023). Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, (pp. 931–937)., New York, NY, USA. Association for Computing Machinery.

Matturro, G., Barrella, K., & Benitez, P. (2017). Difficulties of newcomers joining software projects already in execution. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, (pp. 993–998).

Muller, M., Lange, I., Wang, D., Piorkowski, D., Tsay, J., Liao, Q. V., Dugan, C., & Erickson, T. (2019). How data science workers work with data: Discovery, capture, curation, design, creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, (pp. 1–15)., New York, NY, USA. Association for Computing Machinery.

Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024a). Using

an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024b). Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

OpenAI (2023). Gpt-4: Generative pre-trained transformer 4.

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019). A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, (pp. 507–517).

Robillard, M., Walker, R., & Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, *27*(4), 80–86.

Rule, A., Tabard, A., & Hollan, J. D. (2018). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, (pp. 1–12)., New York, NY, USA. Association for Computing Machinery.

Steinmacher, I., Treude, C., & Gerosa, M. A. (2019). Let me in: Guidelines for the successful onboarding of newcomers to open source projects. *IEEE Software*, *36*(4), 41–49.

Wang, A. Y., Mittal, A., Brooks, C., & Oney, S. (2019). How data scientists use computational notebooks for real-time collaboration. *Proc. ACM Hum.-Comput. Interact.*, *3*(CSCW).

Wang, A. Y., Wang, D., Drozdal, J., Muller, M., Park, S., Weisz, J. D., Liu, X., Wu, L., & Dugan, C. (2022). Documentation matters: Human-centered ai system to assist data science code documentation in computational notebooks. *ACM Trans. Comput.-Hum. Interact.*, *29*(2).

Wang, J., Kuo, T.-y., Li, L., & Zeller, A. (2021). Assessing and restoring reproducibility of jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, (pp. 138–149)., New York, NY, USA. Association for Computing Machinery.

Wang, J., Li, L., & Zeller, A. (2020). Better code, better sharing: on the need of analyzing jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, (pp. 53–56)., New York, NY, USA. Association for Computing Machinery.

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Wenskovitch, J., Zhao, J., Carter, S., Cooper, M., & North, C. (2019). Albireo: An interactive tool for visually summarizing computational notebook structure. In *2019 IEEE Visualization in Data Science (VDS)*, (pp. 1–10).

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, *44*(10), 951–976.

Xiao, R., Hou, X., & Stamper, J. (2024). Exploring how multiple levels of gpt-generated programming hints support or disappoint novices. In *Extended Ab-*

stracts of the CHI Conference on Human Factors in Computing Systems, CHI EA '24, New York, NY, USA. Association for Computing Machinery.

Zhao, Y., Zhang, Y., Zhang, Y., Zhao, X., Wang, J., Shao, Z., Turkay, C., & Chen, S. (2025). Leva: Using large language models to enhance visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, *31*(3), 1830–1847.

Zhou, Z., Li, M., Yu, H., Fan, G., Yang, P., & Huang, Z. (2024). Learning to generate structured code summaries from hybrid code context. *IEEE Transactions on Software Engineering*, *50*(10), 2512–2528.

Zhu, J., Miao, Y., Xu, T., Zhu, J., & Sun, X. (2024). On the effectiveness of large language models in statement-level code summarization. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, (pp. 216–227).