

Interactive Programming Interfaces for Data Science Collaboration and Learning

by

April Yi Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information)
in the University of Michigan
2023

Doctoral Committee:

Professor Steve Oney, Co-Chair
Professor Christopher Brooks, Co-Chair
Dr. Steven Drucker, Microsoft
Professor Philip Guo
Professor Cyrus Omar

April Yi Wang

aprilww@umich.edu

ORCID iD: 0000-0001-8724-4662

© April Yi Wang 2023

To my mother, Haihong Cheng, and my father, Haijun Wang.

ACKNOWLEDGMENTS

First, I am deeply thankful for my Ph.D. advisors, Steve Oney and Christopher Brooks. They have been my teammates, coaches, cheerleaders, and anything you can name it. They always found my early ideas interesting and turned every single meeting into a goldmine of feedback and guidance. It is impossible to finish this dissertation without their help. They have also treated me as an independent scholar from day one, showing me both sides of a coin and encouraging me to make the final decision. They always backed me up for the big time, pushing me to dive into different research communities and work with various researchers and students. They also showed me what it really means to develop a career in the research field and how to strike a balance between work and life. But it does not stop there. They taught me the ideal attitude toward imperfections, criticisms, rejections, and self-doubts. These lessons shape not only my academic journey but also my personal development.

I would also like to thank the members of my dissertation committee — Cyrus Omar, Philip Guo, and Steven Drucker for their constructive comments and suggestions. In particular, I am thankful to Cyrus Omar for serving on my prelim committee and dissertation committee as a cognate member. Our conversations have consistently sparked deep inspiration within me. I am fortunate to have Philip Guo on both my master’s dissertation committee and Ph.D. dissertation committee. His mentorship, encompassing both academic and career aspects, has always motivated me throughout my Ph.D. journey. Working with Steven Drucker and the VIDA group during my Microsoft internship was an invaluable experience. I have gained a lot of inspiration for the future of programming tools through our conversations.

I am fortunate to have developed my research interests within the field of human-computer interaction, crossing paths with numerous brilliant minds also passionate about making technology more usable for humans. I hold a special place in my heart for Parmit Chilana for welcoming me to the HCI world. My gratitude extends to Dakuo Wang, Michael Muller, Robert DeLine, David Piorkowski, Michael Nebeling, Robin Brewer, Xu Wang, Katie Cunningham, Andrew Head, and Yan Chen. I am also thankful for all

the reviewers in the HCI community whose feedback, though at times bitter, has been the catalyst for strengthening my projects. Additionally, I owe a debt of thanks to all the study participants who have contributed their valuable time and effort.

I feel truly blessed to have so many great friends who have been there through the ups and downs of this journey. A huge shout-out to Warren Li, Anjali Singh, Heeryung Choi, Kaiwen Sun, Rebecca Krosnick, Maulishree Pandey, Ashley Zhang, Lei Zhang, John Joon Young Chung, Soya Park, Xinghui Yan, Zhuofeng Wu, Qiaoning Zhang, Yixin Zou, Jane Im, Woosuk Seo, Hrishikesh Rao, Zihan Wu, Qingyi Wang, and Olivia Richards.

And last but not least, my heartfelt thanks to my parents and family for their unconditional love and support. ♥

TABLE OF CONTENTS

DEDICATION	ii
ACKNOWLEDGMENTS	iii
LIST OF FIGURES	xvi
LIST OF TABLES	xxi
LIST OF APPENDICES	xxiv
LIST OF ACRONYMS	xxv
ABSTRACT	xxvii

CHAPTER

1 Introduction	1
1.1 Understand Challenges in Collaborative Data Science	4
1.1.1 A Mixed-Methods Study to Understand Real-Time Collaborative Notebooks	4
1.2 Design Collaborative Computational Notebook Environments for Data Scientists	5
1.2.1 Contextualizing Shared Notebooks with Discussions	5
1.2.2 AI-Assisted Data Science Code Documentation	6
1.2.3 Improving Awareness with Data Changes	6
1.2.4 Resolving Editing Conflicts in Real-Time Collaboration	7
1.3 Facilitate Knowledge Sharing for Future Data Scientists	7
1.3.1 Real-Time Sharing for In-Class Programming Exercises	8
1.3.2 Authoring Explorable Multi-Stage Tutorials	8
1.4 Thesis Statement	9
1.5 A Note on Authorship	10
2 Related Work	11
2.1 Data Science Programming	11
2.1.1 Data Science Workflow	11

2.1.2	Research Programming	12
2.1.3	Exploratory Programming	13
2.1.4	Collaboration in Data Science	14
2.2	Computational Notebooks	14
2.2.1	Jupyter Notebook	15
2.2.2	The Use of Computational Notebook	16
2.2.2.1	Working with Computational Notebooks	16
2.2.2.2	Sharing and Reusing Computational Notebooks	17
2.2.2.3	Teaching with Computational Notebooks	18
2.3	Improving Computational Notebooks	19
2.3.1	Lowering the Barriers to Programming	19
2.3.2	Encouraging Explanations	20
2.3.3	Scaling Data Sources	21
2.3.4	Reactivity	21
2.3.5	Managing Out-of-order Cells and Throw-away Code	22

Part 1 Understand Challenges in Collaborative Data Science 24

3	How Data Scientists Use Computational Notebooks for Real-Time Collaboration	25
3.1	Introduction	26
3.2	Overview of the Methodology	28
3.3	Study 1: Formative Survey on Collaborative Data Science	29
3.3.1	The Survey Instrument	30
3.3.2	Data Analysis	30
3.4	Key Findings from Study 1	31
3.4.1	Data Overview	31
3.4.2	Experience with Collaborative Data Science	32
3.4.2.1	Choices of Tools	32
3.4.2.2	Strategies for Keeping a Shared Understanding	33
3.4.3	High-Level Summary of Findings	34
3.5	Study 2: Observational Study on Real-time Collaborative Data Science	34
3.5.1	Participants and Task	36
3.5.2	Apparatus	36
3.5.2.1	Non-Shared Condition	37
3.5.2.2	Shared Condition	37
3.5.3	Collaboration Extension	37
3.5.3.1	Workflow	37
3.5.3.2	Implementation	37
3.5.4	Study Procedure	38
3.5.5	Data Analysis	39
3.6	Key Findings from Study 2	40

3.6.1	Collaboration and Communication Styles	40
3.6.1.1	Single Authoring Style	41
3.6.1.2	Pair Authoring Style	41
3.6.1.3	Divide and Conquer	42
3.6.1.4	Competitive Authoring	43
3.6.2	Communication Channels	43
3.6.3	Working Across Phases	44
3.6.4	Overall Effectiveness — Accuracy and Questionnaire Results . .	46
3.6.5	Challenges in Using the Collaborative Notebooks	47
3.6.5.1	Interference with Each Other	47
3.6.5.2	Lack of Awareness	48
3.6.5.3	Problems with the Linear Structure	49
3.6.5.4	Privacy Concerns	50
3.6.5.5	Lack of Strategic Coordination	50
3.6.5.6	Contextual Chatting	51
3.7	Discussion	52
3.7.1	Extending Our Understanding of Collaborative Editing Across Contexts	52
3.7.2	Opportunities and Challenges of Collaboration in Computational Notebooks	53
3.7.3	Design Implications	54
3.7.3.1	Improve Awareness of Collaborators’ Activity	54
3.7.3.2	Provide Access Control	55
3.7.3.3	Enable Discussions within Notebooks	55
3.7.4	Limitations	56
3.8	Conclusion	56
3.9	Acknowledgements	56

Part 2 Design Collaborative Computational Notebook Environ- ments for Data Scientists 58

4	Callisto: Contextualizing Shared Notebooks with Discussions	59
4.1	Introduction	59
4.2	Formative Study	61
4.2.1	Method	62
4.2.2	Data Analysis	62
4.2.3	Results	62
4.2.3.1	Purpose	62
4.2.3.2	Relevance	64
4.2.3.3	Granularity	64
4.2.4	Implications	65
4.3	Design of Callisto	65

4.3.1	Enabling Sharing and Real-Time Collaboration	65
4.3.1.1	Basic Collaboration Features	66
4.3.1.2	Shared Runtime and Outputs	67
4.3.1.3	Synchronous Chat	67
4.3.1.4	Edit and Version History	68
4.3.2	Connecting Messages and Notebook Content	68
4.3.2.1	Automatically Inferring References from Context	69
4.3.3	Navigating Messages and Notebook Content	70
4.3.3.1	From Messages to Notebook Content	70
4.3.3.2	From Notebook Content to Messages	70
4.4	Evaluation	71
4.4.1	General Study Protocol (for Both Stages)	72
4.4.2	Participants (for Both Stages)	73
4.4.3	Stage 1: Real-time Collaboration	73
4.4.3.1	Overall Usage	74
4.4.3.2	Creating References (Manual and Automatically Inferred)	74
4.4.3.3	Annotations Aid Communication	75
4.4.4	Stage 2: Following up with the Collaboration Process	75
4.4.4.1	Content Preparation	76
4.4.4.2	Study Setup	76
4.4.4.3	Overall Performance	77
4.4.4.4	Understanding Discussions Around the Cell	77
4.4.4.5	Understanding the Context of the Message	78
4.5	Discussion	79
4.5.1	Reducing the Burden of Communication	79
4.5.2	Improving the Accuracy of Contextual Links	80
4.5.3	Towards Generating Meta-Narratives	80
4.5.4	Limitations	80
4.6	System Implementation	81
4.7	Conclusion	81
4.8	Acknowledgments	81
5	Themisto: AI-Assisted Data Science Code Documentation	82
5.1	Introduction	82
5.2	Formative Study	84
5.2.1	Data Collection	85
5.2.2	Data Analysis	86
5.2.3	Results	86
5.2.3.1	Descriptive statistics of the notebook.	86
5.2.3.2	Data scientists use markdown cells to document a broad range of topics.	87
5.2.3.3	Data science stages.	88

5.2.4	Design Implications	88
5.3	Design and Implementation	90
5.3.1	System Architecture	91
5.3.2	User Interface Design	91
5.3.3	Three Approaches for Documentation Generation	92
5.3.3.1	Deep-Learning-Based Approach	94
5.3.3.2	Query-Based Approach	96
5.3.3.3	Prompt-Based Approach	96
5.4	User Evaluation of Themisto	97
5.4.1	Participants	97
5.4.2	Study Protocol	97
5.4.3	Data Collection and Measurements	99
5.4.4	Results	101
5.4.4.1	Themisto supports participants to easily add documentations to a notebook.	101
5.4.4.2	Co-creation yields longer documentation and improves accuracy and readability.	102
5.4.4.3	Themisto increases participant’s satisfaction , while maintaining a similar quality of the final notebook.	103
5.4.4.4	The three approaches of generating documentation are suitable for different scenarios.	105
5.4.4.5	Will participants use Themisto in their future data science project?	106
5.4.4.6	Participants suggest various design implications for automated code documentation.	107
5.4.4.7	Summary of the Results	108
5.5	Discussion	108
5.5.1	The Documentation Practices in Data Science is Different from in Software Engineering	108
5.5.2	Human-AI Collaboration in Code Documentation in Data Science	109
5.5.3	Design Implications	111
5.5.3.1	Towards Hybrid and Adapted Code Summarization	111
5.5.3.2	Customizing the Recommendations based on Usage Scenarios	112
5.5.3.3	Inverting Themisto – Automatic Code Generation from Documentation.	112
5.5.4	Limitation	112
5.6	Conclusion	113
5.7	Acknowledgements	114
6	DITL: Improving Awareness with Data Changes	115
6.1	Introduction	115
6.2	Design Motivations	118

6.2.1	Understanding the Impact of Code Changes in Debugging	118
6.2.2	Gaining Insights in Data Through Comparisons	118
6.2.3	Improving Awareness in Collaboration	119
6.3	System Design	120
6.3.1	Overview of DITL Study Apparatus	120
6.3.2	Tracking Runtime Variables	120
6.3.3	Comparing Changes in Data Frames	121
6.3.4	Rendering Data Frame Diffs	122
6.3.4.1	Parallel View	122
6.3.4.2	Opacity View	123
6.3.4.3	Delta View	123
6.4	Usability Study	123
6.4.1	Method	124
6.4.1.1	Recruitment	124
6.4.1.2	Study Setup	124
6.4.2	Results	125
6.4.2.1	The need to compare data tables	125
6.4.2.2	DITL makes comparison easier	126
6.4.2.3	Feedback on the Visualizations	127
6.4.2.4	Preferences for integration	128
6.5	Discussion and Future Work	129
6.5.1	Towards a Design Space for Visualizing Data Comparisons	129
6.5.2	Generalizing from Comparing Data Tables to Comparing Arbitrary Charts	129
6.5.3	Integrating DITL in Data Science Programming Environments . .	130
6.5.4	Limitations	131
6.5.4.1	Limitations of DITL	131
6.5.4.2	Limitations of the Evaluation	131
6.6	Conclusion	131
6.7	Acknowledgements	132
7	PADLOCK: Resolving Editing Conflicts in Real-Time Collaboration	133
7.1	Introduction	133
7.2	Design Motivations	136
7.2.1	Implementing the Same-Purpose Code at the Same Time	136
7.2.2	Using or Changing the Same Variable at the Same Time	137
7.2.3	Other Needs for Access Control in RTC	138
7.3	System Overview	138
7.3.1	Cell-Level Access Control	138
7.3.2	Variable-Level Access Control	140
7.3.3	Parallel Cell Groups	140
7.3.4	Implementation	142
7.3.4.1	Cell-Level Access Control	142

	7.3.4.2	Variable-Level Access Control	142
	7.3.4.3	Parallel Cell Groups	143
7.4		Evaluation Overview	147
	7.4.1	Participants	147
7.5		Paired Session: Handling Situations of Editing Conflicts	148
	7.5.1	Study Setup	148
	7.5.2	Task Description	149
	7.5.3	The Clumsy Collaborator	149
	7.5.4	Data Analysis	150
	7.5.5	Result	151
	7.5.5.1	Conflict editing is hard to notice and prevent	151
	7.5.5.2	Perceptions of PADLOCK for preventing conflict editing	151
	7.5.5.3	Improvement of the Parallel Cell	153
	7.5.5.4	Resonate with prior experience	153
7.6		Planning Session: Planning for Various Collaboration Scenarios	154
	7.6.1	Study Setup	154
	7.6.2	Collaboration Scenarios	155
	7.6.2.1	Scenario 1: Asynchronous Collaboration with a Peer	155
	7.6.2.2	Scenario 2: Working with Trainees and a High Cost of Error Recovery	155
	7.6.2.3	Scenario 3: Classroom Sharing with Hierarchical Permissions	155
	7.6.3	Data Analysis	156
	7.6.4	Results	156
	7.6.4.1	Usage of the Collaborative Features for Each Scenario	156
	7.6.4.2	Effectiveness of the Collaboration Plan	158
	7.6.4.3	Improving Access Control	159
7.7		Group Session: Case Study on Open-Ended Collaboration	160
	7.7.1	Study Setup	160
	7.7.2	Task Description	161
	7.7.3	Data Analysis	161
	7.7.4	Result	161
	7.7.4.1	G1: Starting from Pair Authoring	161
	7.7.4.2	G2: Starting from Divide and Conquer	163
	7.7.4.3	Reflections on the Collaborative Session	164
7.8		Discussion and Future Work	165
	7.8.1	Lightweight Collaboration Support for Data Science	165
	7.8.2	From Small Groups to Collaboration at Scale	165
	7.8.3	Blending Sync and Async Collaboration in Long Terms	166
	7.8.4	Improving Awareness of Collaborators' Activities	166
	7.8.5	Limitations	167
	7.8.5.1	Limitations of the Evaluation	167
	7.8.5.2	Limitations of PADLOCK	167

7.9	Conclusion	167
-----	----------------------	-----

Part 3 Facilitate Knowledge Sharing for Future Data Scientists 168

8 PuzzleMe: Leveraging Peer Assessment for In-Class Programming Exercises 169

8.1	Introduction	169
8.2	In-class Programming Exercise Challenges	171
8.2.1	Method	172
8.2.2	Findings	173
8.2.2.1	In-class exercises are common	173
8.2.2.2	Impromptu exercises are valuable	173
8.2.2.3	Hard to scale support for exercises	174
8.2.2.4	Difficulty in connecting students with each other	174
8.2.3	Summary of Design Goals	174
8.3	PuzzleMe Design	175
8.3.1	In-Class Programming Exercises	175
8.3.1.1	Live coding for answer walkthrough	176
8.3.1.2	Monitoring class progress	177
8.3.2	Live Peer Testing	177
8.3.2.1	Conceptual model of testing	177
8.3.2.2	Creating test cases	178
8.3.2.3	Verifying and sharing test cases	178
8.3.3	Live Peer Code Review	179
8.3.3.1	Matching learners	179
8.3.4	Implementation	180
8.4	Evaluation	180
8.4.1	Course Background (Studies 1 and 2)	180
8.4.2	Study 1: Using PuzzleMe in Face-to-Face Lab Sessions	181
8.4.2.1	Study setup	181
8.4.2.2	Data collection	182
8.4.2.3	PuzzleMe encourages students to create and share test cases.	182
8.4.2.4	PuzzleMe scaffolds group discussions.	184
8.4.2.5	PuzzleMe encourages students to explore alternative solutions.	185
8.4.2.6	Limitation	185
8.4.3	Study 2: Integrating PuzzleMe in an Online Lecture	186
8.4.3.1	Study setup	186
8.4.3.2	Results overview	186
8.4.3.3	Code review is correlated to better completion status	187
8.4.3.4	Who initiates the talk? Conversations need nudging.	187
8.4.3.5	Code sharing improves engagement.	188

8.4.4	Study 3: The Practical Applicability of PuzzleMe	189
8.4.4.1	The use case of PuzzleMe in various programming topics	189
8.4.4.2	PuzzleMe lowers the effort for setting up in-class exercises.	190
8.5	Discussion	191
8.5.1	Design Lessons	191
8.5.1.1	The benefits of learnersourced test creation	191
8.5.1.2	The social and cultural value of building collaborative learning platforms	192
8.5.1.3	The challenges of connecting students	192
8.5.2	Future Work	193
8.5.2.1	Towards test-driven learning	193
8.5.2.2	Peer assessment beyond introductory programming	193
8.5.2.3	Towards matching peers intelligently	194
8.5.3	Limitations	194
8.6	Conclusion	195
8.7	Acknowledgements	195
9	Colaroid: Authoring Explorable Multi-Stage Tutorials	196
9.1	Introduction	196
9.2	An Exploratory Analysis of Multi-Stage Programming Tutorials	199
9.2.1	Collecting Representative Multi-Stage Tutorials	199
9.2.1.1	Selection Criteria	199
9.2.1.2	Multi-Stage Tutorials Linked from Stack Overflow	200
9.2.1.3	Multi-Stage Tutorials on FreeCodeCamp	200
9.2.1.4	Data Analysis	200
9.2.2	Results	201
9.2.2.1	How do authors scaffold the stages?	201
9.2.2.2	How do authors structure the code snippets for a stage?	201
9.2.2.3	How do authors present the intermediate code snippets?	202
9.2.2.4	How do authors present the intermediate results?	202
9.2.2.5	How do tutorials support learning by doing?	203
9.2.3	Design Opportunities to Improve Authoring and Reading Experience	203
9.2.3.1	Design Tutorial Authoring Tools to Capture and Document the Entire Incremental Building Process	203
9.2.3.2	Design Tutorial Authoring Tools to Capture the Context of Code Changes	203
9.2.3.3	Design Tutorial Authoring Tools to Preview Multi-Stage Output	204
9.2.3.4	Design Tutorial Authoring Tools to Encourage Learning by Doing	204
9.3	System Design	205

9.3.1	Illustrative Scenario	205
9.3.2	Overview of Colaroid Notebooks	206
9.3.3	Cells as Steps in Colaroid	206
9.3.4	Authoring Tutorials by Documenting Incremental Changes	207
9.3.5	Recording Interactions in Output Widgets	208
9.3.6	Revising and Editing Colaroid Notebook Cells	209
9.3.7	Sharing and Distributing Tutorials	209
9.3.7.1	Leveraging Git for Code Versioning	209
9.3.7.2	Sharing through Cloud Platforms	210
9.3.8	Reading Tutorials	210
9.3.8.1	Explorable Explanation	210
9.3.8.2	Rendering Notebooks into Multiple Formats	210
9.3.9	Implementation	211
9.3.9.1	The Notebook View	211
9.3.9.2	Mapping Git Commit with Tutorial Cells	211
9.3.9.3	Propagating Changes	212
9.3.9.4	Recording Interactions in Output Snapshot	213
9.4	System Evaluation Overview	213
9.5	Study 1: Evaluating the Authoring Experience	214
9.5.1	Method	214
9.5.1.1	Recruitment	214
9.5.1.2	Study Task	215
9.5.2	Results	216
9.5.2.1	Overall Quality of the Tutorials	216
9.5.2.2	Easy to Author	216
9.5.2.3	Perceived Benefits for Learners	218
9.6	Study 2: Evaluating the Reading Experience	219
9.6.1	Method	219
9.6.1.1	Recruitment	219
9.6.1.2	Study Setup	219
9.6.1.3	Study Apparatus	221
9.6.1.4	Tutorial Preparation	221
9.6.1.5	Data Collection and Analysis	222
9.6.2	Results	222
9.6.2.1	How do participants engage with the tutorials?	222
9.6.2.2	Are Colaroid tutorials easier to follow along?	223
9.6.2.3	Does Colaroid support more incremental procedure following?	224
9.6.2.4	Does Colaroid lead to better learning outcomes?	226
9.6.2.5	Summary of the Results	226
9.7	Discussion	226
9.7.1	Outlook	228
9.7.2	Limitations	228

9.7.2.1	Limitations of Colaroid	227
9.7.2.2	Limitations of our Evaluation	228
9.8	Conclusion	229
9.9	Acknowledgements	229

Part 4 Conclusions 232

10 Conclusions and Future Directions 233

10.1	Summary of Contributions	233
10.2	Future Work	235
10.2.1	Understanding Heterogeneous Collaboration in Data Science . . .	235
10.2.2	Human-AI Systems for Data Science Programming	236
10.2.3	Making Data Science Accessible for Everyone	236

APPENDICES 238

A.1	Example of Documentation Generation in Themisto	238
A.2	Coding Book for the Interview Transcripts	241
B.1	Tutorial Lists in Formative Study	242

BIBLIOGRAPHY 245

LIST OF FIGURES

FIGURE

1.1	In this dissertation, I aim to enhance the usability of programming environments for data scientists by amplifying three core design characteristics – collaborative, narrative, and exploratory. This objective is achieved through a three-pronged approach: understanding challenges encountered in collaborative data science; improving collaborative computational notebook environments for data scientists; and streamlining the process of knowledge sharing for future data scientists.	2
2.1	Non-traditional methods for evaluating code cells in computational notebooks: (A) The Nodebook extension [208] enforces an ordered flow of cell execution. For example, no matter how many time A.3 is executed, the output of A.4 would always be 17. (B) The Dataflow extension [100] allows users to call a specific version of the variable in other cells’ output. For example, B.2 changed the value of df by renaming columns. B.3 used the value of df from B.2 to run new computations. (C) The Pluto.jl environment [139] implements reactive programming where relevant cells would be automatically updated when changing a function or variable. For example, updating the statement of x in C.1 would propagate changes to C.3. Pluto.jl prevents multiple definitions of global variables across cells. Statement C.2 and C.1 would trigger an error, causing statement C.3 not able to return values. This figure is adapted from [208, 100, 139].	23
3.1	An example of a Jupyter notebook. (1) A custom collaborative extension for users to share their notebook. (2) A notebook cell that contains code to import libraries and load dataset. (3) An output of a shared data frame. (4) A markdown cell that contains narrative text. (5) An output of a visualization	27
3.2	Overview of how participants iteratively explore the house price prediction task in 15-minute intervals. Participants in the shared condition tended to switch between phases more frequently. The initial attempts at modeling occurred earlier in the shared condition.	45
3.3	Post-task questionnaire results for participants in both conditions.	46
3.4	Group S3 coordinated the work well by planning subgoals in the notebook	48

3.5	An example of participants manually annotating a graph for discussion: P14 circled the outlier point in a scatter plot using MS Paint and sent it back to P15 using Slack.	51
4.1	Callisto captures and stores contextual links between discussion messages and notebook elements with minimal effort from users.	60
4.2	Overview of Callisto: (A) The changelog panel shows users' edit histories; (B) The collaborative notebook editor synchronizes edits, runtime variables, outputs, annotations (see G, H), and cursors (see F) among collaborators; (C) The filter button enables the filtering mode (see Figure 4.3); (D) The user panel lists collaborators that are connected to the notebook. Users can navigate to others' cursor locations by clicking on their name; (E) The embedded synchronous chat pane creates connections between messages and notebook content. Messages mapped to the selected cell are highlighted in light green. Users can create explicit references by clicking the magic wand (see J) and then selecting the relevant part of the notebook—for example, to create an annotation reference (see I).	66
4.3	Filter Mode. When filter mode is enabled, it only displays messages and edits that are marked as relevant to the selected cell.	67
4.4	Diff View. Code differences (see A) and output differences (see B) are highlighted in a diff view. The new and old outputs are overlapped for comparison: hovering the mouse over the output will highlight the difference in purple and pink; the slider underneath controls the transparency between new and old output.	71
4.5	Chat Panel. When selecting one message, a snapshot button (see A) will navigate users to the snapshot of the notebook. When selecting two messages, a diff button (see C) will navigate users to the diff view comparing two snapshots (see Figure 4.4). Users can manually refine the links using the edit button (see B).	72
5.1	We replicated the notebook-level descriptive analysis by Rule et al. [157] to the 80 well-documented notebooks on Kaggle. The left side represents the descriptive visualization of the 80 well-documented computational notebooks from Kaggle (noted as Sample A) and the right side represents the descriptive visualization of the 1 million computational notebooks on Github (noted as Sample B). The highly-voted notebooks on Kaggle are better documented compared to the Github notebooks.	85

5.2	The Themisto user interface is implemented as a Jupyter Notebook plugin: (A) When the recommended documentation is ready, a lightbulb icon shows up to the left of the currently-focused code cell. (B – D) shows the three options in the dropdown menu generated by Themisto, (B) A documentation candidate generated for the code with a deep-learning model, (C) A documentation candidate retrieved from the online Application Programming Interface (API) documentation for the source code, and (D) A prompt message that nudges users to write documentation on a given topic.	92
5.3	An illustration of the three different approaches for documentation generation in Themisto.	93
5.4	A code summarization model for the deep-learning-based documentation generation approach via Graph Neural Network (GNN). There are three steps of data pre-processing (1) We first extract text code pairs from existing notebooks. (2) We generate Abstract Syntax Tree (AST) from code. (3) We tokenized each word and translated them into embeddings. And (4), the GNN model architecture.	94
5.5	Results of the post-task questionnaire. Note that the disagrees are not from the same participant.	102
6.1	As users iterate on their data during analysis, they can use DITL to compare data snapshots. Every time users successfully execute code we save a snapshot (A). Users can compare the code using traditional code diffing tools. Users can also use DITL to compare data iterations with interactive visualizations, descriptive statistics, and data preview (B). User can choose three ways to visualize the differences in each column: delta view (C), opacity view (D), and parallel view (E).	116
6.2	We integrate DITL into a simplified data science programming environment that allows data scientists to edit code, inspect data tables, and compare different data tables. This interface shows that the user is browsing a snapshot tagged 1k9i8j where the edit took place at 12:18:17. (A) Users are able to navigate among saved snapshots, compare code differences and output differences, or switch to the current code editor; (B) Users can edit code in the current code editor which automatically saves a new snapshot upon successful execution, or view code changes in a snapshot; (C) Users can switch between the output panel, the data panel, and DITL.	119
6.3	The data panel allows users to inspect a single data table. (A) Users can select saved data frames from the current code snapshot; (B) The data panel shows the distribution of each column; (C) The data panel shows the summary statistics for each column; (D) The data panel shows a sample of rows from the selected data frame.	121
6.4	DITL uses three approaches for rendering data differences: parallel view, opacity view, and delta view.	122

6.5	Participants' responses to the likert scale questions in the post-task questionnaire.	125
7.1	Editing conflicts in real-time collaborative notebooks can be implicit. As shown on the left, one can get an unexpected execution result because the collaborator accidentally changed the shared variable. As shown on the right, PADLOCK helps data scientists resolve editing conflicts in real-time collaborative editing in computational notebooks.	134
7.2	Overview of the three conflict-free mechanisms in PADLOCK	139
7.3	The collaborative workflow for G1 and G2.	162
8.1	PuzzleMe implements two mechanisms for peer assessment: live peer testing and live peer code review. Live peer testing allows students to create and share test cases with peers in real time. Live peer code review intelligently groups students to encourage meaningful code discussions. .	172
8.2	PuzzleMe is a peer-driven live programming exercise tool. The student view shows: (A) a problem description; (B) an informal test that consists of the given conditions (see B1) and the assertion statement (see B2); (C) a code editor where students can work on their solutions; (D) a test library that contains valid tests shared by instructors and other students; (E) the output message of the current solution; (F) access to the instructor's live coding window; (G) access to peer code review; (H) a leaderboard of the number of problems that students have finished; and (I) the number of students who have finished the current problem.	176
8.3	The live code view and the peer code review view. (A) Instructor can enable live coding mode to demonstrate coding in real-time, and use the built-in sketching feature to assist their demonstration; (B) Live peer code review allows students to check other group members' solutions and provide reviews in a chat widget (as shown in C).	177
8.4	An example of three different levels of test cases for one exercise (Problem description: alphabetically sort the given array, <code>names</code> , and assign the output to a variable named, <code>names_sorted</code>). Test cases were manually coded into three levels: 0 if the test case was wrong, meaningless, or duplicated the default case; 1 if the test case did not create new examples of <code>names</code> but added additional checks on the output <code>names_sorted</code> ; 2 if the test case contained new examples of <code>names</code> and <code>names_sorted</code>	183
8.5	Usage logs from the online lecture. With the help of peers, 10 students who had incorrect solutions were able to pass the problem (as indicated in green horizontal lines). Live peer code review helps students identify cavities in their code and inspires them to explore alternative approaches.	187

8.6	Three example solutions that pass the default test. Solution A is a false positive because students did not use the function arguments correctly. Solution B is the most elegant way to solve the problem. Solution C is correct but does not demonstrate an understanding of advanced list operations.	188
9.1	An overview of Colaroid. Colaroid is implemented as a VS Code extension. The user can open the Colaroid Notebook (B) side by side with their main code editor (A). A Colaroid notebook consists of cells. Each cell captures a history state of the codebase, which contains three components — a text annotation area explaining the rationale behind this state (C), a code editor area displaying the state of the code and highlighting the changes compared to the previous state (D), and an output area rendering the HTML display of the history state (E).	205
9.2	Colaroid implements code versioning and change propagating through git. Suppose the author wants to edit the first step (e.g., changing HTML page title) and propagate the change to subsequent steps. Colaroid maps steps with the hash ID of the code commits in Git. As shown in phase 1 and phase 2, Colaroid will first check out the main branch into a new branch named “change” and reset the head to the code version that needs to be edited. By doing this, authors would see commit A loaded in their code editor. Next, tutorial authors can make changes directly in the code editor. Once they confirm finishing the edits, Colaroid will create a new commit for the changes and merge commits in the later steps into the change branch.	212
9.3	Results of the post-task questionnaire in study 1.	217
9.4	We used GitHub Codespaces for participants to access tutorials. All the three types of tutorials are displayed side by side with the main code editor.	220
9.5	We manually coded the screen recording to understand how students interact with the tutorials. All participants used the full 40 mins on the task. Participants’ engagement time with Colaroid tutorials is significantly more than article tutorials (1) and video tutorials (2). In particular, we noticed that some participants (3) gave up on the video tutorials after watching a segment at the beginning of the study.	224

LIST OF TABLES

TABLE

3.1	The tools that respondents have used for programming, communication and project management during collaboration.	32
3.2	Strategies for keeping a shared understanding	33
3.3	Demographics of Participants in Study 2	35
3.4	Collaboration Styles. The notebook ratio is the percentage of cells that one member contributes in the final notebook. The message ratio is the percentage of messages that one member sends in total Slack messages; N/A means the team uses video-based communication (Google Hangouts). The error score is calculated using Root Mean Square Error (RMSE). Lower error scores are better.	39
3.5	Comparing the outcomes from prediction results and final notebooks (mean: \bar{x} , standard deviation: σ). Working in the same notebook encourages groups to explore more solutions and leads to a better result. . .	47
4.1	Purpose of sending a message: reflecting, planning, check-in, cooperation, and out-of-scope.	63
4.2	Relevance between messages, the notebook history, and the final notebook.	63
4.3	Granularity: the level of detail of the referenced elements	63
4.4	Server-side Usage Logs (mean: \bar{x} , standard deviation: σ): (1) Most contextual links were created by inferred references; (2) The two navigating features were used equally to understand past decisions.	74
4.5	Comparing the outcomes from the second stage of the evaluation (mean: \bar{x} , standard deviation: σ). Callisto helps new collaborators achieve a better understanding of an ongoing project.	77
5.1	We identified 9 categories based on the purpose of markdown cells. Note that a markdown cell may belong to multiple categories of contents or none of the categories.	87
5.2	We coded each markdown cell to which data science stage (or task) they belong. We identified 4 stages with 13 tasks out of the data science life-cycle [187]. Note that a markdown cell may belong to multiple stages or none of the stages.	89

5.3	Example output from the model. (Example A) The generated text well describes the code. (Example B) The generated text vaguely describes the code. (Example C) The generated text is poorly readable, but still captures the keywords of the descriptions.	96
5.4	Demographics of participants	98
5.5	Performance data in two conditions (M: mean, SD: standard deviation): the task completion time (secs), participants' satisfaction with the final notebook (from -2 to 2), graded notebook quality, number of markdown cells, and number of words. In particular, participants spent less time to complete the task in the experimental condition than the control condition ($p = .001$); participants were more satisfied with the final notebook in the experimental condition than the control condition ($p = .04$).	101
5.6	Usage data of the plugin in experimental condition. The results indicate that participants used the plugin for recommended documentation on most code cells (86.11%). For markdown cells in the final notebooks, 46.90% were directly adopted from the plugin's recommendation, while 41.24% were modified from the plugin's recommendation and 11.86% were created by participants from scratch.	104
7.1	We recruited students and alumni from data science programs in our institution. There are 14 participants who finished the paired session, 8 of them chose to participate in the individual session (S2), and 7 of them chose to participate in the group session (S3). Grads. refers to graduate students; Ugrd. refers to undergraduate students.	150
7.2	Features that participants have used for tasks in each study.	157
7.3	For each scenario, the research team solicited three potential cases for editing conflicts and run through participants' notebooks through these cases.	159
8.1	Instructors' course demographics in formative interviews.	173
8.2	The four lab sections were randomly assigned into the treatment condition or the control condition.	182
8.3	The number and quality of test cases students wrote in E1 (mean: \bar{x} , standard deviation: σ). The number of students who improved code (calculated by completion status) after group discussions in E2 (total: N). Our comparison suggests that the number of test cases is significantly different in the two conditions ($p = 0.002$, Mann-Whitney U test with power = 0.98); the number of students who improved code is also significantly different in the two conditions ($p = 0.02$, proportions z-test given the binary data type).	184
8.4	Participants' background in Study 3.	189
8.5	Use cases of two PuzzleMe features—live peer testing and live peer code review—in various programming topics.	191

9.1	For study 1, we recruited 10 teaching assistants and senior students who are experienced in web programming.	214
9.2	Perceptions of the three tutorials. Participants rated their agreement with nine questions on a scale from 1 (strongly disagree) to 5 (strongly agree). (M: mean, SD: standard deviation).	231
9.3	Exploratory Analysis Results.	232
A.1	Example notebook (House Price) (T - Markdown cells created by Themisto only, C - Markdown cells co-created by Humans and Themisto, H - Markdown cells created by Humans only).	240
A.2	Example notebook (Covid Prediction) (T - Markdown Cells Created by Themisto Only, C - Markdown Ceells Co-created by Humans and Themisto, H - Markdown Cells Created by Humans Only).	241
A.3	Coding Book for the Interview Transcripts	242
B.1	Tutorial Lists in Formative Study (1)	243
B.2	Tutorial Lists in Formative Study (2)	244
B.3	Tutorial Lists in Formative Study (3)	245

LIST OF APPENDICES

A	Appendix for Chapter 7	239
B	Appendix for Chapter 9	243

LIST OF ACRONYMS

HCI	Human-Computer Interaction
IDE	Integrated Development Environment
AWS	Amazon Web Services
UI	User Interface
OS	Operating System
FAIR	Findable, Accessible, Interoperable, Reusable
IT	Information Technology
SVM	Support Vector Machine
RMSE	Root Mean Square Error
VCS	Version Control System
OT	Operational Transformation
URL	Uniform Resource Locator
AI	Artificial Intelligence
ML	Machine Learning
GNN	Graph Neural Network
CSCW	Computer-Supported Cooperative Work
GUI	Graphical User Interface
AIOP	AI Operator
AST	Abstract Syntax Tree
NLP	Natural Language Processing

API Application Programming Interface

HTTP Hypertext Transfer Protocol

REPL Read-Eval-Print Loop

MOOC Massive Open Online Course

GSI Graduate Student Instructor

CCN Cyclomatic Complexity Number

PCB Printed Circuit Board

USD United States Dollars

NLP Natural Language Processing

RBT Revised Bloom's Taxonomy

CEO Chief Executive Officer

VP Vice President

ABSTRACT

With the expanding landscape of data science, it becomes increasingly crucial for data scientists to foster collaborative practices to enhance productivity. Computational notebooks enable data scientists to run and share exploratory analysis. The notebook design facilitates the rapid iteration of code chunks and inspection of intermediate results through an interactive execution interface. The combination of exploratory code and human-readable explanations creates computational narratives that are effortlessly shared and reproduced. This dissertation investigates the challenges associated with the ineffective usage of computational notebooks for collaboration, and proposes possible interactive designs to improve the notebooks for both data science professionals and educational settings. This dissertation unfolds into three parts aimed at enhancing the usability of data science programming environments. Firstly, to discern the obstacles faced in collaborative data science, I conducted mixed-methods inquiries to understand how data scientists currently utilize real-time collaborative editing in computational notebooks. Based on the findings, I proposed a series of design approaches for addressing the challenges, including providing contextually-linked discussions, leveraging AI-assisted data science code documentation, visualizing data changes, and lightweight editing control mechanisms that can make computational notebooks more narrative. Lastly, my dissertation illuminates the distinct collaboration needs required to support data science teaching and learning. I explored a series of designs that support live peer evaluation for testing and reviewing in-class programming exercises and authoring explorable multi-stage tutorials. In summary, my dissertation contributes to improving the usability and effectiveness of computational notebooks for data science collaboration and learning, facilitating their ability to create and share computational narratives.

CHAPTER 1

Introduction

Over the past decade, data science plays an increasingly important role across many sectors of the economy. [116]. Data science uses statistical techniques to interpret data from various sources and make data-driven decisions. This can help businesses identify new opportunities, optimize marketing strategies, and increase efficiency. In areas like public health and environmental science, data science can help solve critical societal challenges, from predicting disease outbreaks to moderating climate change. Effective collaboration and communication are crucial for data scientists working on complex large-scale problems. For example, data scientists work across disciplines with a variety of stakeholders in practice [204]; data scientists collaborate with both internal and external teams throughout the analysis pipeline [138]; citizen data scientists collaborate in an open source manner to collectively explore topics of shared interests [80, 35].

However, it is worth noting that collaboration remains a challenge for many data science tools [181]. Although there are many mature collaborative tools for software engineering (e.g., version control and social coding tools), they may not be as effective for data scientists due to some key differences between the two fields [204, 181]. For instance, data science work tends to be more exploratory, which can result in lower-quality code and incomplete documentation. Furthermore, it is difficult to manage the exploration history when iterating ideas quickly [91]. These issues can be amplified in a collaborative setting, where maintaining a shared understanding of past design decisions across team members becomes crucial.

On the other hand, computational notebooks such as Jupyter Notebooks have been widely used for data-related work among academia, industry, and data science education [137]. These unique programming environments allow programmers to integrate code, tables, graphs, and prose into a single, cohesive narrative referred to as a *computational narrative*.

Interactive Programming Interfaces for Data Science Collaboration and Learning

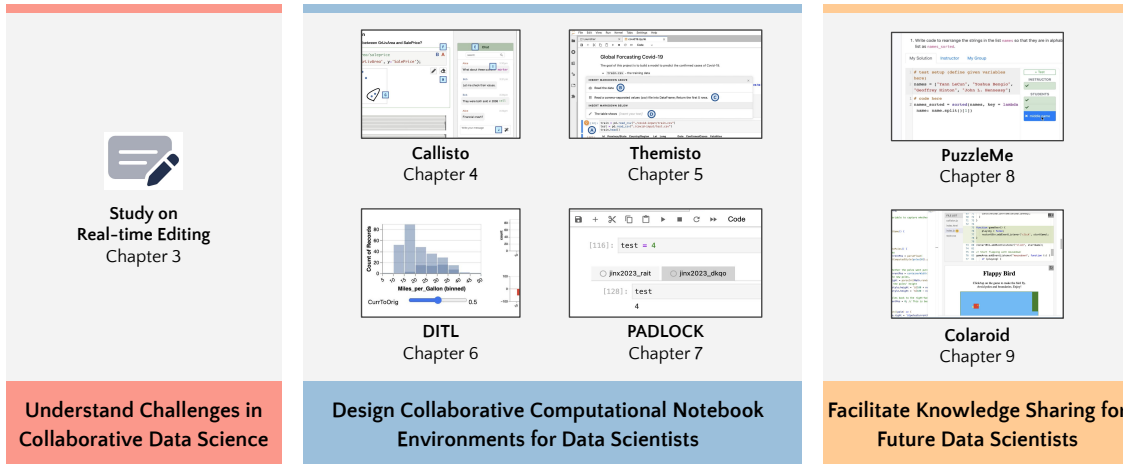


Figure 1.1: In this dissertation, I aim to enhance the usability of programming environments for data scientists by amplifying three core design characteristics – collaborative, narrative, and exploratory. This objective is achieved through a three-pronged approach: understanding challenges encountered in collaborative data science; improving collaborative computational notebook environments for data scientists; and streamlining the process of knowledge sharing for future data scientists.

Computational notebooks possess three key characteristics that make them effective for data science programming — they are collaborative, narrative, and exploratory. Firstly, computational notebooks are a collaborative medium, allowing data scientists to work together by creating and sharing computational narratives that encompass not only the code assets but also the thought processes, ideas, and rationales behind the code. Recent computational notebook platforms like JupyterLab [17], Deepnote [16], and Observable Notebook [18] have revolutionized the collaborative experience of working together on computational notebooks through real-time technologies. These platforms enable collaborators to synchronize edits in real-time and execute code on a shared interpreter. Second, computational notebooks are narrative, embodying the practice of the *literate programming* paradigm pioneered by Knuth [96]. By interweaving source code with natural language descriptions, data scientists can write programs as if they are crafting an essay, allowing for the convenient sharing of thoughts, hypotheses, and explanations. Lastly, computational notebooks support code exploration by executing code through an interactive kernel, providing the Read-Eval-Print Loop (REPL) execution paradigm. Jupyter notebooks consist of “cells” which typically contain small chunks of code or narrative

text in the Markdown format. Users can execute cells (usually from top to bottom) and observe their outputs, which may include visualizations, data frames, or rendered narrative text. This structure enables computational notebooks to be a live and exploratory programming environment, allowing consumers of the notebook to actively engage with the content by reproducing results and experimenting with alternative approaches.

However, studies have identified several limitations with computational notebooks [157, 158, 91]. For quick exploration, data scientists sometimes generate messy and informal notebooks, which can be difficult for others (or even the author) to read later on [157]. Data scientists have to use strategies like actively pausing the experiment to curate and clean notebooks into narratives, which may hinder the exploration process [91]. The tension between quick exploration and instructive explanation can be contextually sensitive depending on how exploratory and open-ended the task is. These problems may become more pronounced when the process involves multiple stakeholders — for example, when data scientists work closely together or when using computational notebooks for teaching and learning data science programming.

In my dissertation, I re-envision the workflow and interfaces for data science programming environments, adhering to the design characteristics of collaborative, narrative, and exploratory. I first conduct mixed-methods inquiries to understand challenges in collaborative data science. This work has identified the advantages of working closely together on a real-time shared notebook over working on individual notebooks, such as creating a shared context and reducing communication costs. In the meanwhile, this work has found several remaining challenges with synchronous editing, such as producing less organized notebooks. To solve the challenges with collaborative data science, I build and deploy systems for improving collaboration for data science practitioners, which include: (1) designing interactive techniques for capturing connections between discussions and notebook entities, (2) leveraging AI techniques for summarizing data science code, (3) using visualization techniques to explain the impact of code changes, and (4) designing editing control mechanisms to avoid conflict edits in real-time collaboration. Lastly, this dissertation also facilitates the knowledge-sharing experience in programming through building systems that support: (1) live peer evaluation for in-class programming exercises, and (2) authoring explorable multi-stage tutorials.

Specifically, my dissertation is driven by an overarching research question: **How can we design interactive programming environments that support data scientists and learners in communicating and accomplishing various data-related tasks?**

In the next few subsections, I will briefly delineate the three-pronged approach of my thesis work: (1) understanding challenges in collaborative data science, (2) designing collaborative computational notebook environments for data scientists, and (3) facilitating knowledge sharing for future data scientists.

1.1 Understand Challenges in Collaborative Data Science

Computational notebooks (like Jupyter) are often used by data scientists doing exploratory work because they enable interactive and incremental development, allow inline documentation, and provide immediate visual output. Collaborative notebook editing platforms like Jupyter, Google Colab, and Deepnote allow collaborators to synchronize edits in real time and execute code on a shared interpreter. Understanding how these tools fit in with the unique workflows of data scientists can help improve the design of future data science collaborative tools. Through a mixed-method study [181], I examined two main approaches data scientists use for collaboration: (1) working on individual notebooks and updating work asynchronously, and (2) working closely together on a shared notebook with edits synchronized in real-time.

1.1.1 A Mixed-Methods Study to Understand Real-Time Collaborative Notebooks

While real-time computational notebooks enable new modalities for collaboration, synchronous editing comes with its own challenges and may not always improve work efficiency. To motivate the problem, I conducted a mixed-method study to understand how synchronous notebook editing may change the way data scientists collaborate in computational notebooks [181] (chapter 3). I first conducted a formative survey to understand the common tools and mechanisms that data scientists use for collaboration and communication. The results suggest two approaches for data scientists to collaborate: (1) the traditional collaboration setting where team members work on individual notebooks and update the progress asynchronously by sending each other the notebooks; (2) the emerging collaboration setting where team members work closely together on a shared notebook where all the edits are synchronized in real-time. To further explore the differences between these two approaches, I carried out an observational study with pairs of

data scientists working remotely to solve a predictive modeling problem. This work has created a taxonomy of common collaboration styles in data science and identified several advantages of synchronous editing, such as creating a shared context, encouraging more exploration, and reducing communication costs. Despite all the benefits, synchronous editing comes with its own challenges and may not always improve work efficiency. This work further discovered several challenges with synchronous notebook editing such as producing messy and less organized notebooks, causing conflict edits when there is no strategic planning.

Meanwhile, this work suggests that the collaboration needs and challenges may shift based on a number of factors. For example, the type of data science problem, the expertise of collaborators [138, 132, 55], team size, the synchronicity of collaboration, and whether the purpose of collaboration is for productivity or learning [177, 32], may all affect how users perceive the collaboration experience.

1.2 Design Collaborative Computational Notebook Environments for Data Scientists

Beyond understanding the challenges in collaborative data science through studies, I also build systems to understand and expand the design space to aid collaboration. Due to the exploratory nature of the work, data scientists benefit from documentation that covers the explanations of the process, reasoning about the decision-making, and interpretation of the results. Unfortunately, data scientists often write messy and drafty analysis code in computational notebooks as they need to quickly test hypotheses and experiment with alternatives [157]. All these tensions can be amplified in a collaborative setting where it is important to keep a shared understanding of past design decisions across team members. This work explored the space by studying how interactive techniques such as providing contextually-linked discussions [184], leveraging AI-assisted data science code documentation [183], visualizing data changes [179], and lightweight editing control mechanisms [185] can make computational notebooks more narrative.

1.2.1 Contextualizing Shared Notebooks with Discussions

To help collaborators better understand the rationale of a shared notebook, I built Callisto [184], a Jupyter extension with the ability to connect discussion messages with the shared

notebook elements with minimal effort from users (chapter 4). When teams of data scientists collaborate on computational notebooks, their discussions often contain valuable insight into their design decisions, but are very long and disconnected from the computational notebook. Callisto extends the Jupyter platform in several ways: (1) it enables users to connect discussions with elements in the shared notebook, including code, output, cells, or edits, (2) it then leverages these connections to make it easier to navigate discussions and notebook content — for example, to find discussions about a particular part of the notebook. A two-stage evaluation study showed that Callisto can ease user onboarding to new notebooks by helping them understand the design rationales of its authors.

1.2.2 AI-Assisted Data Science Code Documentation

Contextualizing notebook elements with discussions helps data scientists to make sense of a shared notebook that is constructed through *synchronous* editing. To improve the documentation of messy analysis notebooks that are written *asynchronously*, I further explored AI-assisted documentation [183] (chapter 5). Existing NLP and AI literature suggested deep learning models that can “translate” code into natural language descriptions as if translating two languages. However, these code summarization models are examined in the context of software engineering. As I discovered in a formative study, data science documentation captures more aspects than software engineering documentation (e.g., explaining the code, interpreting results, and reasoning about hypothesis). Thus, I built Themisto, a Jupyter extension that combines a deep-learning approach with a query-based approach to retrieve online API documentation for source code, and user prompt approach to nudge users to write documentation [183]. Our evaluation showed that automated documentation generation techniques [109] reduced the time for writing documentation, reminded participants to document code they would have ignored, and improved participants’ satisfaction with the final notebook.

1.2.3 Improving Awareness with Data Changes

Callisto and Themisto enable data scientists to better understand the *code changes*. I further argue that understanding the iterative *data changes* that the code produces should be as important as code changes throughout an analysis. Code differences do not always reveal data differences. For example, removing missing values from one column of a

dataset may also affect the distributions of the dataset's other columns. Data scientists currently need to take the initiative to write additional code to browse or plot the data in order to track the effect of code changes. I explored the idea of visualizing differences in datasets as a core feature of exploratory data analysis, a concept called Diff in the Loop (DITL) (chapter 6). Using a table-based diff view, users can either compare different datasets or compare the same dataset at different snapshots. The evaluation shows that DITL helps data scientists understand the impact of code changes in debugging, gain insights into data through comparisons, and improve awareness in collaboration.

1.2.4 Resolving Editing Conflicts in Real-Time Collaboration

To help data scientists avoid interference with each other's work in a shared notebook, I proposed a real-time collaborative editing framework (PADLOCK) to address the challenges [185] (chapter 7). PADLOCK leverages the context of data science development to provide three domain-relevant mechanisms to improve collaboration on computational narratives. The first, cell-level access control, prevents collaborators from viewing or editing a collection of cells. This mechanism aims to allow ad-hoc locking of code cells to support volatile collaboration patterns. The second mechanism, variable-level access control, extends the access control from cell-level to shared variables. This mechanism is designed to prevent implicit editing conflicts and allow collaborators to protect important shared variables. The third, parallel cell groups, leverages the familiar programming concept of encapsulation through scoping with the Jupyter cell user interface control. This mechanism allows individuals to pursue exploratory solutions while not having to be concerned about interference with others. The evaluation of PADLOCK has shown that these mechanisms can effectively prevent editing conflicts in shared notebooks; and they support a wide range of ad-hoc and volatile collaborative workflows.

1.3 Facilitate Knowledge Sharing for Future Data Scientists

The growth of the data science industry has led to a growing demand for introductory programming and data science education. Computational notebooks have seen widespread adoption as a medium for creating rich descriptive documents about code. In particular, such notebooks have been used by data science instructors to create code tutorials and

exercises [103, 44]. They support a kind of exploration and tinkering that is central to “learning by doing” [97] — as in a computational notebook, a code cell can be modified and executed, allowing readers to explore how changes to the code influence the results. How can we design systems to support collaborative teaching and learning data science programming? I explored a series of designs that support live peer evaluation for testing and reviewing in-class programming exercises [177] and authoring explorable multi-stage tutorials [180].

1.3.1 Real-Time Sharing for In-Class Programming Exercises

In-class programming exercises are small-scale programming exercises for students to practice during lectures or labs. In-class exercises allow students to receive real-time feedback from instructors and peers. This immediate feedback can help students correct mistakes and misconceptions on the spot, reinforcing the correct programming concepts. Yet, a key challenge that remains is the insufficient support provided by existing teaching platforms and programming environments for integrating live peer assessment, especially within the context of in-class programming exercises. I designed PuzzleMe [177], a novel programming notebook that allows instructors to conduct engaging in-class programming exercises (chapter 8). PuzzleMe leverages peer assessment to support a collaboration model where students provide timely feedback on their peers’ work. In particular, it supports live peer testing, which can improve students’ code robustness by allowing them to create and share lightweight tests with peers. I conducted a two-week deployment study in introductory Python courses at the University of Michigan. The evaluation proves that the sharing of the test cases and the code solution encourage students to write useful test cases, identify code problems, correct misunderstandings, and learn a diverse set of problem-solving approaches from peers.

1.3.2 Authoring Explorable Multi-Stage Tutorials

In many domains of programming, code is developed through a cyclical process of editing, compiling, and running the code. This resulted in the granularity of the changes do not allow the code to be executed piecewise. Thus, many programming tasks do not fit for the REPL model in computational notebooks. Moreover, in authentic practice, these changes may take place between various code assets. The implementation of a single feature might be split across many places in the code and difficult to isolate into a single cell.

As a result, tutorial authors often need to manually curate these steps from their coding environment into a static text document. It is a tedious process for learners as well to follow these static text tutorials and reproduce the process in their local coding environment. What if we could harness the advantages of computational notebooks to aid programmers in documenting their incremental code construction for a wider range of programming tasks? This dissertation explores how we can extend the computational notebook design to support authoring explorable multi-stage tutorials [180]. Specifically, I am interested in bridging the divide between the authentic computing environment and the tutorial editing environment. This would enable authors to capture their authentic coding process and transform it into a narrative, while also allowing learners to load the process from a tutorial into their own local computing environment. I designed Colaroid, a VS Code extension that facilitates the creation of high-quality multi-stage tutorials [180] (chapter 9). Colaroid tutorials are augmented computational notebooks that showcase snapshots of a project through snippets and outputs. The extension highlights source code differences and provides complete source code context for each snippet. Additionally, users can load and experiment with any stage of the project within a linked IDE. Through two laboratory studies, I have discovered that Colaroid provides a simple and efficient means of developing multi-stage tutorials, while also offering benefits to readers when compared to video and web-based tutorials.

1.4 Thesis Statement

Effective collaboration and communication are essential for data scientists working on complex large-scale problems in both professional and learning settings. Through the forementioned projects, I aim to demonstrate the following thesis statement:

By better understanding the challenges in various data science collaboration tasks, we can design tailored programming environments that are more collaborative, narrative, and exploratory to improve the quality and efficiency of collaboration.

More specifically, collaborative programming environments that support seamless sharing of coding assets, improve the efficiency and awareness of teamwork; narrative environments that enable rich, easy-to-create, and up-to-date explanations, facilitate communication and shared sense-making of code and data changes; exploratory environments

that allow data scientists to easily reproduce an authentic analysis and build on top of it, encouraging the active exploration of the data science process.

1.5 A Note on Authorship

I am the primary author of the research reported in this dissertation. However, this work is done in collaboration with my advisors Steve Oney and Christopher Brooks from UMSI, my colleagues Yan Chen, Ashley Zhang, Zihan Wu, and others from UMSI, my mentor Steven Drucker and other colleagues from Microsoft Research, and my mentor Dakuo Wang and other colleagues from IBM Research. Chapter 3 on the data scientists' collaboration styles is published at CSCW'19 [181] and is co-authored with Steve Oney, Christopher Brooks, and other colleagues from UMSI. Chapter 4 on creating contextual links between discussions and shared computational notebooks is published at CHI'20 [184] and is co-authored with Zihan Wu, Steve Oney, and Christopher Brooks. Chapter 5 on AI-assisted data science code documentation is inspired by the AutoAI work from IBM Research. This work is published at TOCHI'21 [183] and is co-authored with my mentor Dakuo Wang and other colleagues at IBM Research. Chapter 6 on visualizing data changes is informed by the live data science programming environment Glinda [47] from Microsoft Research. This work is published at CHI'22 [179] and is co-authored with my mentors Steven Drucker, Rob DeLine, and other colleagues at Microsoft Research. Chapter 7 on resolving editing conflicts in shared computational notebooks is currently under submission and is co-authored with Zihan Wu, Steve Oney, and Christopher Brooks. Chapter 8 on in-class exercise notebooks is published at CSCW'21 [177] and is co-authored with Yan Chen, John Joon Young Chung, Steve Oney, and Christopher Brooks. Lastly, chapter 9 on explorable tutorial authoring environments is published at CHI'23 [180] and is co-authored with Andrew Head, Ashley Zhang, Steve Oney, and Christopher Brooks. In chapters 3-9, I use the first-person plural (we/our) to indicate co-authorship.

CHAPTER 2

Related Work

To contextualize the dissertation, I draw upon prior research on data science work practices and computational notebooks. This helps to situate the design principles within the broader context of existing research and establishes a strong theoretical foundation for the subsequent chapters.

2.1 Data Science Programming

Data science refers to the process of extracting knowledge and insights from data [50]. Data scientists use computational methods to collect data, understand data, and help business stakeholders to make decisions [50]. The field of data science has grown rapidly over the last decade amidst the rise of big data and breakthroughs in technologies like machine learning that expand our capabilities for understanding data [43]. According to a survey of the United States workforce on LinkedIn [4], the demand for data scientists will continue to increase as more industries (e.g., finance, business, healthcare) adopt big data to make business decisions.

2.1.1 Data Science Workflow

The process of doing data science has been categorized and discussed among statisticians, computer scientists, HCI researchers, and others (e.g., [124, 86, 93, 71, 118]). O’Neil and Schutt distinguished the data science process into several iterative phases [124]:

1. *Collecting* data from a variety of sources (e.g., emails, logs, and medical records)
2. Building and using pipelines for *data munging* (e.g., joining, scraping, and wrangling)

3. *Cleaning* data to ensure its validity and accuracy for analysis (e.g., manipulating duplicates, filtering outliers, and tuning missing values)
4. *Exploring and hypothesizing* the relationship between variables using different techniques (e.g., generating statistical summary, plotting pairwise relationships)
5. *Applying machine learning algorithms* or statistical models based on the type of problems (e.g., k-nearest neighbor, linear regression, and naive Bayes)
6. Finally *interpreting and communicating* results to different audiences (e.g., managers, co-workers, and clients)

Not all stages are required depending on the type of data science task. The exploration process can be non-sequential, as feedback from later stages may result in additional work in earlier stages. Based on the workflow, Zhang et al. studied how different roles of data science workers collaborate in practice [204]. They identified five distinguished roles of data science workers (including engineers, managers, researchers, communicators, and domain experts) and aligned their interactions and collaboration practices with the data science workflow. They found two types of documentation practices. One type of documentation is for explaining and collaborating with data science colleagues, which is usually inline with the source code or reported in document-editing tools during the analysis. The other type of documentation is for delivering to clients, which are usually presentations or reports after the analysis. In addition, their findings highlighted the importance of documentation for both data and code artifacts, where the former is often absent in practice and the latter can sometimes miss details on lower-level decision-making.

2.1.2 Research Programming

Research programming is another activity that is related to data science, mainly used by the field of science and engineering. *Research programming* describes any programming activities that seek to obtain insights from data [71]. Guo characterized research programming into four main phases, including preparation of the data, analysis, reflection, and dissemination of results. The preparation phase includes acquiring, reformatting, and cleaning data. The analysis phase and reflection phase can take place alternatively to explore alternatives and make comparisons. The dissemination process involves writing reports and deploying executable code. Guo highlighted several challenges related to documentation:

- In the analysis phase, programmers suffer from data management problems to track the provenance of repeatedly generated results and modified scripts;
- In the reflection phase, programmers often take notes in both physical and digital formats while both formats are difficult to organize and connect back to the original context;
- In the dissemination phase, programmers face the challenge to consolidate all the notes, scripts, emails, output files to aid report writing.

Followed by the discussion on exploring alternatives and making comparisons, Liu et al. [110] studied how researchers make analytic decisions and report the decision-making process in end-to-end data analysis. They used analytic decision graphs to visualize the decision processes throughout the analysis. They summarized rationales for analytic decision-making (e.g., methodological concerns, insights from prior work, constraints in data). They also identified reasons for conducting alternative analysis (e.g., opportunism, robustness, contingency).

2.1.3 Exploratory Programming

Data science is one type of exploratory programming. Exploratory programming refers to the practice of writing code to explore, experiment, and iterate with different ideas in order to solve an open-ended problem. Kery and Myers summarized five common characteristics of exploratory programming [89]. First, exploratory programming can take place in a variety of scenarios including learning programming through play, creative digital art and music, data science, and software engineering. Second, code quality tradeoffs commonly exist in exploratory programming when programmers need to sacrifice code quality during the exploration phase to focus on iteration. Third, usability factors such as *the Closeness of Mapping* (the close mapping between concepts in task domains and code representations) and *Viscosity* (the ease of adding and reverting changes to an existing program) can affect the work efficiency of exploratory programming. Fourth, the exploration process often involves repeated changes to input, parameter, or code snippets. Lastly, collaboration can be extremely challenging in exploratory programming.

2.1.4 Collaboration in Data Science

Prior research has found that data scientists in software companies often work collaboratively [93]. For example, some data science teams have adopted a triangular structure where they divide the task into collecting data, cleaning data, and analyzing data. However, collaboration in data science can be challenging. Transferring findings from data science work to business actions requires successful communication between data scientists and stakeholders who are usually non-technical professionals [135]. Kandel et al. [86] revealed that collaboration between data scientists rarely happens in domains like marketing and finance. One major reason is that the diversity of tools and programming languages has made it laborious to share intermediate code, especially when it is not well documented. Correspondingly, Kery and Myers [89] addressed the difficulty of maintaining a shared understanding of the exploration progress since the intermediate code and data artifacts can be experimental and messy. Nonetheless, with the growing demand for data analysis, efficient collaboration between data scientists will become increasingly important and challenging. Building upon the work of Kandel et al. [86], and Kery and Myers [89], my dissertation further investigates the benefits and trade-offs of mechanisms that enable real-time collaboration and communication in data science work.

2.2 Computational Notebooks

Researchers and practitioners have long explored approaches for creating and sharing digital documents for data analysis (e.g., [73, 171, 36]). Establishing a common format for documenting data analysis can make it easier to present, reproduce, share, and collaborate. One approach is to capture digital assets (e.g., code, output, documentation) and computational environments (e.g., browsing, UI interaction, file versioning) from the OS level [73] — tracking any computing activities such as browsing histories and note editings. Another approach integrates digital components (e.g., text-based lab notes, emails, web pages) into a combined entry [171]. Yet, the most popular tools for data scientists are computational notebooks — web-based platforms that allow users to write and execute code, inspect output, and integrate text annotations, figures, interactive visualization, and other rich media (e.g., Apache Zeppelin¹, Spark Notebook², Observable³, and Jupyter

¹<https://zeppelin.apache.org>

²<http://spark-notebook.io>

³<https://observablehq.com>

Notebook).

2.2.1 Jupyter Notebook

Project Jupyter evolved from IPython [136], a terminal-based interactive shell that originally designed for creating interactive visualizations for scientific computing. Wrapping IPython as the kernel, Project Jupyter is designed as a web-based platform for authoring a single document that combines code cells and intermediate results. The evolution of Project Jupyter is influenced by the rise of data science. Data science is exploratory and fluid, and the process benefits from creating reproducible computational narratives for iterative exploration and interactive inspect of intermediate results. Jupyter Notebook is an open source project that is also extensible through optional add-ons. As Figure 9.1 shows, Jupyter notebooks consist of “cells” — typically small chunks of code or narrative text in the Markdown format. Users can execute cells (typically, but not necessarily, from top to bottom) and observe their outputs, which can include visualizations, data frames, or rendered narrative text.

Most other notebook platforms have a very similar user interface to Jupyter but differ in the programming languages they support (Jupyter uses Python by default but its architecture can support other languages). Other computational notebook platforms include Observable (which uses JavaScript), RStudio⁴ (which uses R Markdown), Wolfram Notebooks⁵ (which uses the Wolfram Programming Languages), and Zeppelin (which allows multiple programming languages to be used in the same notebook). Some notebook platforms use a different computational architectures. For example, some notebook platforms enable *reactive* notebooks that automatically run cells when necessary (as opposed to requiring that users run cells manually, as Jupyter does). For example, Observable notebooks run cells in “topological” order—data dependencies between cells are tracked and changing a cell automatically re-runs other cells that depend on its result. Compared to other notebook services, Jupyter has a larger community of users given its’ long history and prevalence among different contexts. Jupyter also has more customized extensions because of its large community.

⁴<https://www.rstudio.com/>

⁵<http://www.wolfram.com/notebooks/>

2.2.2 The Use of Computational Notebook

Several studies have investigated the effectiveness of computational notebooks in various contexts. This section provides an overview of previous research on (1) the advantages and challenges associated with computational notebooks, (2) the sharing and recycling of computational notebooks, and (3) the utilization of computational notebooks for teaching and learning.

2.2.2.1 Working with Computational Notebooks

Computational notebooks improve the data science workflow compared to traditional script programming. Subramanian et al. [169] conducted a qualitative inquiry to compare the two common modalities in data science programming — computational notebooks and rigid scripts. They found that although scripts are most commonly used in data science programming, and are perceived to be formal and reliable, computational notebooks better serve the exploration needs and suit the actual data science workflow. Randles et al. [148] investigated how Jupyter notebooks can be used for open science under the principles of Findable, Accessible, Interoperable, Reusable (FAIR). In fact, some academic venues encourage paper authors to include notebooks with their submissions (e.g., the Distill Journal [49] in the area of machine learning).

However, the flexible structure of computational notebooks can lead to several issues. When the problem gets complex, data scientists tend to write lower quality code, leave documentation incomplete, change the execution order, or accidentally overwrite important analyses while iterating on different ideas. Kery et al. studied how professional data scientists used Jupyter notebooks in their daily work to create computational narratives [91]. They found that computational notebooks can become easily unorganized if data scientists do not manage and keep track of the variants they explored. Data scientists have to use strategies like actively pausing the experiment to curate and clean notebooks into narratives, which may hinder the exploration process. Rule et al. conducted a large-scale analysis of over 1 million open-source computational notebooks and found that only one in four held explanatory text [157]. For quick exploration, data scientists sometimes generate messy and informal notebooks, which can be difficult for others (or even the author) to read later on [157]. The tension between quick exploration and instructive explanation can be contextually sensitive depending on how exploratory and open-ended the task is. Chattopadhyay et al [30]. categorized nine pain points with computational notebooks, which include setup (loading and cleaning data), exploring and analyzing,

managing code, reliability, archival, security, sharing and collaboration, reproducing and reusing, and delivering to products. Singer [165] examined the complexity in the context of the Jupyter Notebook framework and divided the complexities into incidental complexities and intrinsic complexities. The incidental complexities refer to the defect in the user interface design. For example, the code cell outputs are still retained after the client disconnected from the kernel. This decoupled persistence would cause confusion that the runtime state is available to generate the output. The intrinsic complexities refer to challenges that require more engineering effort. For example, the current notebook design does not support modularity, meaning that one notebook's code can not be imported to another notebook.

2.2.2.2 Sharing and Reusing Computational Notebooks

Computational notebooks facilitate sharing and communicating the story of data analysis. Källén et al. [85] conducted a large-scale analysis of code cloning in Jupyter notebooks hosted on GitHub. They found that cell-level cloning is common in Jupyter notebooks, where type 1 clones (exact copy) are seen more often than type 2 clones (with variable renaming) and type 3 clones (with few statements changing). A qualitative review of the most common clones suggests that these clones take place unintentionally, which indicates that data scientists may naturally write similar code snippets without copying from others. Zhang et al. [205] proposed a novel machine learning model to represent the code and surrounding comments in Jupyter notebooks. They applied the model to the corpus of Jupyter notebooks on GitHub to answer open questions about sharing the reusing computational notebooks. They found that academic notebooks contain more on data exploration and less on developing models. They also found that including scientific notebooks can increase the impact of the publications.

Several works have examined the practice of sharing computational notebooks in online communities. For example, Cheng and Zachry [35] studied the practice of sharing and contributing the data analysis stories on Kaggle — the most popular data science online community. They interviewed data scientists about their motivation, practices, and challenges when participating in the Kaggle competition and contributing to community knowledge. Tauchert et al. [174] studied the motivation and usage for organizers to host Kaggle competitions. They characterized the practice as crowdsourcing data science where competitions can inspire organizers to discuss with participants and learn state-of-art approaches.

On the other hand, keeping a shared understanding of past explorations across team members can be challenging when working on notebooks together. Kery et al. [89] highlighted the needs for sharing and group exploration in exploratory programming. In particular, it can be difficult for groups to manage notebooks when multiple people are making exploratory changes to the source code and may leave poor notes to document the rationales. Koesten et al. [98] examined collaborative practices with structured data through an interview study. They revealed tools on a spectrum that spans across the data science life cycle, which includes data portals for storing and sharing datasets (e.g., CKAN, Figshare), data analysis tools (e.g., Google Sheets, Jupyter Notebooks), wiki-based platforms (e.g., Wikidata), and versioning tools (e.g., OSF, GitHub). They identified several user needs for collaborative data tools, such as change control, supporting conversation, and allowing custom data access. This dissertation [181] studied the use of computational notebooks for real-time collaboration. I found that synchronous editing can improve collaboration by creating a shared context and reducing communication costs. I also pointed out that synchronous editing can potentially harm collaboration by activity interference and lack of documentation.

2.2.2.3 Teaching with Computational Notebooks

Given the benefits of supporting exploratory programming, computational notebooks are widely adopted not only in data science professional work, but also in data science education. A handful of resources are available to guide instructors to the best practices of using Jupyter Notebook in their classrooms (e.g., the open book about using Jupyter for teaching and learning [24]). Kross and Guo [103] interviewed practitioners who taught data science and found that Jupyter notebooks have been widely used by instructors to deliver course materials. Instructors and students benefit from Jupyter notebooks by easily writing computational narratives with a low cost for setting up an environment. However, Johnson [84] pointed out the technical challenges and pedagogical considerations that may prevent instructors from using the Jupyter notebooks. In particular, he mentioned technical challenges such as the hidden state in the notebook that may confuse students, the limited debugging capabilities, and the lack of close integration with learning management systems. He also argued that programming in notebooks does not facilitate software engineering best practices.

2.3 Improving Computational Notebooks

The popularity of computational notebooks has led to a growing effort to create more effective and adaptable notebook systems and to develop new interaction techniques in both academic and industrial settings. Lau et al. conducted a comprehensive review of 60 notebook systems across academia and industry, analyzing their features and the typical workflow for computational notebooks, including importing data, editing code and text, running code to generate cell outputs, and publishing notebooks [107]. In this section, I examine the design goals of notebook features and organize the discussions around several themes: (1) lowering the barriers to data science programming, (2) encouraging more explanations, (3) scaling different data sources, (4) reactivity of the programming environment, (5) managing messes with out-of-order cells and throw-away code, (6) improving the delivering of results, (7) facilitating team collaboration.

2.3.1 Lowering the Barriers to Programming

Given that quick iteration is an important attribute in exploratory programming, many computational tools have made design decisions to lower the barriers to programming. Beginning with the choice of languages, Jupyter Notebook supports mainstream and established programming languages (e.g., Python and R) for data analysis while Mathematica and Matlab choose tailored programming languages for novice and end-user programmers with math and engineering background.

Novice programmers can benefit from writing code in natural language descriptions. For example, Gulwani et al. implemented the NLyze system as a natural language-based interface for spreadsheet data analysis [70]. The core algorithm leverages ideas like keyword programming and semantic parsing to achieve an accurate and robust result. Wolfram released the Wolfram natural language understanding system that converts natural language descriptions into the Wolfram Language.

Recently, many research prototypes have emerged to explore a close transition between code and graphical interface. Matlab allows users to explore how different algorithms work with the data interactively, and generate a program to reproduce or automate the work. Many editors include contextual hints for function arguments. For example, providing a dropdown list of candidate argument values for an API call, using colors to highlight different types of data structures. Tools like the nteract data explorer [123] and Pandas Profiling [128] help users automatically generate interactive overview of the

dataset. Domain specific computational notebook tools like the GenePatternNotebook [6] allow users to add a new type of cell to run analysis through interactive web forms without having to write code. Drosos et al. proposed the Wrex extension which allows users to demonstrate data transformation tasks directly on an interactive grid view of the data frame, and generates synthesized code based on the examples [51]. Kery et al. designed a set of GUI tools for users to program in computational notebooks through direct manipulation (e.g., filtering data; editing images) [92]. Wu et al. designed the B2 extension that generates code from operations on interactive visualizations so that users can explore the problem in interactive visualizations while tracking the steps [195]. Automated machine learning products like AutoAI [188], H2O.ai [7], and DataRobot [42] provide a graphical interface for users to simplify the process of analyzing and modeling data, and generate computational notebooks to reproduce the process.

2.3.2 Encouraging Explanations

As mentioned previously, many programmers did not follow the literate programming paradigm while working with computational notebooks. Writing thoughts through the development of code may conflict with some programmers' goal to quickly experiment with ideas. However, not keeping detailed notes may result in the hesitation to later polish notebooks for sharing and presentation. Important decisions and rationales may be missed from the narratives. In chapter 4, I seek to capture valuable insights in discussions by connecting the conversations with computational notebooks [184]. This approach is particularly helpful when the creation of notebooks involves conversations between multiple collaborators. New notebook collaborators can benefit from the close connected discussions to avoid misinterpretations and duplicated work. However, this approach does not work when the notebook has a single author. One potential direction to encourage more explanations is to simplify the process of writing explanations. For example, some tools (e.g., the GenePattern Notebook [6]) use WYSIWYG editors that allow users to manipulate the text through rich formatting, which benefit users who are not familiar with the document markup languages. Tools like Paircast [126] demonstrates another possibility to document the code with audio-based transcribing techniques.

2.3.3 Scaling Data Sources

Another thread of innovations examines the need for scaling different data sources and making it easy to connect the notebook with the data sources. For example, Zhang and Guo [206] presented the DS.js system which enables users to get started with data analysis on tables and data source on any webpage. This approach can particularly encourage data science learners to explore interesting data sources and learn data science techniques. In addition, tools like Datasette [1] and Qri [146] provide direct integration to Jupyter Notebook which gives users access to community shared datasets on GitHub. On the other hand, tools and packages explore ways to create and modify data in notebook tools. For example, the ipysheet [82] extension integrates the spreadsheet feature in Jupyter Notebook where users can input values for a data frame. The ipyannotate [81] extension allows users to annotate data through a graphical interface, which can be useful for many machine learning tasks. Lau et al. summarized the types of data sources into local files, cloud storage, large data, and streaming data [107]. It is worth noting the streaming data, where the notebook is connected to real-time streaming data sources for computation. DeLine et al. [46] studied the design of data science environments for live data through the Tempe prototype. In particular, the design of Tempe is informed by Tanimoto's liveness taxonomy [173] where stored data keeps level-3 liveness (edits would trigger any necessary re-computation) and stream data keeps level-4 liveness (edits would trigger updates to ongoing computations).

2.3.4 Reactivity

Common computational notebooks implement two different programming paradigms. Observable [18] and Pluto.jl [139] implement reactive or dataflow programming where affected cells would be automatically updated when changing a function or variable. These tools also enable users to embed functions and variables inline with Markdown syntax, which allow rendered text to be updated with changes on the notebook or the data. Another advantage of using the reactive programming paradigm is that users do not need to worry about cell orders, which makes the notebook logically consistent. Users can organize the notebook content more flexible by moving interesting visualizations and narratives upfront, and moving complex and technical function definitions into the appendix. To achieve reactive programming, both environments limit multiple definitions across cells so that the cells do not form an infinite dependency loop.

Other computational notebook tools like Jupyter follow the more conventional REPL paradigm — reads the users' command, evaluates the command, and prints the results. For example, Jupyter notebook prints the output only when users execute the corresponding code cell. Executing other cells do not trigger updates or re-evaluation of this code cell. This approach will save costs for large computations (e.g., triggering recomputation on model training manually). This approach also help generate computational narratives where outputs are clearly matched with the history states of the analysis. A major drawback of this approach is that programmers need to mentally manage the execution orders, which harms the reproducibility of the notebook. Without clear instructions, it is not easy for collaborators to execute the notebook in a proper order and replicate the analysis. In next section, we will discuss approaches for managing out-of-order cells.

2.3.5 Managing Out-of-order Cells and Throw-away Code

Although the design of cell structure allows programmers to explore code freely without having to worry about the order of execution, it leads to messy, cluttered, and inconsistent notebooks with out-of-order cells [157]. In particular, many of the code cells contain experimental code that may or may not contribute to the narrative. Head et al. use code gathering techniques to help programmers tracing relevant cells that contribute to a particular cell output [75]. A usability study shows that the tool can help data professionals better organize and clean their notebooks. Kery et al. [88] took a different approach to develop algorithmic and visualization techniques for programmers to better forage past exploration histories. Their evaluation found that this approach can improve the success rate for finding specific information from a past data science project.

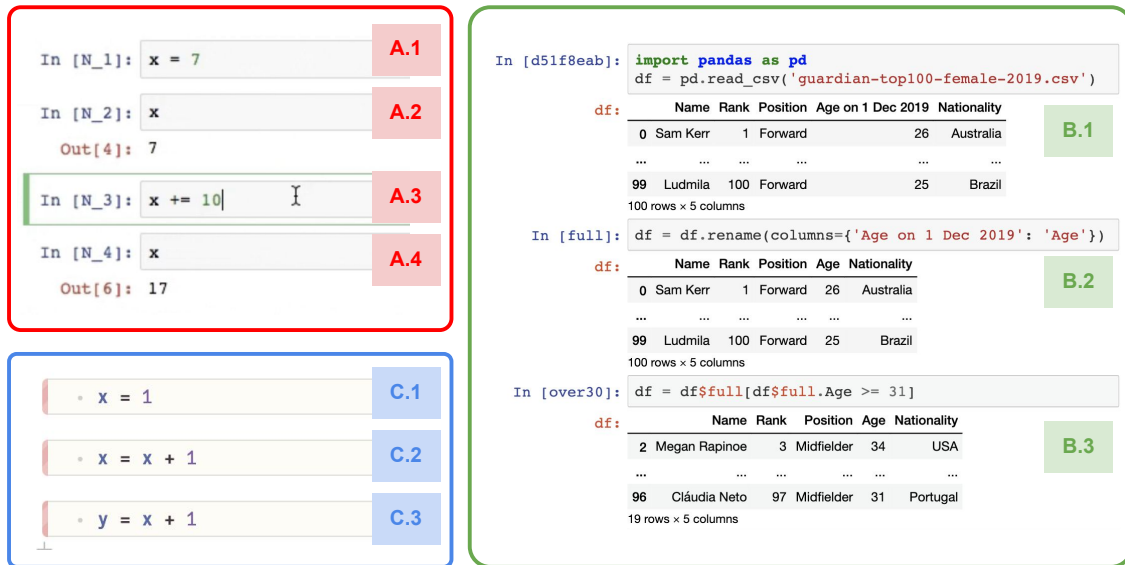


Figure 2.1: Non-traditional methods for evaluating code cells in computational notebooks: (A) The Nodebook extension [208] enforces an ordered flow of cell execution. For example, no matter how many time A.3 is executed, the output of A.4 would always be 17. (B) The Dataflow extension [100] allows users to call a specific version of the variable in other cells' output. For example, B.2 changed the value of `df` by renaming columns. B.3 used the value of `df` from B.2 to run new computations. (C) The Pluto.jl environment [139] implements reactive programming where relevant cells would be automatically updated when changing a function or variable. For example, updating the statement of `x` in C.1 would propagate changes to C.3. Pluto.jl prevents multiple definitions of global variables across cells. Statement C.2 and C.1 would trigger an error, causing statement C.3 not able to return values. This figure is adapted from [208, 100, 139].

Part 1

Understand Challenges in Collaborative Data Science

CHAPTER 3

How Data Scientists Use Computational Notebooks for Real-Time Collaboration

Effective collaboration in data science can leverage domain expertise from each team member and thus improve the quality and efficiency of the work. Computational notebooks give data scientists a convenient interactive solution for sharing and keeping track of the data exploration process through a combination of code, narrative text, visualizations, and other rich media. In this paper, we report how synchronous editing in computational notebooks changes the way data scientists work together compared to working on individual notebooks. We first conducted a formative survey with 195 data scientists to understand their past experience with collaboration in the context of data science. Next, we carried out an observational study of 24 data scientists working in pairs remotely to solve a typical data science predictive modeling problem, working on either notebooks supported by synchronous groupware or individual notebooks in a collaborative setting. The study showed that working on the synchronous notebooks improves collaboration by creating a shared context, encouraging more exploration, and reducing communication costs. However, the current synchronous editing features may lead to unbalanced participation and activity interference without strategic coordination. The synchronous notebooks may also amplify the tension between quick exploration and clear explanations. Building on these findings, we propose several design implications aimed at better supporting collaborative editing in computational notebooks, and thus improving efficiency in teamwork among data scientists.

3.1 Introduction

The complexity of data science work and the demand to adopt data science practices in various domains has grown rapidly in the last decade. With this increase in adoption, there is a need to facilitate collaboration among data science workers, domain experts, and consumers. Data scientists often create *computational narratives*, which combine data, code to process those data, and natural language explanations to form a narrative. Some even consider computational narratives to be the engine of collaborative data science [3]. *Computational notebooks* allow data scientists to create and share computational narratives. *Jupyter Notebook*¹, a computational notebook platform that supports more than 40 programming languages, has been widely used for writing and sharing computational narratives in various contexts [137]. For example, data science instructors use Jupyter notebooks to create interactive lecture notes or textbooks. Data science learners can experiment with these interactive lecture notes to deepen their understanding or explore alternative solutions [103]. Researchers use Jupyter notebooks to demonstrate their computational work and share their data analysis process for open science, which makes it easy for others to reproduce the results [148]. As Figure 3.1 shows, Jupyter notebooks allow users to weave together source code, narrative text, visualization, computational outputs, and other rich media using structured cells.

Prior studies have revealed the challenges in constructing and sharing computational narratives through notebooks. For example, data scientists are reluctant to keep up-to-date explanatory notes, which impedes sharing and collaboration [157]. Studies have also explored ways to lower the barriers for writing and sharing computational narratives: folding content selectively [155]; local version control mechanisms [87]; and managing and reorganizing content [75]. Most of these innovations are designed and evaluated for sharing the computational narrative after it is finished, leaving data scientists to work on individual notebooks. Recently, tools like Google Colab² have demonstrated the possibility for *synchronous editing* — multiple users are able to edit the same notebook and changes are updated in real-time, which may revolutionize the ways data scientists collaborate.

However, synchronous editing comes with its own challenges and may not always improve work efficiency. Studies have identified several issues with synchronous editing in other contexts. For example, in narrative writing, multiple users rarely synchronously edit

¹<https://jupyter.org>

²<https://colab.research.google.com>

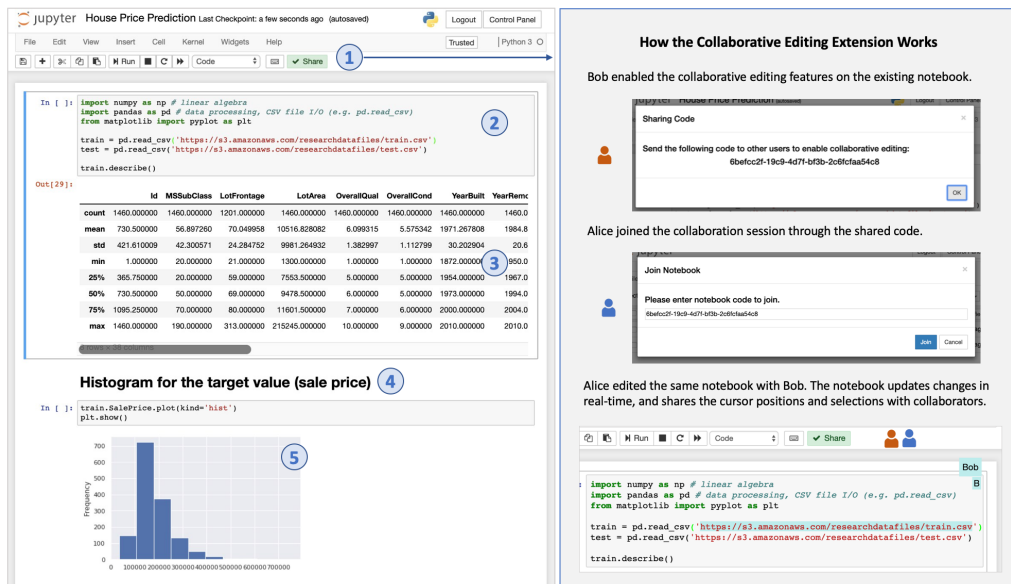


Figure 3.1: An example of a Jupyter notebook. (1) A custom collaborative extension for users to share their notebook. (2) A notebook cell that contains code to import libraries and load dataset. (3) An output of a shared data frame. (4) A markdown cell that contains narrative text. (5) An output of a visualization

adjacent content [41], and programmers can interfere with each others' work when using synchronous code editors [65]. While many of these studies are constructed around the context of collaborative writing and programming, it remains unknown how synchronous editing in computational notebooks might work for data scientists. More specifically, we are interested in three questions:

1. What tools and strategies do data scientists currently use for collaboration?
2. Compared to working on individual notebooks in a collaborative setting, how does synchronous notebook editing change the way data scientists collaborate in computational notebooks?
3. What challenges, if any, do data scientists perceive in synchronous notebook editing?

To address these questions, we first conducted a formative survey with 195 data scientists who were familiar with Jupyter notebooks. Based on the common tools and mechanisms they use for collaboration and communication, we further conducted an observational study with 24 intermediate data scientists working in pairs remotely to solve a

predictive modeling problem. To understand a broader spectrum of synchronous notebook editing, we assigned participants with different settings for collaboration, which included: (1) working on *synchronized notebooks* — participants worked on notebooks with synchronous editing similar to that in Google Docs, and could communicate through other tools, (2) working on *individual notebooks* — participants were not able to edit the same notebook, but could communicate the intermediate code and output through other tools. We also provided participants different options for communication which included: (1) text-based messaging, (2) video chat. Through this observational study we empirically probed both the benefits and challenges for the different ways of collaborating with computational notebooks.

Our key finding reveals that working on synchronized notebooks can improve the collaboration outcomes by reducing communication costs and encouraging more exploration in a shared context. However, working on synced notebooks requires participants to be more strategic when coordinating their work. Working on synced notebooks is more likely to lead to unbalanced participation where one team member does the majority of implementations and ideation. In addition, participants found other challenges in using synchronized notebooks such as interference with each other, lack of awareness, and privacy concerns. These findings suggest ways in which we need to improve the design of collaborative notebook editing tools to better foster teamwork among data scientists.

The main contributions of this work are the empirical insights we present on how data scientists collaborate using computational notebooks. These insights lead to design implications to enhance collaborative computational notebooks for fostering collaboration among data science learners and practitioners. This work extends prior work on real-time collaborative systems and is broadly applicable to understanding collaboration in exploratory and open-ended tasks.

3.2 Overview of the Methodology

This project investigates three research questions:

- RQ1 What tools and strategies do data scientists currently use for collaboration?
- RQ2 Compared to working on individual notebooks in a collaborative setting, how does synchronous notebook editing change the way data scientists collaborate in computational notebooks?

RQ3 What challenges, if any, do data scientists perceive in synchronous notebook editing?

To understand tools and strategies data scientists currently used in practice (RQ1), we first conducted a survey with 195 data scientists/data science students who came from diverse backgrounds. We summarized the tools they used for programming, communication, and project management, as well as their strategies for collaboration. In particular, we identified two approaches for data scientists to collaborate: (1) the traditional collaboration setting where team members work on individual Jupyter Notebook and update each others' work asynchronously, and (2) the emerging collaboration setting where team members work closely together on a shared Jupyter Notebook and all the edits are synchronized in real-time.

To further compare how data scientists' collaboration styles varied between two approaches (RQ2), we conducted an observational study with 24 intermediate data scientists working in pairs remotely to solve a predictive modeling problem. We summarized the common collaboration styles that emerged in the two collaboration settings. We reported the comparison of communication styles, performance, and perceptions of the collaboration experience. We also analyzed the challenges that participants faced in using real-time collaborative editing features (RQ3).

3.3 Study 1: Formative Survey on Collaborative Data Science

We conducted a formative survey to investigate data scientists' previous experiences with collaboration. In particular, we aimed to understand the tools they used for programming, communication, and project management, as well as their strategies for collaboration.

We sent the survey to data science interest groups in a university and to individuals who completed a specialization course on Coursera that teaches data science using Python. Respondents had to meet the following criteria to take the survey: (1) be familiar with Python and Jupyter notebooks, (2) have formal training in data science, and (3) have worked individually on at least one data science project. To motivate participation, we randomly picked two respondents and rewarded them each a \$25 gift card.

3.3.1 The Survey Instrument

The online survey consisted of four sections: (1) informed consent, (2) demographics and data science experience, (3) experience with collaborative data science, and (4) willingness to participate in the observational study.

In the first section, we explained the purpose of the survey and collected respondents' consent. The second section asked about demographics (age, gender, educational background, job role), experience with data science (e.g., what techniques they have used or learned in the past), and degrees of identification as the given roles: computer scientists, software engineers, data scientists, and statistician analysts. In the third section, we asked respondents to recall their most recent experience of working with other people on a data science project. If respondents did not have any experience in collaboration, we provided them a hypothetical scenario:

“Imagine you are participating remotely at a 2-day long data science hackathon with 2 other team members on a predictive modeling problem.”

Respondents were then asked to provide more details about the project, for example, describing the context of the project (e.g., purpose, problem, and dataset), and selecting activities that are involved in the project (e.g., collecting data, sampling data, feature selection, data visualization). In addition, respondents were asked to list the tools they have used when collaborating with others on data science projects for the purposes of *programming*, *communication*, and *project management*. Followed by an open-ended question, we asked their strategies for keeping a shared understanding across the team. In the last section, we asked whether respondents would be willing to participate in an observational study. We collected the names and e-mail addresses of participants that indicated they were willing to participate.

3.3.2 Data Analysis

We first filtered the responses by deleting incomplete entries and merging duplicated entries from the same respondents. For the open-ended questions, we took an inductive analysis approach to explore themes for each question. Four coders worked individually on a small sample of the responses and developed a list of potential codes. After discussing and merging the coding scheme, we coded the same sample independently. We then compared the result by computing a Fleiss' Kappa score to measure both the

reliability of the coding scheme and the agreement among the coders. We iterated on the coding scheme until an appropriate level of agreement was achieved among the four coders (Fleiss’s kappa, $\kappa = 0.74$).

3.4 Key Findings from Study 1

We reported the key findings from the formative survey study.

3.4.1 Data Overview

The survey received 195 valid responses in total (23.08% female and 76.92% male). Among 195 responses, 35 are from data science interest groups in a university and 160 are from individuals who completed a data science specialization course on Coursera. The majority of respondents (73.8%) were age 20–40. Respondents came from a variety of job roles: students (29.74%), data scientists (25.13%), software engineers including Information Technology (IT), system architecture (14.87%), researchers (9.74%), managers including CEOs, VPs, and product managers (9.23%), business analysts (8.20%) and others (3.09%, e.g., drilling engineer).

The respondents were generally well trained in data science. The majority of them (94.36%) held or were pursuing a bachelor or higher degree. Most of these degrees are in technical fields (e.g., computer science, information science, electrical engineering, applied science, or data science). When asked about their previous experience in applying common techniques in data sciences, the respondents indicated they were skilled in linear regression (96.92%), decision trees (92.31%), SVMs (85.64%), neural networks (82.05%), non-linear regression (70.26%), deep learning (65.13%), Bayesian modeling (62.56%), and other advanced techniques (e.g., XGBoost, reinforcement learning).

The respondents also reported engaging in a variety of data science activities: data cleaning (67.69%), applying machine learning algorithms (57.95%), data visualization (52.82%), exploratory data analysis (49.74%), collecting raw data (44.10%), writing a report (38.97%), feature selection (38.46%), applying statistical models (31.28%), doing an oral presentation (31.28%), data sampling (28.20%), hypothesis testing (22.56%), and building data products (21.03%).

Purpose	Tool	Percent
Programming	Jupyter Notebooks	88.72%
	Integrated Development Environments (IDEs) (e.g., RStudio, PyCharm)	51.79%
	Code Editors (e.g., Atom, Sublime)	46.15%
	Google Colab	12.31%
Communication	E-mail	79.49%
	Face-to-face Communication	68.72%
	Instant Messaging	55.90%
	Google Docs	40.51%
	Video Conferencing	38.46%
Project Management	Version Control (e.g., Github, Bitbucket)	49.74%
	Task Tracking (e.g., Trello, Jira)	21.03%
	Shared Storage Services (e.g., Google Drive, Dropbox)	3.07%

Table 3.1: The tools that respondents have used for programming, communication and project management during collaboration.

3.4.2 Experience with Collaborative Data Science

When asked about their previous experience in collaborative data science, most respondents (73.3%) reported that they had prior experience collaborating with other people on a data science project. Of the respondents who had previous collaboration experience, most (72.72%) collaborated on a data science project for work, while the others mentioned other purposes such as course projects (30.07%), competitions or hackathons (18.89%), or personal side projects (13.20%).

3.4.2.1 Choices of Tools

We asked respondents to list the tools they have used for programming, communication and project management during collaboration. For respondents who do not have or can not remember their past collaboration experience (26.7%), we gave them a scenario of participating in a data hackathon in teams and asked about their choices of tools and collaboration strategies. For programming purposes, most respondents mentioned Jupyter notebooks (88.72%), IDEs (51.79%), and code editors (46.15%). In addition, some respondents mentioned Google Colab (12.31%), a computational narrative environment

which is an alternative to, but similar to, Jupyter notebooks. For communication purposes, e-mails (79.49%) and face-to-face communication (68.72%) are most mentioned by respondents, followed by instant messaging (55.90%), Google Docs (40.51%) and video conferencing tools (38.46%). For project management tasks, roughly half of respondents mentioned version control suites (49.74%) such as Github and Bitbucket for sharing code and datasets. Several respondents also mentioned using shared storage services such as Google Drive and Dropbox for managing project assets. Some respondents mentioned task tracking tools (21.03%) such as Trello and Jira. Respondents also mentioned using Google Calendar, spreadsheets, or physical whiteboards to keep track of tasks and project progress. Our results are summarized in Table 3.1.

Strategy	Percent	Example Response
Discussions and meetings	54.36%	<i>There were weekly meeting among team members to keep track of the progress of each element of the project</i>
Frequently check-ups	51.79%	<i>Communicate actively and frequently; check-up on every hour</i>
Documentation	48.20%	<i>Keep notes in Google Docs; ... comments in code;</i>
Organization	28.72%	<i>Divide up the work into definable parts, make sure that everyone knows the progress that everyone else has made and how it impacts their part of the project</i>
Shared Assets	25.13%	<i>Common repository for files; share the same place for storing project, and name the file clearly</i>
Others	5.01%	<i>Code review to ensure code matched intent; all worked in a friendly manner</i>

Table 3.2: Strategies for keeping a shared understanding

3.4.2.2 Strategies for Keeping a Shared Understanding

When asked about their strategies for keeping a shared understanding across team members, most respondents mentioned regular discussions and project meetings (54.36%). For example, respondents mentioned “weekly meetings with the team to follow up the stages of deployment”, and also frequently reported that they used check-ups (51.79%) such as

keeping the other team members informed of any changes made to the code. Some of these respondents reported that they would work closely in a physical space to reduce the communication cost by talking face-to-face. Documenting (48.20%) is another common strategy used for collaboration. For example, respondents mentioned that they would keep all the intermediate findings in shared Google Docs. Some respondents also mentioned coordination strategies such as planning ahead and being clear about everyone's responsibility (28.72%). In addition, respondents mentioned that they would share any intermediate results and code using version control tools or shared folders (25.13%). The others mentioned strategies such as code reviews to help them maintain a shared understanding. The results are summarized in Table 3.2.

3.4.3 High-Level Summary of Findings

In summary, respondents to our survey were made up of a variety of practitioners and students who are well trained in data science and are familiar with Python and Jupyter notebooks. Most respondents had previous experience in collaborating with others on a data science project, and working in individual Jupyter notebooks with version control tools was the most popular setting for collaboration. Team members constantly discuss and keep everyone informed about the progress of the project, as well as maintain shared notes. Although Google Colab was relatively new and was not be used by many respondents, several respondents mentioned Google Colab as an option for collaborative editing. With this more holistic understanding of collaboration among data scientists, we decided to narrow our focus and probe into the differences afforded by traditional collaboration settings, where team members work on individual Jupyter notebooks and update each others' work asynchronously, and the emerging real-time synchronized editing collaboration setting, where team members work closely together on a single Jupyter notebook with shared edits.

3.5 Study 2: Observational Study on Real-time Collaborative Data Science

We conducted an observational study with 24 data scientists working remotely in pairs to solve a predictive modeling problem. We tested two conditions, with users working on either individual notebooks or notebooks that enable synchronous editing in a collabora-

PID	GID	Country	Occupation	Major
P01 (M)	S1	U.S.	Student	Master in Information Science
P02 (F)	S1	U.S.	Student	Bachelor in Statistics
P03 (M)	N1	U.S.	Data Scientist	Master in Data Science
P04 (M)	N1	U.S.	Student	Master in Computer Science
P05 (F)	N2	Canada	Business Analyst	Master in Management
P06 (M)	N2	Pakistan	Software Engineer	Bachelor in Computer Science
P07 (M)	N3	India	Student	Bachelor in Information Technology
P08 (M)	N3	India	Student	Bachelor in Computer Science
P09 (M)	N4	U.S.	Student	Bachelor in Computer Science
P10 (M)	N4	Brazil	Data Scientist	Bachelor in Computer Science
P11 (M)	S2	Canada	Student	Master in Computer Science
P12 (M)	S2	Canada	Student	Bachelor in Computer Science
P13 (M)	S3	Canada	Student	Bachelor in Computer Science
P14 (M)	S3	Canada	Student	Bachelor in Computer Science
P15 (M)	S4	U.S.	Student	Bachelor in Economics
P16 (M)	S4	U.S.	Student	Master in Information Science
P17 (M)	S5	China	Software Engineer	Bachelor in Computer Science
P18 (F)	S5	China	Student	Ph.D. in Computer Science
P19 (M)	S6	U.S.	Data Scientist	Master in Computer Science
P20 (F)	S6	U.S.	Business Analyst	Master in Business
P21 (M)	N5	India	Student	Bachelor in Computer Science
P22 (M)	N5	India	Student	Master in Computer Science
P23 (M)	N6	India	Student	Master in Computer Science
P24 (M)	N6	India	Student	Master in Business

Table 3.3: Demographics of Participants in Study 2

tive setting. Our goal was to examine how collaboration styles varied between the two conditions, and to gain empirical insights on the benefits and trade-offs for each setting.

Groups chose from two communication mechanisms that were commonly used for collaboration: Slack for text-based messaging and Google Hangouts for video-based communication. In pilot studies, we found it difficult to control the communication mechanism due to technical limitations (e.g. participant microphone or network issues). Thus at the beginning of the study, we asked individual groups to decide which communication mechanisms they wanted to use throughout the study.

3.5.1 Participants and Task

We recruited participants who had a sufficiently substantial background in data science: (1) be familiar with Python and Jupyter notebooks, (2) have formal training in data science, and (3) have experience with predictive modeling. For reference, we provided an example predictive modeling task to potential participants for evaluating whether to opt-in to the study. We reached out to 12 initial participants from respondents to our first survey who indicated a willingness to participate in future research, and used them as seeds to recruit the rest of participants through snowball sampling. Participants (4 females, 20 males, average age = 24.62) all had basic knowledge in data science related fields (e.g., computer science, business analytics, statistics, economics), as listed in Table 3.3. Seven participants currently worked as data scientists and analysts, and most (23/24) have collaborated with others on a data science project before.

Participants were randomly assigned to pairs and were instructed to work collaboratively on a predictive modeling problem. The task given was to predict housing sale prices using 80 features (e.g., lot size, type of road access, original construction date). The task involved features not relevant to the prediction, outlier records, and missing values. Pairs were asked to develop their own strategies to judge the importance of features, handle pre-processing, and apply predictive models for sales prices. This task was based on a beginner-level competition on Kaggle³, and indirect questions were asked in the recruitment process to ensure that participants had not worked on the same task previously. We chose the predictive modeling problem since it captures the majority activities in the data science pipeline (e.g., exploratory data analysis, data cleaning, modeling, and evaluation). In addition, the problem is open-ended, which left space for groups to try advanced models and improve the prediction result.

3.5.2 Apparatus

Participants joined the study remotely and their browser screens, webcams, and microphones were recorded by the research team with consent. Groups could choose to use Slack for sending text, images, or code snippets, or Google Hangouts for video calling. A JupyterHub⁴ instance was made available so that participants could access the computational environment and run their notebooks from the cloud. Groups were assigned to

³<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>

⁴<https://jupyter.org/hub>

work in either the shared condition or non-shared condition as described below.

3.5.2.1 Non-Shared Condition

In the non-shared condition, participants logged on to the JupyterHub platform and worked on individual notebooks. They were allowed to exchange the notebook file, set up a git repository, or send code snippets through chat or any other tool.

3.5.2.2 Shared Condition

In the shared condition, a Jupyter notebook collaboration extension was enabled on the server to support synchronous editing in notebooks. Participants logged on to the Jupyter-Hub platform, created a shared notebook and invited their teammates to join the notebook.

3.5.3 Collaboration Extension

The collaboration extension allows code to be executed on a single interpreter while the output and runtime variables are shared among collaborators (Figure 3.1).

3.5.3.1 Workflow

To demonstrate the workflow of the extension, consider a data scientist Bob is working with his colleague Alice remotely on an exploratory data analysis problem. The extension allows Bob to enable the collaborative editing mode on the existing notebook, and share the session code with Alice. Alice then joins the notebook using the session code and edits the same notebook with Bob. The notebook updates changes in real-time, and shares the cursor positions and selections between Alice and Bob.

3.5.3.2 Implementation

We use Operational Transformations (OTs) to handle real-time information exchange among notebooks in the shared session, where OT is a widely-used technology for supporting consistency maintenance and concurrency control in real-time groupware systems (e.g., Google Docs). There are two types of synchronization strategies in our extension. For operations that do not involve code interpreter (e.g., editing code, adding cells, deleting cells), we use a decentralized model to update all notebooks in the shared session with recent edits. For operations that involve with code interpreter (e.g., execution a code cell,

rendering a markdown cell), we first send the operation to a host notebook. We then execute the cell on the host's interpreter and update clients with output and runtime variables after the interpreter finishes execution. The extension is implemented by a Node.js web server that connects to a Postgres database, and a web-based client that implemented as a Jupyter Extension.

3.5.4 Study Procedure

The observational study consisted of four sessions, each of which lasted an hour. We gave participants goals for every session: to better understand data (session 1), to clean the data (session 2), to create a basic predictive model for the data (session 3), and to create a more advanced prediction model to further improve the results (session 4). Although we encouraged participants to meet the goals we set, we did not enforce these goals or any particular task ordering. Before the first session, groups were given 15 minute orientation to become familiar with each other and the study setting. We provided participants with written instructions about the study task. Groups were allowed to use external resources throughout the study.

At the end of each session, participants were asked to fill out a post-session questionnaire wherein they were asked to describe the exploration progress, their own contributions, and any difficulties they encountered in that session. Participants were also asked to evaluate a set of statements on a seven-point Likert scale. There are three themes in the statements: *communication* (e.g., being able to follow the conversation), *awareness of others* (e.g., being aware of what issues teammates are struggling with), and *group maintenance* (e.g., enjoying working with others). Lastly, the first two groups who completed the study were asked about their feedback on study settings and instructions.

After the final session, groups submitted a final prediction result, together with a merged notebook report that explained the exploration process. To motivate participants, we rewarded each participant a \$40 base incentive. The group with the best prediction result was awarded an additional \$10 for each member.

In addition, we conducted one-on-one post-task interviews to investigate how participants perceived the study. We first asked them to reflect on their collaboration experience (e.g., what was their strategy for coordinating work). Next we used the critical incident technique [58] to investigate if there was a situation where they could not follow the conversation, they did not feel aware of others' work, or they disturbed others' work. Participants were then prompted to think about how these incidents could be addressed if

they could change the features of the tool.

3.5.5 Data Analysis

We used an inductive analysis approach with other methods (e.g., affinity diagramming, memoing) from grounded theory [37] to analyze the data. Screen recordings, open-ended questions from the post-session survey, and interview transcripts were first observed by the lead author to identify: (1) common collaboration patterns, (2) activities that are involved in the exploration, and (3) any challenges that were faced during collaboration. Using an open-coding approach, we first created a coding scheme based on initial observations. Four coders independently coded two samples to refine the coding scheme. We then discussed and used affinity diagramming to synthesize emerging themes. Next, we went through several iterations to check another two samples individually. The purpose of this step was to confirm the legitimacy of the coding scheme and to check the inter-rater reliability. After several iterations, four coders reached a suitable level of agreement (Fleiss’s kappa, κ 0.71).

Group ID	Collaboration Style	Notebook	Message	Error Score
N1	Competitive Authoring	88%	61%	0.21
N2	Divide and Conquer (datasets)	71%	54%	0.36
N3	Competitive Authoring	84%	61%	0.25
N4	Competitive Authoring	69%	55%	0.48
N5	Divide and Conquer (datasets)	68%	59%	0.23
N6	Competitive Authoring	91%	51%	0.12
S1	Divide and Conquer (tasks)	58%	61%	0.13
S2	Single Authoring	94%	N/A	0.15
S3	Divide and Conquer (tasks)	71%	58%	0.21
S4	Divide and Conquer (tasks)	67%	N/A	0.16
S5	Single Authoring	83%	75%	0.23
S6	Pair Authoring	86%	N/A	0.16

Table 3.4: Collaboration Styles. The notebook ratio is the percentage of cells that one member contributes in the final notebook. The message ratio is the percentage of messages that one member sends in total Slack messages; N/A means the team uses video-based communication (Google Hangouts). The error score is calculated using Root Mean Square Error (RMSE). Lower error scores are better.

3.6 Key Findings from Study 2

We first present the results of the common collaboration styles that emerged in the two collaboration settings. Based on the framework of collaborative writing developed by Posner and Baecker [142], we propose four collaboration styles. Then, we present an analysis of our comparison of communication styles, finding that working in the synchronized notebook can reduce the communication costs by establishing a shared context. We also report the differences in performance (e.g., final prediction scores, number of alternative models, notebook lengths) between the two collaboration settings. Lastly, we present the challenges that participants faced in using real-time collaborative editing features.

3.6.1 Collaboration and Communication Styles

Based on what we observed in our study and extending the framework developed by Posner and Baecker [142] for collaborative writing styles, we refine and propose four collaboration styles for data science tasks based on team members' contributions in ideation and implementation.

- **Single Authoring:** The single authoring style is extended from the *single writer* strategy from Posner's framework. In this style, one team member contributed the majority of ideas and did the majority of implementation, while the other did not provide substantial contributions.
- **Pair Authoring:** The pair authoring style is extended from the *scribe* strategy from Posner's framework where one team member completed the majority of the implementation while the other contributed ideas, engaged in discussions and reviewed the results. We chose the term pair authoring because it is as analogous to the collaboration style in pair programming where one person writes the code and the other reviews the code. This is different from pair programming where two programmers frequently switch roles: here, the individuals engaged in pair authoring stick to their roles from the beginning to the end.
- **Divide and Conquer:** The divide and conquer style is a combination of the *separate writers* strategy and *joint writing* strategy from Posner's framework. Here, participants divided the task into subgoals and explored the subgoals independently. We observed two types of divide and conquer strategies — **(1) dividing datasets**, where participants split the dataset into half and explore in parallel using the same

technique, (2) **dividing tasks**, where participants split the task and work on different parts in parallel.

- **Competitive Authoring:** We proposed this collaboration style based on our observation of team members going through an idea together and competitively implementing it. It is different from divide and conquer that team members wrote the code for the same purpose and reached the consensus to use the code by whoever finished first. There is no equivalent strategy in Posner's framework.

We then used these definitions to code each group's collaboration style. We analyzed the notebook contribution ratio (the percentage of cells that one member contributes in the final notebook) and the message ratio (the percentage of messages that one member sends out of the total number of Slack messages) with the code and notes from screen recordings to inform our judgment. We report how groups adopted different collaboration styles using a collaborative editing notebook (shared condition) and using individual notebooks (non-shared condition) in Table 3.4.

3.6.1.1 Single Authoring Style

Two groups adopted the single authoring style when working in the synchronized (shared condition) notebook. We observed an unbalanced contribution in their final notebooks. Participants who contributed less reported that without strategic coordination and communication, they did not have a task that fit their level of expertise and ended up walking through the same task with their teammate. One explanation that arose is that the participants might feel pressured and off-topic, not knowing how to engage in the task:

... My teammate is better and faster in doing the task. Sometimes I know he is trying an idea, but it may take me a while to figure it out and he just jumped to the next task... I wish I could have a separate window to try the code in my own pace... (P12 from S2)

3.6.1.2 Pair Authoring Style

One group used the pair authoring style when working in the same notebook. Two members agreed that one person was in charge of implementation while the other helped with ideation and finding documentation. As opposed to single authoring, where one group

member contributes the majority of code and ideas, in pair authoring, both group members felt engaged in decision making. They further explained why they found pair authoring useful using the pair programming metaphor of a *driver* who writes the code, and a *navigator* who reviews the code:

... There is a dynamic in pair programming called “Driving and Navigating”. The idea is that two people on one keyboard is difficult to do well, so one should primarily be leading... (P19, the main driver from S6)

... Data science is not just about writing code, you know. My role is as important as my teammate. I can think about the big picture while my teammate is working on the details in Python. It is not necessary for me to step into his code... (P20, the main navigator from S6)

3.6.1.3 Divide and Conquer

We observed the pattern of “divide and conquer” from five groups, with three groups from the shared condition and two from the non-shared condition. Participants used different strategies to decide how to divide the task. Some groups planned ahead to discuss the goal of the session and used that to guide how to divide the task. These groups often decided whom to assign the sub-tasks to based on group members’ skills. For example, participants in group S3 used markdown cells to list things they wanted to explore and put group members’ name aside to track the tasks. Other groups did not plan ahead and used ad-hoc planning, keeping each other updated about what they were doing so that the other group member could find a new task to work on. For example, P2 in group S1 said to P1, “*OK, while you fix the stuff, I’ll create one hot encoding for categorical variables*”. However, participants explained their concerns that not knowing enough about another person’s skill set made it difficult to split the work:

... I wish I had a more personal relationship with my fellow research participant so I didn’t feel weird about judging who should do what based on skill... (P13 from S3)

... Basically, I don’t like having to make a judgment about the ability of my partner without having the benefit of knowing our strengths and desires relative to each other. It makes me self-conscious that I’m not listening to my partner enough... (P16 from S4)

In addition, we identified two types of divide and conquer strategies: dividing *datasets* and dividing *tasks* based on alternative solutions. Both groups from the non-shared condition split the dataset and had each member walked through half of the features. This strategy can boost efficiency in exploratory data analysis given that there are roughly 80 features in the dataset. It can help two members to establish common ground in the general task. However, this might result in duplicated implementation. For example, both group members wrote their own code to plot the distribution of certain features. None of the groups in the shared condition used this strategy. Instead, they divided the task by alternative solutions. For example, in the data modeling stages, participants came up with different models (e.g., decision trees, elastic nets) and assigned each person the task to explore one model.

3.6.1.4 Competitive Authoring

Non-shared groups often used competitive authoring, where group members would competitively implement the same idea. They would either choose to copy the code from whomever finishes the implementation first or choose to keep their own version of implementation. For example, two group members were both working on the implementation of linear regression, while one member got their code working faster than the other member. The other member would agree to copy the code to their own notebook and move on to the next task. There can be an unbalanced distribution in the final work and working efficiency can decrease because one member is always writing “unnecessary” code. However, from the individuals’ perspective, they sometimes felt it necessary to explore the code on their own:

... Each would do the same tasks and share the insights ... I don’t think it is wasting time. When we both wrote the code, at least we were on the same page. When he shared his code, I can have a better understanding of what was going on... (P24 from N6)

3.6.2 Communication Channels

Web conferencing is perceived to have high communication bandwidth with synchronous and immediate information exchange, whereas instant messaging and chat tools have a lower communication bandwidth and support both synchronous and asynchronous information exchange [175]. The fact that participants chose to use Slack rather than Hangouts

in the non-shared condition (without prompting from the study coordinators) indicates that the affordances of Slack with a variety of media formats, as well as the ability to represent conversation artifacts (e.g. source code) in a manner which fit their needs. Data scientists need to constantly share intermediate code or outcomes with each other. Working in the shared notebook reduces the communication costs by providing a shared context for discussion. For example, a participant from the non-shared condition explained that they preferred Slack messaging over Hangouts, and they wished to have the shared notebook for better communication:

... would be much nicer to work on the same notebook rather than copying code in Slack but worked much better than Hangouts... (P7 from N3)

In addition, we examined the types of messages participants sent in each condition. Overall, we found that participants in the non-shared condition used Slack more often to send files, code snippets, and output (e.g., data pieces, error messages, visualization, screenshots of the notebook). Every group in the non-shared condition exchanged their notebooks at least once during the study session. Groups in the non-shared condition also sent more execution output such as data pieces and images in the Slack channel. Moreover, they sent at least four times as many code snippets during the study, while groups in the shared condition rarely did so. This result suggests that working in the same notebook can reduce the communication costs by establishing a shared context.

3.6.3 Working Across Phases

As we describe in section 3.5.4, we gave participants recommended goals for each of the four sessions but did not require that they met these goals or worked in any particular order. In order to understand participants' exploration patterns and how they differ across both conditions, we segmented the screen recordings into 15-minute intervals and categorized participants' activities. We found that participants typically did any given activity for a minimum of 10–15 minutes so 15 minutes was an appropriate level of granularity. For each 15-minute interval, we categorized their activity into one of the common data science phases described by O'Neil and Schutt [124]⁵:

- **Preparing:** Reading task instructions and setting up the environment

⁵We omitted the “data munging” and “data collection” phases because participants were given an existing dataset and did not need to collect raw data themselves.

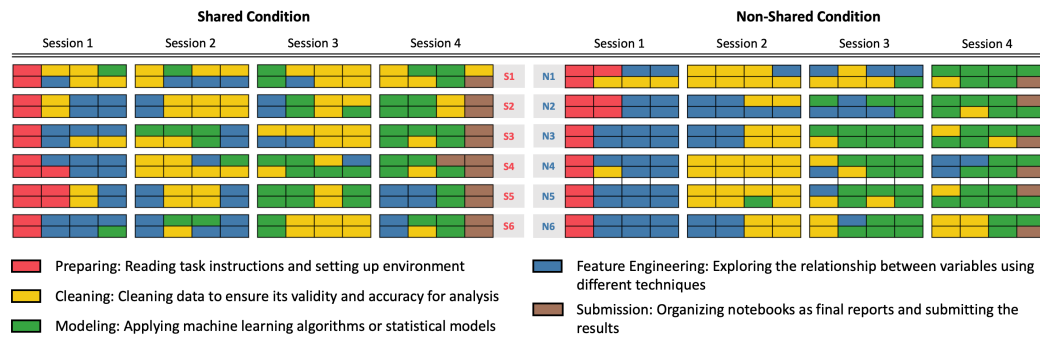


Figure 3.2: Overview of how participants iteratively explore the house price prediction task in 15-minute intervals. Participants in the shared condition tended to switch between phases more frequently. The initial attempts at modeling occurred earlier in the shared condition.

- **Cleaning:** Cleaning data to ensure its validity and accuracy for analysis
- **Modeling:** Applying machine learning algorithms or statistical models for prediction
- **Feature Engineering:** Exploring the relationship between variables using different techniques
- **Submission:** Organizing notebooks as final reports and submitting the results

Figure 3.2 shows the activity classifications for every group across every session. Participants did not perform tasks in sequence — they frequently backtracked and switched between different phases of analysis as necessary. Participants in the shared condition tended to switch between phases more frequently. We calculated how many times participants switched between phases (when the adjacent tiles have different colors in Figure 3.2) and ran a two-sample T-test on the result. We found that participants in the shared condition switched more frequently (avg=8.58, $p=0.000043$) than participants in the non-shared condition (avg=5.83). This result indicated that working on the same notebook provides collaborators with convenience to branch through tasks. Perhaps as a result of this convenience, the initial attempts at modeling (green squares in Figure 3.2) occurred earlier in the shared condition than in the non-shared condition. As Figure 3.2 also shows, in every group in the non-shared condition, one participant bore the responsibility of organizing the notebooks for submission (brown squares in Figure 3.2). By contrast, nearly

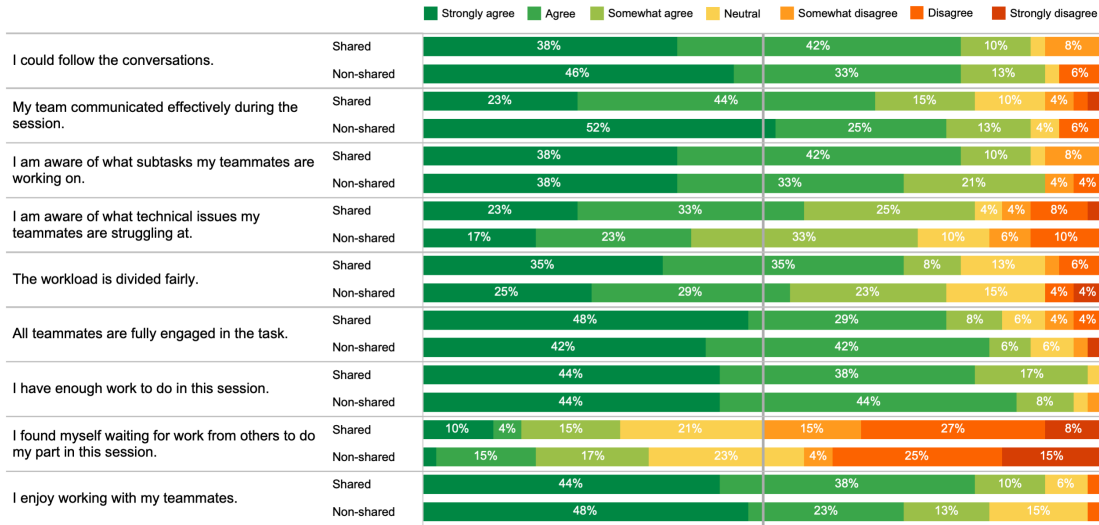


Figure 3.3: Post-task questionnaire results for participants in both conditions.

every participant in the shared condition helped organize the notebook and results for the final report.

3.6.4 Overall Effectiveness — Accuracy and Questionnaire Results

We found an improvement in the exploration process across two conditions. As shown in Table 4.5, we run a two sample T-test to compare the outcomes from the final notebooks and prediction results across two conditions. Groups working in the same notebook (avg error score⁶ = 0.17) achieved a better prediction result compared to groups working in individual notebooks (avg error score=0.27, p=0.09). Groups working in the same notebook explored more alternative models (avg=6.17) compared to groups working in individual notebooks (avg=3.00, p=0.05). In addition, we compared the post-session survey results across all sessions, as illustrated in Figure 9.3. Participants’ rating of their enjoyment working with teammates was significantly improved when working in the shared condition after the first session (p=0.04). We also compared the total lines of code in the final notebook and found a significant difference in the shared condition (avg=186.67) and the non-shared condition (avg=90.33, p=0.04). This result suggests that working in the same notebook encourages groups to explore more solutions and leads to a better result. For example, P2 noted:

⁶The error score is calculated using Root Mean Square Error (RMSE)

... Overall, I think the tool is amazing! This tech can really increase productivity in data science teams!... (P2 from S1)

		Shared	Non-shared
Error Score	\bar{x}	0.17	0.27
	σ	0.04	0.13
Number of Models	\bar{x}	6.17	3.00
	σ	2.99	2.10
Lines in the Notebook	\bar{x}	186.67	90.33
	σ	82.10	64.50
Percentage of Annotation Cells	\bar{x}	0.19	0.20
	σ	0.11	0.13

Table 3.5: Comparing the outcomes from prediction results and final notebooks (mean: \bar{x} , standard deviation: σ). Working in the same notebook encourages groups to explore more solutions and leads to a better result.

When we compared the ratio of annotation cells and total cells from the final submission, we did not see differences across two conditions. Moreover, the average ratio of annotation cells to total cells was lower in the shared condition (avg=0.19) compared to the non-shared condition (avg=0.20). This result indicates that when working in groups, participants would not pay extra attention to add annotations into the notebook compared to working in a private notebook.

3.6.5 Challenges in Using the Collaborative Notebooks

Despite all benefits that collaborative editing features offer with respect to sharing context and improving productivity, we discovered several challenges in using the collaborative notebooks. We present the key findings below.

3.6.5.1 Interference with Each Other

Over half of the participants in the shared condition (7/12) raised concerns about interference with each other in the post-task interviews. Participants would take ownership of the code cells they created. They would expect others not to edit “their” cells. Some participants even took actions such as inserting blank cells between each others’ editing

Pre-processing and cleaning

Steps

1. Replace discrete values with indices
2. Remove data samples with too many missing features
3. Normalize continuous variables
4. Compute correlation, or use other techniques to select features

```
In [24]: 1 def count_nans(data):  
2         for name in data:  
3             count_nan = len(data[name]) - data[name].count()  
4             print(name, 'num of nans:', count_nan)
```

```
In [74]: 1 def label_encoding(data):  
2         for name in data:  
3             if data[name].dtype == 'object':  
4                 data[name] = data[name].astype('category')  
5                 data[name + '_cat'] = data[name].cat.codes  
6
```

Figure 3.4: Group S3 coordinated the work well by planning subgoals in the notebook

areas to prevent interference. Participants also reported that different naming styles could cause trouble, especially changing a variable name halfway through the exploration:

... When using Jupyter notebooks together, it's hard to keep track of variable names. Everyone might use a different name and may cause issues. For example, my teammate used *train_df* as name, and later changed it to something else, but I wanted him to keep using the original name... (P2 from S1)

In addition, we observed that during exploration, some participants directly modified the shared data frame (Figure 3.1.3) without making copies or notifying their teammates. For example, P14 from S3 spent a long time debugging the error score for the basic linear regression model and finally realized that his teammate had transformed the scale of the shared data frame for other purposes.

3.6.5.2 Lack of Awareness

Although the collaborative Jupyter notebook shares cursor positions and selections with collaborators, participants reported that this mechanism was not enough to understand

what their teammates' activities. First, participants would have to scroll the notebook frequently to check their teammates' edits, which can be difficult when the notebook grows rapidly during quick exploration. Second, if the participant only wants to know the high-level task his teammate is working on, it takes time for him to read and understand the code when it is not well annotated. Lastly, the cursor information may not reflect their teammates' activities, especially when doing data science needs reasoning and decision making. For example, participants reported:

... I want it to be easier for my research partner to show me what they're working on. I felt it was difficult to do something quickly on the side without affecting what my partner was working on... (P17 from S5)

... It's hard to keep track of what my teammate is doing while he's not writing code on notebook because I don't see him physically... (P12 from S2)

3.6.5.3 Problems with the Linear Structure

Cells are displayed linearly in Jupyter notebooks but the actual execution order may not follow a linear structure. As mentioned, participants may divide notebook cells into regions based on a sense of ownership. Participants may also iterate and jump through the code cells for ad-hoc divide and conquer. Our analysis of the final notebooks indicates that although the amount of code grew significantly faster in shared notebooks than in individual notebooks, the percentage of annotation cells in the shared notebooks was smaller than in individual notebooks. Such ill-organized cells made it difficult for collaborators to navigate the notebook. In fact, we were not able to run the cells sequentially to reproduce the results in two of the six notebooks submitted by groups in the shared condition. One participant suggested separating the preliminary exploration with the main notebook:

... I want to distribute and segment work more easily, but notebooks fundamentally struggle with this due to the restrictive UI. The cells are stacked top to bottom. There is no concept of a "scratch cell" or "main script". This makes it difficult to say, "here is the main section of code, and we're writing exploratory code on the side to prototype improvements". I appreciate that kind of paradigm ... (P19 from S6)

Participants also referred to different ways to support non-linear structures. For example, one participant proposed to track the execution order in a file which is similar to the

example of “Makefiles” in C programming. Another participant suggested to break down the structure of notebook cells into parallel branches. He offered an example:

... I’d love to see cells as nodes and edges to their dependencies. Imagine LR (logistic regression) and RF (random forest) models. LR needs data to be scaled and preprocessed. RF does not need it. You could set up cells with explicit dependencies to the appropriate preprocessing steps. You can imagine this as data flowing through containers of code... (P15 from S4)

3.6.5.4 Privacy Concerns

Two groups in the shared condition wrote using the single authoring style—one participant in both groups contributed the majority of ideas and implementation. Both participants who shied away from the task mentioned that they felt pressured to edit the code cells despite knowing that their teammates were better than them. These participants also mentioned that they only wanted to share the code when it was done. Such concern was also reported by an experienced participant:

... At the beginning I was hesitant to edit the notebook. It was much better later on because I knew he was busy with something else so that he wouldn’t pay attention to my code... (P2 from S1)

These concerns are analagous to the finding of Wang et al. that some writers are concerned with being judged or distracted in the context of collaborative writing [189].

3.6.5.5 Lack of Strategic Coordination

Lack of strategic coordination also resulted in unbalanced contributions. After the last session, participants reported a lower agreement on “I have enough work to do in this session” in the shared condition than in the non-shared condition ($p=0.08$, Figure 9.3). The participant who contributed more in the single authoring group acknowledged that the work was not divided equally because they did not plan ahead:

... I feel I am not splitting work well enough. I was thinking about how to get the work done and just tried the ideas on myself... (if doing it again) I will probably ask him to help with data cleaning... (P11 from S2)

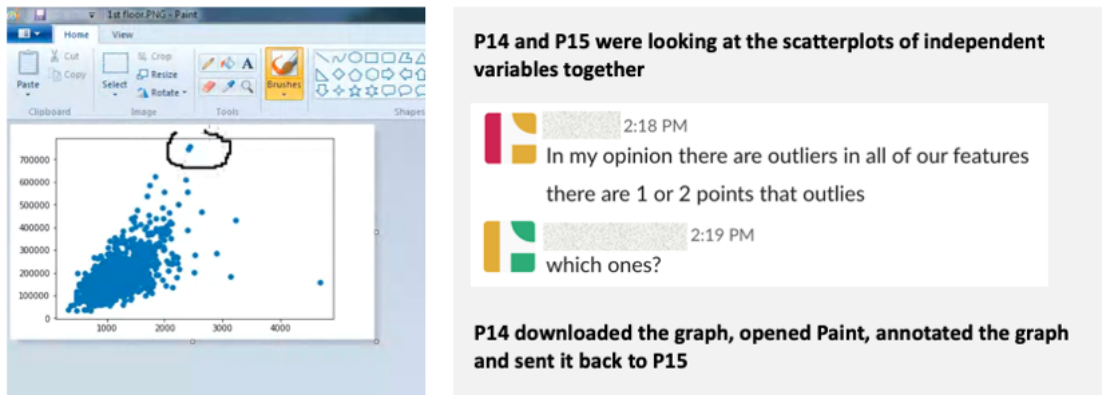


Figure 3.5: An example of participants manually annotating a graph for discussion: P14 circled the outlier point in a scatter plot using MS Paint and sent it back to P15 using Slack.

Another group (S3) demonstrated successful coordination. They listed subgoals in the notebook and assigned tasks based on preferences (Figure 3.4). This planning and negotiation also helped them to divide the code cell regions with nested headings. While some groups also discussed the subgoals verbally or in Slack, we did not see a clear notebook structure. Retrieving such information can be useful to automatically help collaborators to manage their shared notebook. We will elaborate on this in the discussion.

3.6.5.6 Contextual Chatting

As we describe in section 3.6.2, participants in the shared notebook condition sent fewer code snippets, images, execution results, and screenshots on average — likely because all of these elements were automatically synchronized across all team members, which eliminates the need for users to share them manually. As a result, we expected participants in the shared condition to believe their teams communicated more effectively than participants in the non-shared condition. However, we found the opposite—participants in the shared condition reported that they felt their groups communicated worse than participants in the non-shared condition (as measured by participants’ agreement with “my team communicated effectively during the session” after session 2, $p=0.08$, Figure 9.3). One possible reason for this is that participants in the shared condition switched between phases more frequently (as Figure 3.2 illustrates), which could make communication more challenging.

In the post-task interview, we specifically asked about the difficulties participants had

with communication. Most participants (9/12) mentioned that constantly switching between Slack and Jupyter notebook was distracting. Several participants did not enable notifications of Slack messages and were not able to respond to their teammates immediately. Participants also felt the need to refer to some output (e.g., a specific column of a table, an area in visualization) during discussion but current tools do not support deictic references. Figure 3.5 shows an example of participants manually annotating a visualization for discussion: P15 was not clear where the outlier points were in the scatter plot so P14 had to download the graph from notebook, opened Microsoft Paint to circle the point and sent it back to P15 through Slack.

3.7 Discussion

Our main findings indicate that synchronous editing helps data scientists maintain a shared understanding while reducing communication costs, thus improving the overall efficiency of collaboration. However, current synchronous editing features can be challenging to use and require collaborators to be strategic with respect to coordination. Below, we contextualize our findings with existing frameworks and findings on collaborative editing in other contexts, and highlight the needs for a human-centered approach to study different collaboration scenarios in data science. We discuss future directions to improve the current design of synchronous notebook editing features to better support teamwork among data scientists.

3.7.1 Extending Our Understanding of Collaborative Editing Across Contexts

Some of the challenges we identified with real-time notebook editing are related to prior studies on collaborative writing systems and collaborative coding systems. For example, the privacy concern of being watched by others while working has been observed in other contexts [189, 41]. This issue may be more pronounced for collaborative data science, because data science requires a large amount of experimentation with code and collaborators may hold different programming backgrounds and domain knowledge, something noted by others (e.g. [93]) and observed by ourselves in this work. In addition, there is an opportunity for future human-centered studies to explore different roles in collaborative data science. Additional surveys and ethnographic work can aid the understanding

of the whole spectrum of human-centered data science work. It is also worth exploring novel designs in this space. For example, folding code blocks [155] and implementation details might allow domain experts (e.g., marketing specialists) to be able to understand and experiment with model parameters in a notebook. Another challenge for data scientists working on the same notebook is interference of work, which is also a challenge for collaborative code editors [64]. However, it is less likely for compilation errors to impede collaboration in shared notebooks than in collaborative code editors, as the design of Jupyter notebooks allows users to run individual cells. If one user writes a code cell with compilation error, other users can simply skip this cell and run other codes. We found the interference happened in shared notebooks mainly due to conflicts in shared data frames.

In addition to findings consistent with other studies of collaborative editing, collaborative editing in computational notebooks has its unique aspects. The mixed form of code and other types of media has distinguished computational notebooks from textual documents and pure code scripts. For instance, collaborative writing systems usually share static text synchronously whereas programming typically share their codebase through asynchronous Version Control Systems (VCSs)⁷. In shared notebooks, however, it remains unknown what the level of synchronicity should be (e.g., sharing static text and code, sharing the output, sharing the code interpreter), in part because of the emphasis on the sensemaking and experimentation processes.

Further, whereas it is common for programmers to segment code into modules based on their functions and eventually work on different files, data scientists rarely split their work into multiple notebooks. Thus, integrating version control locally [87] can be one potential solution to help collaborators track each others' edits.

3.7.2 Opportunities and Challenges of Collaboration in Computational Notebooks

Despite all the benefits of working in shared notebooks—encouraging more exploration and reducing communication costs—it is not easy to judge whether working in collaborative notebooks as currently designed is better than working on individual notebooks. For example, data science learners may find it more useful to work on a private notebook and to explore a task privately first before discussing the results with their collaborators. Reflecting on the context of collaborative writing, the common collaborative editing features

⁷Code editors that synchronize text content in real-time are more useful for learning and tutoring purposes where there are few lines of code and the cost of potential conflicts are minimal.

for writing include tracking changes for review, adding comments, adding access control for the whole document. Tools like Google Docs are designed to support more than real-time editing, and studies have found that users rarely edit the same piece simultaneously in practice [41]. How teams choose to use collaborative writing tools will depend on their goals and work preferences. For example, the “track changes” and comments features may be more useful when collaborators engage in the same document asynchronously. Thus, designers should take a user-centered design approach and reflect on different purposes of collaboration when extending the collaborative editing features to the context of notebook editing.

Our observational study explored one specific scenario where data scientists who did not know each other worked simultaneously over four hours to solve a predictive modeling problem. It may not be representative of all of all data science collaboration scenarios. Nonetheless, it is important as a first step to understand the challenges in current collaborative notebook editing features. We believe that some challenges can transfer to other collaboration scenarios. For instance, when collaborators edit the same notebook in a different time, they may still want more awareness information on what their partner is working on. Future work should explore how to generalize the design to serve the needs for various collaboration scenarios in real-world data science practice.

3.7.3 Design Implications

Studies have explored approaches to improve the infrastructure of computational notebooks for individual authoring and sharing (e.g., enabling content folding [156] and tracking exploration history [90]). Our study explicitly examined the benefits and challenges for multiple users to edit the same notebook collaboratively. The results suggest to the needs for a better collaborative notebook infrastructure. Below we discuss several design opportunities.

3.7.3.1 Improve Awareness of Collaborators’ Activity

Our current synchronous notebook editor design tracks and displays the locations of cursors of collaborators. However, sharing cursor information did not seem to be enough for users to perceive changes from others given a shared interpreter (back-end python process) state. Further, it is challenging to track the shared cursor when the notebook gets long. It would be valuable to explore what information requires high awareness and how

to present the awareness information—particularly for large and complex notebooks. For example, we may infer the context of the code that one person is working on from the nearest narrative text or headings and share the heading with other users. We may also broadcast important changes made to certain cells (e.g., cells that initiate variable names, or import libraries), or more explicitly share changes to the shared interpreter state..

3.7.3.2 Provide Access Control

We observed a strong tendency for participants to take ownership of the cells they worked on. Although the notebook and execution environment were shared, participants did not want others to edit their cells without permission. On the other hand, some participants did not want others to see their edits on a cell until they deemed it as “finished”. These findings suggest that access control mechanisms may be appropriate to integrate into the collaborative notebook infrastructure. We propose that there are at least two types of access control, which we called either *editing control* or *visibility control*. For editing control, we could imagine integrating the local versioning design [90] to protect a cell so that collaborators can submit their edits for approving. For visibility control, users could instead choose to disable the real-time synchronicity for certain cells or choose to fold the implementation details (e.g., [156]), thus hiding the work until it was ready.

3.7.3.3 Enable Discussions within Notebooks

Frequent communication is important for data scientists to stay updated on progress, reason about decisions, and coordinate work. Participants found it difficult to use third-party instant messaging tools because they had to constantly switch between applications. In addition, we observed that participants referred back to the shared notebook content often. This suggests that there is value in exploring the design of an in-notebook chat window. One potential benefit of such design is to support deictic references to a specific part of the notebook (e.g., [125]).

Moreover, there is rich information in chat messages; users report their progress, make plans, or explain parts of code. Investigating how to utilize the chat history to help users annotate the notebook may help moderate the tension between quick exploration and clear explanation [155].

3.7.4 Limitations

Since we only looked at one specific scenario of collaboration, our results and design implications may not effectively represent the needs for a better collaboration tool for other collaboration scenarios in data science. For example, the type of data science problems, the expertise of collaborators, team size, the synchronicity of the collaboration, and whether a final narrative is the end goal may all affect how users perceive the synchronous editing features. In addition, we scheduled the study sessions with remote participants at their convenience, which resulted in pairs more likely from the same region. Future work should explore broader collaboration settings and broader demographics.

3.8 Conclusion

We probed into how synchronous editing in computational notebooks might change the way data scientists collaborate on a predictive modeling task. Our survey findings highlight the tools and strategies that data scientists currently used in collaboration practice. Based on the design of current synchronous editing features in computational notebooks, our empirical observation reveals that working on the same notebook results in different collaboration styles compared to working on individual notebooks. The key findings suggest that synchronous editing tools improve collaboration by helping data scientists maintain a shared context and improve work efficiency. However, the current real-time collaborative editing features may lead to several problems (e.g., interference with each others' work, unbalanced contributions). The challenges in using the current real-time collaboration features suggest that we need better collaborative editing features for computational notebooks. We discuss how our results extend prior work on collaborative editing and how the HCI community can play a vital role in broadening the understanding of collaborative data science with a human-centered approach. Finally, we propose design implications to enhance synchronous editing in computational notebooks and to improve collaboration among data science workers.

3.9 Acknowledgements

We thank John Penington for his work creating a proof of concept implementation of the shared Jupyter notebook. We also thank Natalie Gross, Jamie Neumann, and Rebecca Parada for their help with coding and analyzing data from our second study. We thank all

of our participants across both studies and our reviewers for their valuable feedback. We also thank the Michigan Institute for Data Science (MIDAS). This material is based upon work supported by the National Science Foundation under Grant No IIS 1755908.

Part 2

Design Collaborative Computational Notebook Environments for Data Scientists

CHAPTER 4

Callisto: Contextualizing Shared Notebooks with Discussions

When teams of data scientists collaborate on computational notebooks, their discussions often contain valuable insight into their design decisions. These discussions not only explain analysis in the current notebook but also alternative paths, which are often poorly documented. However, these discussions are disconnected from the notebooks for which they could provide valuable context. We propose Callisto, an extension to computational notebooks that captures and stores contextual links between discussion messages and notebook elements with minimal effort from users. Callisto allows notebook readers to better understand the current notebook content and the overall problem-solving process that led to it, by making it possible to browse the discussions and code history relevant to any part of the notebook. This is particularly helpful for onboarding new notebook collaborators to avoid misinterpretations and duplicated work, as we found in a two-stage evaluation with 32 data science students.

4.1 Introduction

Data scientists benefit from collaborations to leverage expertise from each other and to improve the efficiency of their work. Computational notebooks are powerful tools for collaborative data science because they allow data scientists to document and replicate the exploration process through the creation of computational narratives—documents that combine code, explanatory text, and intermediate output. New tools like Google Colab [5] and Deepnote [16] enable data science teams to work in the same notebook in real time, creating new possibilities for collaboration.

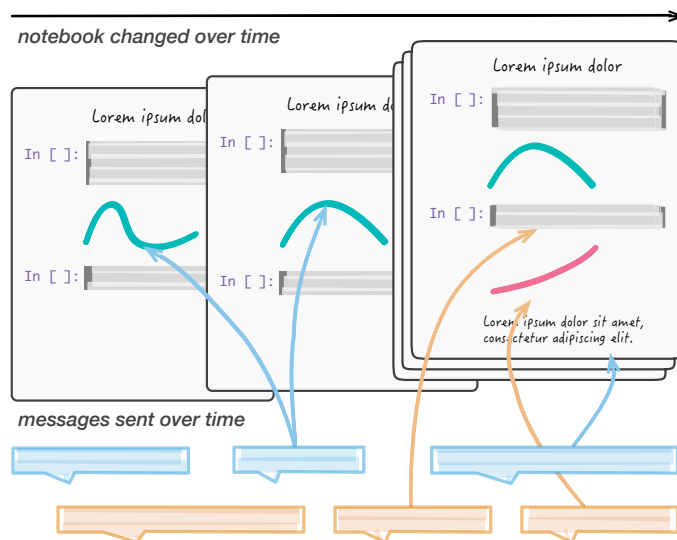


Figure 4.1: Callisto captures and stores contextual links between discussion messages and notebook elements with minimal effort from users.

Effective communication between data science team members is critical for productive teamwork. Collaborators need to understand what their teammates have done so far, what they plan to do, what they have given up, and how their work fits in with the team’s overall goals. Team members can improve their shared understanding by (a) writing clearer code, (b) documenting their work, and (c) discussing as a team. Improving code clarity (a) and writing clear documentation (b) are often impractical for data scientists, who frequently write makeshift code to experiment, explore, and test hypotheses [98, 181, 155, 158, 89].

Data science teams often have rich team discussions (c) through communication channels such as e-mail, instant messenger, and face-to-face meetings [181, 98]. These discussions are often crucial for collaborators to work together effectively, as they can provide valuable context about notebook authors’ goals and design rationales. However, these discussions are disconnected from the computational notebooks being discussed—they typically occur in channels outside of the notebook and references to notebook content are implicit (e.g., “there’s a bug in the second cell” or using a notebook screenshot). This means that team members typically need to have a shared context to make sense of the discussion (i.e., they need to understand what “the second cell” means or which part of the notebook a screenshot refers to). This can be particularly challenging as the notebook evolves (for example, if the “second cell” is moved after it is referred to). As a result, although discussions are helpful for collaborators who are actively involved in it, they can be difficult to understand for new team members or anyone catching up on the

discussion [203] who does not have this shared context.

In this paper, we propose to improve collaborative data science by connecting discussions with computational notebooks. We first describe the results of a formative study where we found that chat messages can be invaluable to understanding collaborative computational notebooks but are difficult to comprehend afterwards. We then introduce *Callisto*, a plugin for Jupyter [3]. We designed Callisto with the insight that *discussions are an integral part of collaborative computational notebooks* and connecting discussions with notebook content can make the notebook easier to understand for its authors and for subsequent readers. Callisto augments Jupyter with several collaborative features—most notably, the ability to explicitly reference notebook elements in chat messages. These connections make it easier to understand the context of a given message and to find discussions that are relevant to a specific notebook element.

We conducted a two-stage evaluation of Callisto. In the first stage, we evaluated how Callisto supports real-time team communication and found that it can reduce communication costs. In the second stage, we evaluated whether Callisto can help data scientists better understand a discussion that has already taken place. We found that Callisto can ease user onboarding to new notebooks by helping them understand the design rationales of its authors. As one of our participants put it, *“by reading the code, I know what they were doing. But with the chat messages, I can know what they were thinking.”*

This paper contributes:

- empirical evidence of the challenges that data scientists encounter when catching up with an ongoing group project,
- the design of Callisto with a set of features to make chat messages more useful for understanding the past exploration process in the notebook,
- empirical insights into how users engage with and perceive these features, and
- evidence that creating mappings between messages, notebook elements, and versions helps data scientists understand and follow up on the exploration pipeline.

4.2 Formative Study

To better understand how discussions can be useful for explaining the data-exploration process, we analyzed chat messages collected from three data science group projects. In doing so, we aimed to investigate three questions: (1) Why do collaborators send

messages to one another? (2) How do messages connect with the evolving notebook? and (3) What aspects of the notebook do collaborators talk about?

4.2.1 Method

We recruited six data science students from data science special interest groups in both university and online learning environments. We asked participants to work remotely in pairs on a beginner-level data science task¹ using a collaborative Jupyter editor for four hours. This collaborative Jupyter editor synchronizes edits between users and allows collaborators to see each other’s cursors. Pairs also worked in a shared runtime, meaning that code ran on a single interpreter, with outputs shared between collaborators. There were no other explicit communication mechanisms (chat, voice, etc.) enabled in the editor, and participants were given access to a third-party text chat (Slack) for communication. We collected chat messages, final notebooks, and screen recordings during the study.

4.2.2 Data Analysis

Our formative study uses a similar data analysis approach (in a different setting) to Yarman et al.’s work on cross-media referencing [199]. Two members of the research team used open coding to classify the collected data. We used the first 50 messages to create an initial code list, using final notebooks and video recordings as secondary evidence to help recall messages’ context. After discussing and merging the code list, the two members independently coded the same sample of 50 messages and achieved an agreement of $\kappa = 0.40$. We revised codes to reduce ambiguity and achieved an agreement of $\kappa = 0.83$ between raters after two rounds of iteration.

4.2.3 Results

In total, we analyzed 760 chat messages to better understand their purpose, their relationship to the evolving notebook, and the specific aspects of the notebook they mention.

4.2.3.1 Purpose

We found five broad purpose categories: reflection (244), planning (87), check-in (121), cooperation (67), and out-of-scope messages (244), as Table 4.1 shows. Messages could

¹<https://kaggle.com/c/house-prices-advanced-regression-techniques>

Purpose	Example	<i>n</i>
Reflecting	“This plot confirms the correlation for sure.”	244
Planning	“Let’s throw away columns that have lots of missing values.”	87
Check-in	“Just did a square root.”	121
Cooperation	“Ok, while you fix the stuff, I’ll create one hot encoding for categorical [variables].”	67
Out-of-scope	“Oh no!!”	244

Table 4.1: Purpose of sending a message: reflecting, planning, check-in, cooperation, and out-of-scope.

Relevance	Example	<i>n</i>
Ideas that were only discussed but never implemented	“Do you think we can just assign 1,2,3,4,5 to it and create one column instead?” (“No, I am afraid ...”)	29
Ideas that had not yet been implemented when the message was sent, but appeared in the notebook later	“How about we start with numerical columns?”	150
Ideas that had been implemented in the notebook when the message was sent, but did not appear in the final notebook	“Something went off. The MSE [mean square error] is huge.”	72
Ideas that had been implemented when the message was sent and appeared in the final notebook	“For the test data I did a fillna with 0.”	108

Table 4.2: Relevance between messages, the notebook history, and the final notebook.

Granularity	Example	<i>n</i>
Directly referred to a specific line of code	“I think LabelEncoder is going to treat NA as a new encoding.”	97
Directly referred to the output of a cell	“I am not too convinced if our MSE values are good enough.”	119
High-level ideas across multiple cells	“I just converted the categorical [data] to numerical [data].”	206

Table 4.3: Granularity: the level of detail of the referenced elements

fit into multiple categories, such as planning and cooperation. Messages coded as *reflection* tended to expand on the reasoning behind past decisions, while *planning* messages discussed potential features that had not yet been implemented. *Check-in* messages were updates between collaborators on what they had done, while *cooperation* messages generally discussed collaboration strategies.

These four categories show that chat messages can help explain the data-exploration process, describe the purpose of the code, and provide a high-level interpretation of the results. Not surprisingly, however, we categorized an additional 30% of messages as *out-of-scope*, because they did not convey useful information for explaining the exploration process and instead contained duplicate information or socialization messages.

4.2.3.2 Relevance

Our analysis (shown in Table 4.2) revealed that the final notebook is generally not reflective of the whole exploration process. Data scientists often explored ideas in discussions that they later rejected and wrote no analysis for. Even if they did implement an analysis to explore an idea, the code was often modified or removed during a “cleanup” stage [91, 75]. As such, messages between collaborators can fill in missing details of the exploration process. We suggest that revealing this information to new collaborators (e.g. someone taking over a project or another data scientist helping with an analysis) may avoid duplication of work while simultaneously revealing hidden assumptions. However, given that the Jupyter Notebook’s current design does not have built-in collaboration features nor track edit histories (which can give context to discussions), it is difficult to use chat messages to understand a previous exploration process.

4.2.3.3 Granularity

We also investigated the granularity of the notebook elements collaborators referred to (Table 4.3), finding that 97 messages directly referred to a specific line of code. These messages were related to API usage, debugging, or sharing the current status. A further 119 messages directly related to the output of a cell, including the data frame, visualizations, and statistical values. Finally, 206 messages described high-level ideas implemented across one or multiple cells.

4.2.4 Implications

We derive three design implications from our findings:

Chat messages are useful for explaining the exploration process. We were able to better understand the motivations for doing specific analyses, the purpose of the code written to run them, the interpretation of their results, and alternative analysis paths (tested or rejected without implementation). These details are often missing or poorly captured in traditional Jupyter Notebook artifacts.

Chat messages are difficult to follow. Chat messages are long and tedious to read because of scattered insights, a large amount of out-of-scope information, and information that requires notebook context to understand (which is likely to change before being finalized, as Table 4.2 shows). This makes it difficult for newcomers to build on earlier work.

Notebook elements are frequently referred to in chat messages. Notebook elements, such as fragments of code, output of executions, and cells containing a variety of statements, are frequently mentioned in chat messages. The lack of connection between these elements and discourse limits insight into decisions and results.

4.3 Design of Callisto

We designed Callisto to improve collaborative data science by better connecting discussions with notebook content. Callisto extends the Jupyter Notebook platform in several ways. First, it allows users to share notebooks, collaborate in real time, and discuss with collaborators. Second, it enables users to connect discussions with elements in the shared notebook, including code, output, individual cells, or edits. Third, it leverages these connections to make it easier to navigate discussions and notebook content—for example, to find discussions about a particular part of the notebook. We describe the design of each of these facets of Callisto in more detail below.

4.3.1 Enabling Sharing and Real-Time Collaboration

Although the creators of Jupyter recognized the importance of real-time collaboration, they left it as future work² [95]. Several offshoots of the Jupyter project [5, 16] have

²At the time of writing, Jupyter does not support real-time collaboration.

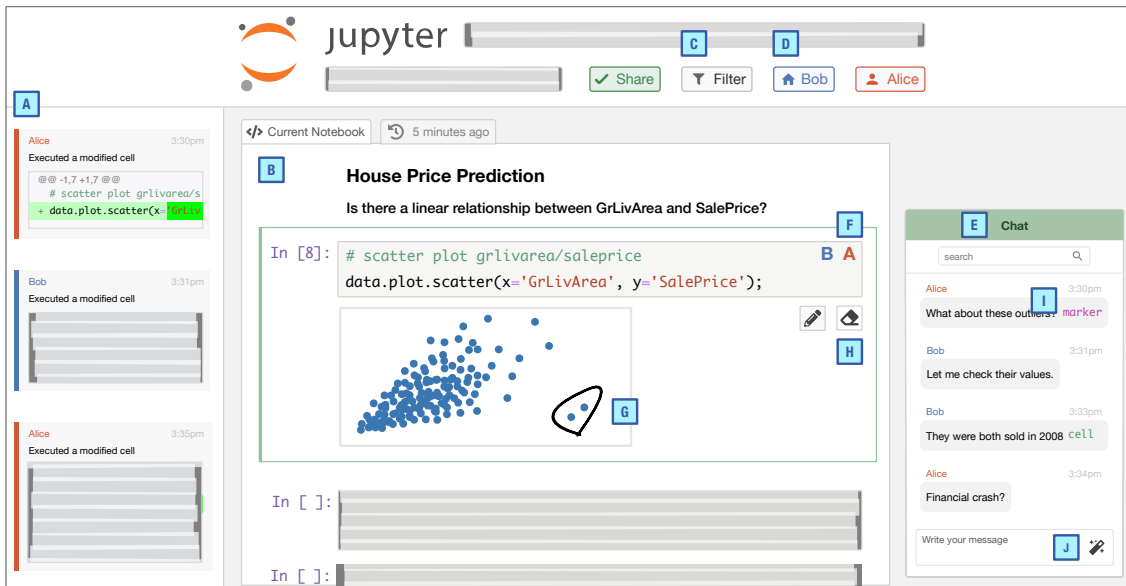


Figure 4.2: Overview of Callisto: (A) The changelog panel shows users’ edit histories; (B) The collaborative notebook editor synchronizes edits, runtime variables, outputs, annotations (see G, H), and cursors (see F) among collaborators; (C) The filter button enables the filtering mode (see Figure 4.3); (D) The user panel lists collaborators that are connected to the notebook. Users can navigate to others’ cursor locations by clicking on their name; (E) The embedded synchronous chat pane creates connections between messages and notebook content. Messages mapped to the selected cell are highlighted in light green. Users can create explicit references by clicking the magic wand (see J) and then selecting the relevant part of the notebook—for example, to create an annotation reference (see I).

incorporated collaborative features such as synchronized editing and shared cursors. Callisto starts by enabling notebook sharing and edit synchronization in Jupyter. We designed Callisto as a Jupyter plugin, rather than as a fork of the codebase, to allow users to easily share any standard Jupyter notebook and maintain compatibility with future versions of the Jupyter platform.

4.3.1.1 Basic Collaboration Features

The Callisto plugin augments the standard Jupyter UI with several widgets, as Figure 9.1 shows. First, Callisto adds a “share” button that generates a unique Uniform Resource Locator (URL) for collaborators to join the shared notebook session. A panel lists the collaborators that are connected to the notebook (Figure 9.1.D). When collaborators join the notebook, their edits are synchronized in real time with other collaborators. They can

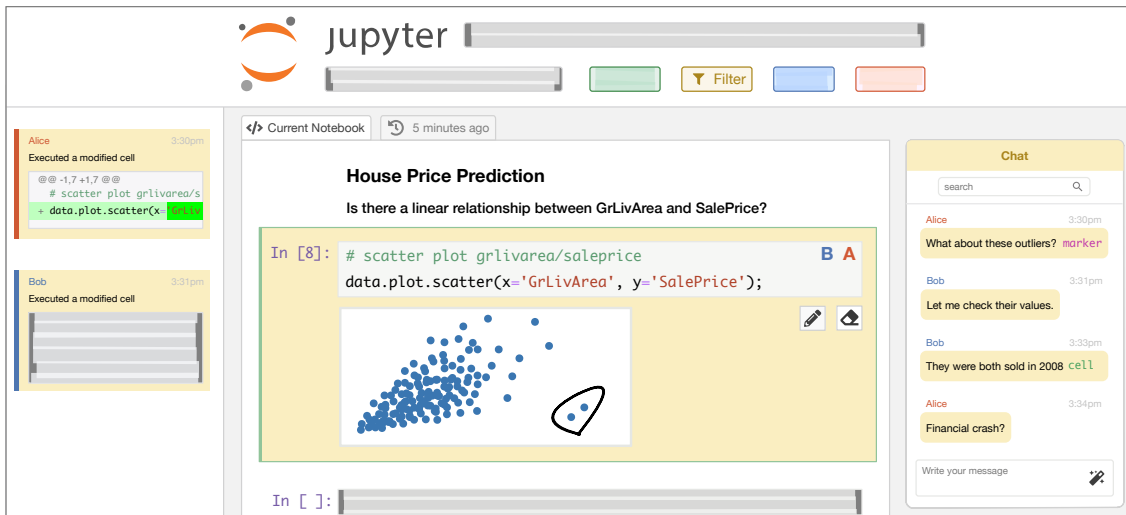


Figure 4.3: Filter Mode. When filter mode is enabled, it only displays messages and edits that are marked as relevant to the selected cell.

also see every other user’s cursor location and selection (Figure 9.1.F) and navigate to any other user’s location by clicking on their name in the list of collaborators (Figure 9.1.D).

4.3.1.2 Shared Runtime and Outputs

One important difference between computational notebooks and standard code is that computational notebooks are divided into smaller *cells* that can be run individually. Cells run in a common variable space, meaning that the ordering and timing of cell execution can (and typically does) influence execution outputs. This can be confusing for users, particularly in situations where one user’s output cannot be replicated by other users who have different runtime states. Thus, rather than giving users their own runtime, Callisto connects every collaborator to a single shared runtime. This means that the state of the program is shared—if the value of a variable is modified (by executing code that modifies its value), its value is updated for every collaborator. Cell outputs (the results of running a cell, which can be textual, graphical, or shared data frames) are also shared automatically, which gives all collaborators a shared point of reference.

4.3.1.3 Synchronous Chat

Jupyter does not have built-in messaging features, which means that data science teams typically communicate through external tools such as e-mail or Slack [181]. As we found in our formative study, these communications can be valuable for understanding the de-

sign behind a notebook, but there is a cost in switching between applications for writing and communicating. Thus, Callisto embeds a synchronous chat pane directly in the shared notebook (Figure 9.1.E). This built-in chat pane allows us to capture contextual information and create connections between messages and notebook content, which we will introduce later.

4.3.1.4 Edit and Version History

Prior work has found that shared editors and cursors are helpful for collaborators but are not enough to build awareness of what they have worked on [181]. This is partly because they only allow users to see what collaborators are working on *at that specific moment*. Building awareness of collaborators' activity instead requires a more complete view of their actions. To provide this, Callisto includes a panel showing users' edit histories (Figure 9.1.A). This panel shows a history of notebook versions and a preview of user edits (which are displayed as diffs—additions and deletions from the previous snapshot). Every user action (such as cell edits, deletions, insertions, and executions) is recorded and displayed for users to see and better understand what their collaborators are working on. This panel also allows users to check notebook diffs in a complete view (by clicking on any diff summary), which will show the code and output differences (see Figure 4.4).

4.3.2 Connecting Messages and Notebook Content

As we found in our formative study, data scientists often refer to the computational notebook in their discussions. Prior work [125] has proposed enabling chat messages to refer to regions of code. However, our study participants referenced more than code; they referenced program output (which can be graphical or textual) and specific notebook cells. They also referenced things that were not explicitly part of the computational notebook, such as prior notebook versions or code edits themselves (e.g., “*I made this change...*” referring to edits they made to fix the buggy code). These references were implicit; they required readers to infer what they referred to. Callisto is the first system to explicitly encode these references.

Encoding connections between messages and notebook content allows users to give their messages clear context and can make the computational notebook easier to interpret and navigate for future readers. Inspired by our formative study analyzing the granularity of the notebook elements collaborators referred to, messages make five types of references

to notebook elements in Callisto:

- *code references* are associated with a specific range of code at the time when that reference was created,
- *cell references* are associated with a cell in the notebook,
- *snapshot references* point to a previous notebook version,
- *annotation references* allow users to refer to a specific portion of output (images or tables) by drawing annotations on that output and referencing those annotations, and
- *diff references* point to an edit in the notebook.

References can either be created explicitly by users or inferred by Callisto through context (as we describe in more detail below). Users explicitly create references by clicking the chat input’s magic wand (Figure 9.1.J) and then selecting the relevant part of the notebook or version history panel.

4.3.2.1 Automatically Inferring References from Context

Although explicitly creating references requires little overhead (clicking the “edit link” button (Figure 4.5.B) and then the relevant part of the notebook), we built features to further reduce the effort required by automatically inferring references from users’ work context—the cell that is currently selected or that they are editing, which their message likely pertains to. Although active collaborators might have no trouble decoding these messages’ context (possibly by looking at where that user’s cursor currently is), it can be more difficult for future collaborators as they catch up on prior discussions. Thus, Callisto automatically attaches a cell reference to the currently selected cell if users do not add an explicit reference.

This method of inference might produce erroneous references. For example, if the purpose of the message is planning, the message might relate to the cell that the user is going to edit, instead of the cell he just edited. However, we believe false negatives (when relevant context is not captured) are much more costly than false positives (when the context captured is not relevant) for users, as it is easier to ignore extraneous information than to recover missing information. Users can also manually correct errors from automatic inferences.

4.3.3 Navigating Messages and Notebook Content

By connecting messages and notebook content, Callisto gives a richer context to notebook elements and makes it easier to understand prior discussions. This can be helpful for both current collaborators and future readers. There are two broad uses for these connections: to understand the context of a given message (from messages to relevant notebook content) or to find the part of the discussion that is relevant to a specific part of the notebook (from notebook content to relevant messages). The former demonstrates “what changes were made” while the latter explains “why changes were made” [172].

4.3.3.1 From Messages to Notebook Content

While collaborating, data scientists often need to determine which part of a notebook a given message pertains to. Active collaborators and users reading past discussions benefit from certainty about a given message’s context. In order to help build this context for messages, Callisto allows users to navigate from a reference in the chat panel to the relevant part of the notebook. References in the discussion panel appear like Web links. When a user clicks on the reference, Callisto highlights the relevant elements in the notebook (and scrolls to them if necessary). Messages might become “out of date” if they reference an element of the notebook that is later modified or deleted. To ensure references stay relevant, Callisto automatically “backtracks” references; if a user clicks on a reference to an element that was later changed, Callisto shows them the referenced content in a snapshot view.

Subsequent notebook readers might also want to understand how the content of the notebook changed as the discussion moved on—what collaborators were doing between messages. To allow readers to understand how the notebook evolved through the discussion, Callisto enables them to compute the difference between any set of notebook versions. For example, if a user selects two chat messages, a diff button will appear in the chat panel, as Figure 4.5 shows. This will trigger Callisto to render the code and output differences between the state of the notebook when each of those messages was sent.

4.3.3.2 From Notebook Content to Messages

Computational notebooks are often shaped by many design decisions, failed experiments, and progressive iteration. For collaborative computational notebooks, explanations of

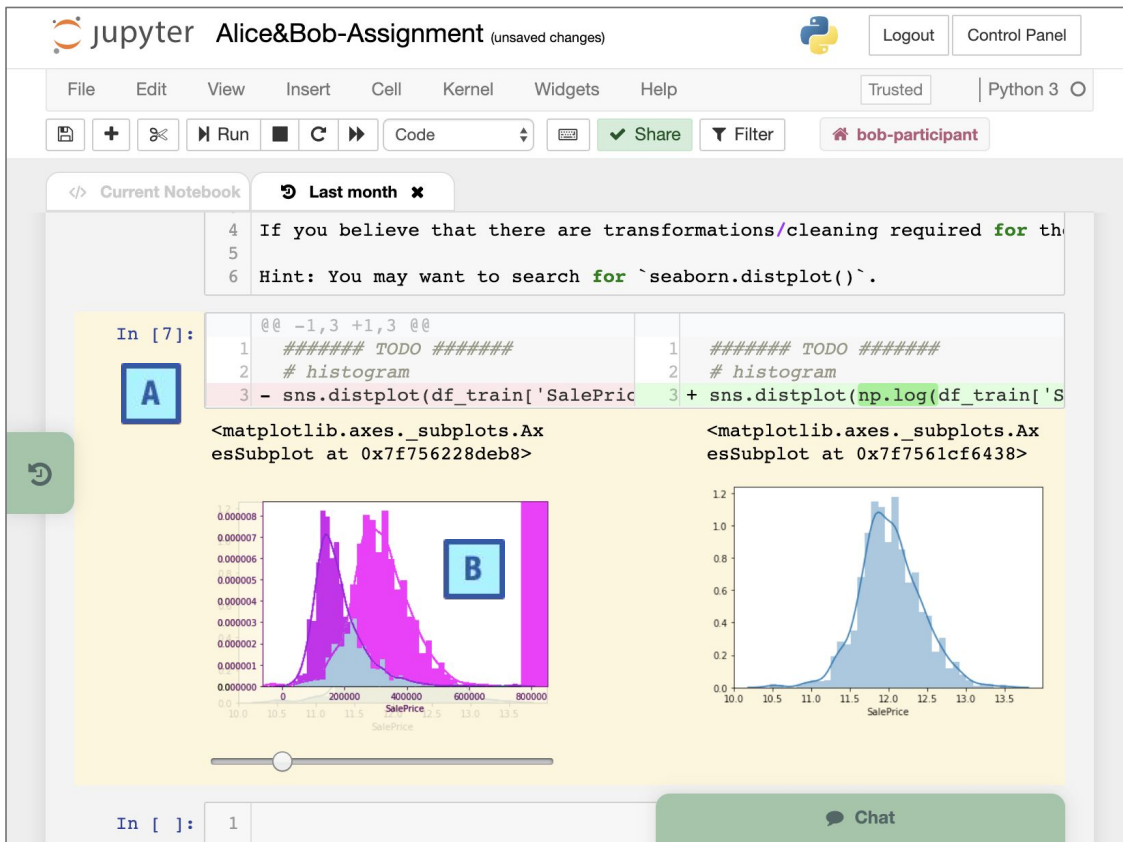


Figure 4.4: Diff View. Code differences (see A) and output differences (see B) are highlighted in a diff view. The new and old outputs are overlapped for comparison: hovering the mouse over the output will highlight the difference in purple and pink; the slider underneath controls the transparency between new and old output.

why the notebook ended up the way it did can often be inferred through careful examination of discussions between collaborators. By linking notebook content to discussion messages, Callisto allows users to see which parts of a discussion are relevant for a particular part of the notebook. As Figure 4.3 shows, users can click on a cell to display relevant discussions.

4.4 Evaluation

We designed a two-stage evaluation study with 32 data science students to assess how Callisto assists new collaborators when joining the collaborative notebook. We first observed participants working in pairs on a data science task in real time to test Callisto's usability (the *real-time collaboration study*). We then conducted a comparison study with

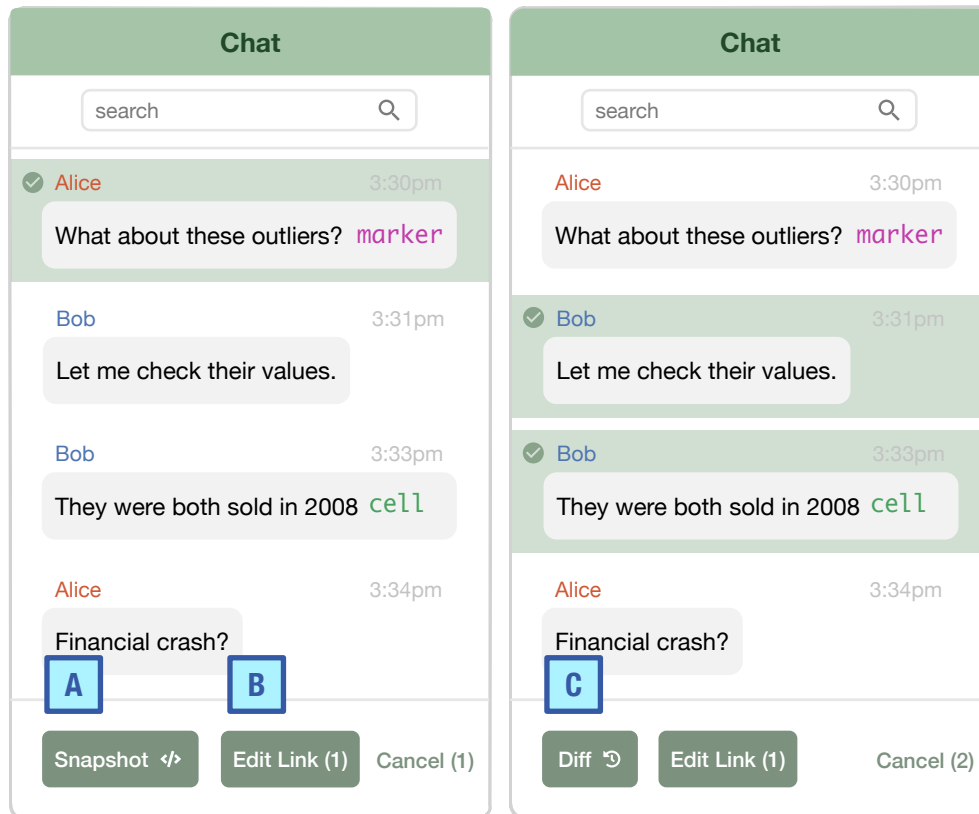


Figure 4.5: Chat Panel. When selecting one message, a snapshot button (see A) will navigate users to the snapshot of the notebook. When selecting two messages, a diff button (see C) will navigate users to the diff view comparing two snapshots (see Figure 4.4). Users can manually refine the links using the edit button (see B).

a third individual joining the shared project using Callisto or a lite version of the system with no contextual links (the *follow-up study*).

4.4.1 General Study Protocol (for Both Stages)

The real-time collaboration study and the follow-up study follow a similar study protocol. We invited each participant for a 90-minute lab session. Before the study, participants reported their data science backgrounds on a pre-task questionnaire. Each participant was given a 15–20-minute training session on the tool, with example tasks to complete. After the study, we conducted a 10–15-minute semi-structured interview with each participant. We collected data from server-side usage logs, screen recordings, and post-task interviews. We also took observational notes during the study.

4.4.2 Participants (for Both Stages)

We reached out to data science programs and interest groups on campus, filtering qualified participants based on the courses they had taken and other data science-related experience. Overall, qualified participants were familiar with Jupyter Notebook, Python, and common exploratory data analysis packages (e.g., Pandas, NumPy). Most of them had experience of collaborating on an exploratory data analysis project.

We recruited 32 participants in total (11 female, 20 male, 1 non-binary, average age = 25). Participants were from a variety of data science-related programs (8 undergraduate students, 5 master’s students, 18 Ph.D. students, and 1 full-time employee who recently graduated; students’ majors included computer science, information science, health information, statistics, and economics).

Based on participants’ prior knowledge, we rated their experience level as beginner³ (n=6), intermediate⁴ (n=10), or expert⁵ (n=16). We randomly assigned participants into one of the two stages with a balanced distribution of experience level. There was no overlap in participants across study stages. We compensated participants with \$25 United States Dollars (USD) gift cards.

4.4.3 Stage 1: Real-time Collaboration

In Stage 1, we investigated the perceived usability of Callisto for real-time collaboration. We observed eight participants (P1–P8) working in pairs to solve a data science task together using the full version of Callisto. Participants were invited to the study site at the same time and sat in separate rooms. We informed participants at the outset that they would not have enough time to complete the task and a new collaborator would take over the remaining work.

The data science task was modified from a Kaggle competition (predicting house sale price). To scope the task within the study duration, we asked participants to only perform exploratory data analysis. We provided a basic framework of the notebook for participants to begin with, as well as example API usage code for common data analysis packages.

³**Beginner:** has taken 1–2 data science classes, basic experience with Pandas and Python, but little experience with data science problems

⁴**Intermediate:** limited experience with data science problems

⁵**Expert:** is familiar with libraries frequently used in data science, and very experienced in solving data science problems

S1 Real-time Collaboration (4 pairs)	\bar{x}	σ
Cell Edits	64.50	34.57
Messages	101.00	76.40
Creating Pointers	7.25	1.71
Making Annotations	11.25	7.45
Erroneous References	8	15.3
S2 Follow-up (10 individuals)	\bar{x}	σ
Clicking on Pointers	8.40	4.51
Viewing Notebook Snapshot	10.20	5.45
Viewing Notebook Diffs	15.30	7.60
Inspecting Cells for Curated Messages	44.80	48.93
Inspecting Messages for Related Cells	40.00	20.95

Table 4.4: Server-side Usage Logs (mean: \bar{x} , standard deviation: σ): (1) Most contextual links were created by inferred references; (2) The two navigating features were used equally to understand past decisions.

4.4.3.1 Overall Usage

As Table 4.4 shows, each group edited the notebook an average of 64.50 times during the 45-minute exploration. We recorded a cell editing event when a cell being executed was modified from its last execution. Participants frequently used the annotation feature (11.25 times per group) when discussing outputs. However, not all annotations were used for creating references. In fact, only 7% of the messages contained references that participants manually created. Most of these manually created references (26 out of 29) were cell pointers.

4.4.3.2 Creating References (Manual and Automatically Inferred)

In most cases where participants manually created a reference in a message, they used cell pointers with the default textual description “cell”. Participants gave a variety of reasons for using cell pointers over other pointers, including that discussions are often not around a particular piece of code, and that pointing to a cell requires less effort than creating other pointers. Participants also mentioned that the ability to check their collaborator’s cursor served the same function as the pointer when they were talking about code cells in real time:

I can know my collaborator’s cursor so it is easy to know what she is talking

about. So we didn't use much references, only a few cell links. (P3, expert)

We identified five cases where participants could have used references to make their communication more efficient. For example, one participant could have directly pointed to a number in the table by creating an annotation reference, but instead he described it as *“the two values with bigger GrLivArea as rows with IDs 1182 & 691”*. Worse still, some participants described locations relative to their field of view, which further increased the difficulty for new collaborators to parse the message (*“If you scroll up to the cell above, it looks like the ID is always one higher than the Pandas index.”*).

Because participants created relatively few manual references (7.25 out of 101 messages on average per group), Callisto's ability to automatically infer relationships between messages and notebook cells is crucial. In order to understand how well Callisto's reference inference feature works, we manually checked the messages and found that 92% of messages were connected to the correct context (only a total of eight messages were mismatched with the inferred cell references).

4.4.3.3 Annotations Aid Communication

Participants used the annotation feature frequently, and we investigated its popularity in the post-task interview. Most participants agreed that the annotation feature reduced communication costs:

A lot of our discussions are about the graphs. I really like the ability to draw on the graphs so we knew what exactly we were talking about. (P4, intermediate)

[When using Slack] I have to make a screenshot and save it on desktop. I do not like saving too many images on the desktop so I like this tool. (P1, beginner)

4.4.4 Stage 2: Following up with the Collaboration Process

In Stage 2, we evaluated how a new collaborator better followed up with an ongoing collaborative project. We compared two versions of Callisto in this stage: a *lite version* where no contextual links are captured and stored, only basic collaboration features are enabled; and the *full version*.

4.4.4.1 Content Preparation

We designed the assets (the notebook history, chat messages, and their connections) for the “ongoing collaborative project” by merging and modifying the collaboration assets produced in Stage 1. The combined project used in Stage 2 contained 42 cell edits, 132 messages, and 19 manual references. In the lite version of Callisto, we replaced the manual references with a textual description of the location in the notebook. To ensure these textual descriptions were realistic, we observed two more groups (in addition to the pairs described in the previous section) doing tasks in Stage 1 using the lite version. We identified several strategies that participants used to point to notebook elements, and replaced the references based on the three most common strategies: cell execution number, pasting the content directly, and describing the location of the content.

4.4.4.2 Study Setup

We recruited 20 participants and randomly assigned them to one of the two conditions: the experimental condition using Callisto, and the control condition using the lite version. We informed participants that the previous collaborators (Alice and Bob) were in a rush and did not finish the exploration.

We asked participants to explore the notebook and answer five questions⁶ related to Alice and Bob’s prior analysis. The questions were designed using Revised Bloom’s Taxonomy (RBT) to assess participants’ understanding [102]. For example, *outlining* features that Alice and Bob have explored, and *summarizing* their findings about the distribution of sale price. Participants had six minutes to read details from the notebook and answer each question. We collected the answers and measured time and participants’ self-reported confidence level (on a seven-point Likert scale) for each question. At the end, we gave participants 10 minutes to use the tool in depth to follow up on their work (e.g., clean the notebook, add more explanations, or continue exploring the problem).

To assess how well participants understood the ongoing collaboration process, we designed a rubric to grade their answers to the five questions (maximum score = 50). Two external data science experts independently graded their answers. We performed a Pearson correlation coefficient test and found a strong agreement on the rating ($r = 0.97$, $p < 0.001$).

⁶See supplementary materials for the full rubric and set of questions.

		Callisto	Control
Questionnaire Score*	\bar{x}	30.05	23.75
	σ	6.27	3.85
Time (sec)	\bar{x}	1577.21	1416.05
	σ	237.35	320.28
Self-reported Confidence	\bar{x}	5.00	5.26
	σ	0.66	1.16

Table 4.5: Comparing the outcomes from the second stage of the evaluation (mean: \bar{x} , standard deviation: σ). Callisto helps new collaborators achieve a better understanding of an ongoing project.

4.4.4.3 Overall Performance

As Table 4.5 shows, participants in the experimental condition (avg = 30.05) achieved a higher score than participants in the control condition (avg = 23.75), with a two-sample t-test suggesting that the difference is significant ($p = 0.014$). There was no significant difference in the time costs or the self-reported confidence level between the two conditions.

To investigate why participants performed better in the experimental condition, we studied their usage logs and screen recordings. As Table 4.4 shows, participants in the experimental condition used the two navigating features in Callisto equally to understand past decisions and discussions.

4.4.4.4 Understanding Discussions Around the Cell

Participants in both conditions reported a need to check the chat messages even though the notebook already contained some code comments and explanatory texts. They complained that the code comments were not well written:

Some comments are hard to parse. (P22, expert, experimental condition)

They could have used the markdown cells more to conclude the results. (P16, expert, experimental condition)

Comparatively, participants in the control condition found it difficult to follow the chat messages due to the sheer quantity. We observed that three participants in the control condition misaligned the chat messages with the notebook content when answering a ques-

tion about how Alice and Bob analyzed the linear relationship between `SalePrice` and `YearBuilt`. They answered the question incorrectly because they described the discussions about the linear relationship between `SalePrice` and another feature (`GrLivArea`, which appeared earlier in the analysis). Two participants in the control condition wanted chat messages to be mapped with notebook content:

I wish there is a way to attach the messages to the cell that they discussed. It will save me time. (P30, intermediate, control condition)

While participants in the experimental condition benefited from the established connection between messages and notebook, they further reported the filter feature helpful in curating discussions around cells. On average, each participant inspected cells 44.8 times to filter related messages, checking 14.9 unique cells. In addition, we observed that most participants (9 out of 10) preferred to keep the filtering mode enabled as they dove deeper in the notebook:

Because the chat is so long, I think it is not useful until I filter it down. (P18, expert, experimental condition)

4.4.4.5 Understanding the Context of the Message

We observed participants used the contextual links in Callisto the other way (from messages to notebook content) to understand the context of a message. Participants inspected messages 40 times to check related cells in the final notebook, or perform further actions such as checking snapshots (10.2 times) or comparing diffs (15.3 times). 27.4 unique messages were inspected by each participant, indicating that participants may go back and forth to check messages and related cells.

We further investigated why checking and comparing notebook edits from messages helped participants better understand the analysis from observation notes and screen recordings. We illustrated one interesting case of how participants approached the answers in the questionnaire differently. One question asked how Alice and Bob analyzed the outliers in the `GrLivArea`. Participants from the experimental condition were able to find all relevant analyses with two alternative hypotheses, where the code cell for testing one hypothesis was overwritten by the code cell that tested the second hypothesis. However, most participants from the control condition only reported the second hypothesis on the final notebook.

In addition, Callisto helped participants better understand how a code change resulted in an output change. As shown in Figure 4.4, Alice and Bob applied a log transformation to correct the distribution of `SalePrice`, only commenting vaguely on the results in the chat (“*the result looks much better*”). Participants in the experimental condition were able to compare the notebook diffs between this message and the one above, gaining an intuitive comparison of how the output changed from the diff view (see Figure 4.4.B). In contrast, participants in the control condition needed to first guess what might have changed in the notebook, then revert changes (e.g., remove the `np.log`) and execute the cell to compare the output.

4.5 Discussion

Reflecting on Callisto’s design, we discuss how future tool builders of computational notebooks and data science researchers can build on our work.

4.5.1 Reducing the Burden of Communication

Our findings revealed that participants in the real-time collaboration setting are hesitant to make accurate and polished references, or to create references, even if the interaction takes only two clicks. This corresponds to studies in other domains that report user reluctance to write quality annotations or comments during active work [27, 113, 22]. Future work should consider optimizing this process by designing shortcuts or providing suggested references inferred by edits in the notebook (e.g., a newly added annotation).

As prior work [181] shows, data scientists use a variety of communication tools, including high-bandwidth communication channels such as video conferencing or face-to-face meetings. Capturing information exchanged in these channels is difficult yet important to reduce the burden of text-based communication. It is worth studying the benefits and challenges of using different communication channels in data scientists’ daily work to leverage past discussions for a better understanding of shared work.

In addition, we believe similar techniques could work in other domains where remote collaborators co-design a shared artifact that changes over time, as long as the reference types are domain appropriate. For example, Callisto’s features could be adapted for a shared CAD tool where designers collaborate on a 3D model, but our results and designs may not apply for highly modular work (such as multiple authors writing different chapters of a textbook with minimal interaction).

4.5.2 Improving the Accuracy of Contextual Links

As most of the messages (around 93%) relied on inferred references, we believe that it is important to explore ways to further improve the accuracy and recall of inferred references. Mismatched contextual links happened for several reasons. If a message describes a future action, the relevant cell may not exist when the message is sent. In this case, we may consider using Natural Language Processing (NLP) techniques to infer whether the message should be connected to the cell edited before sending the message or the cell edited after sending the message. Another possible reason is that a message might reference a cell the writer’s collaborator is working on, instead of the one the writer is working on. It is worth exploring other strategies (e.g., considering common cells that nearby messages connect to) to automatically infer the context.

4.5.3 Towards Generating Meta-Narratives

New collaborators not only need to understand the computational narrative itself but also how that narrative evolved—the *meta-narrative* behind the narrative. Callisto is a representation of meta-narratives for computational notebooks. Creating an explicit meta-narrative object can be useful for onboarding new collaborators during the data-exploration process, as we found in our evaluation. These meta-narratives could also be useful in education; many programming lectures involve creating a form of meta-narrative. They could also be used in “traditional” writing. Future research could explore alternative representations for meta-narratives for a variety of domains.

4.5.4 Limitations

Callisto is designed and evaluated in the scope of within-notebook collaboration, where collaborators work in the same notebook and treat the final narrative as an end goal. The setup of the formative study is designed to encourage real-time chatting and collaboration, which may not be an accurate representation of most collaboration and communication scenarios. In addition, our in-lab evaluation contains several limits to external validity: participants are all students from the authors’ home institution; participants may not be proficient enough in Callisto given the short training time; we only evaluated one type of data science problem and provided the framework of the notebook rather than asking them to start from scratch.

4.6 System Implementation

Callisto⁷ consists of two Jupyter Notebook extensions—one small extension for Jupyter’s file browser (to make it easier to join shared notebooks) and the “main” extension for Jupyter Notebooks (Figure 9.1)—and a Node.js backend. Callisto keeps collaborators in sync (including notebook content, chat, lists of collaborators, and runtime state) through OTs, as implemented through ShareDB [2]. To maintain connections between messages and cells, Callisto tracks the edit history through the lifetime of the notebook and stores a unique id in the metadata of each cell, which stays constant as cells are inserted, deleted, and rearranged.

4.7 Conclusion

In conclusion, we have proposed the design of Callisto to leverage valuable chat messages in collaborative data science. Our two-stage evaluation study with 32 data science students confirmed that Callisto eases new-collaborator onboarding by helping them understand the design rationales of the notebook’s authors. In particular, Callisto successfully captures contextual links during the real-time collaborative creation of the notebook without hindering exploration, while the establishment of contextual links and the set of interactions for navigating the notebook significantly improve new notebook collaborators’ understanding of past discussions and decisions.

4.8 Acknowledgments

We thank all of our participants and our reviewers for their valuable feedback. We also thank the Michigan Institute for Data Science (MIDAS). This material is based upon work supported by the National Science Foundation under Grant Numbers IIS 1755908 and EHR 1915515.

⁷Callisto’s source is available at github.com/littleaprilfool/callisto

CHAPTER 5

Themisto: AI-Assisted Data Science Code Documentation

Computational notebooks allow data scientists to express their ideas through a combination of code and documentation. However, data scientists often pay attention only to the code, and neglect creating or updating their documentation during quick iterations. Inspired by human documentation practices learned from 80 highly-voted Kaggle notebooks, we design and implement Themisto, an automated documentation generation system to explore how human-centered AI systems can support human data scientists in the machine learning code documentation scenario. Themisto facilitates the creation of documentation via three approaches: a deep-learning-based approach to generate documentation for source code, a query-based approach to retrieve online API documentation for source code, and a user prompt approach to nudge users to write documentation. We evaluated Themisto in a within-subjects experiment with 24 data science practitioners, and found that automated documentation generation techniques reduced the time for writing documentation, reminded participants to document code they would have ignored, and improved participants' satisfaction with their computational notebook.

5.1 Introduction

Documenting the story behind code and results is critical for data scientists to collaborate effectively with others, as well as their future selves [89, 138, 104, 118]. The story, code, and computational results together construct a computational narrative. Unfortunately, data scientists often write messy and drafty analysis code in computational notebooks as they need to quickly test hypotheses and experiment with alternatives. It is a tedious

process for data scientists to then manually document and refactor the raw notebook into a more readable computational narrative, thus many people neglect to do so [157].

Many efforts have sought to address the tension between *exploration* and *explanation* in computational notebooks. For example, researchers have explored the use of code gathering techniques to help data scientists organize cluttered and inconsistent notebooks [75], as well as algorithmic and visualization approaches to help data scientists forage past analysis choices [88]. But these efforts focus on the cleaning and organizing of existing notebook content, instead of creating the new content. Another work developed a chat feature that enables data scientists to have simultaneous discussions while coding in a notebook [184], and linked their chat messages as documentations to relevant notebook elements as in Google Docs [186]. However, these chat messages are too fragmented and colloquial to be used for documentation; besides, in real practice data scientists and business analysts rarely work on notebooks at the same time and actively message each other.

We began our project by asking, “What makes a well-documented notebook?” To answer this question, we first conducted an in-depth analysis of how human data scientists document notebooks. Publicly shared user notebooks on Githubs are often not well documented [157], thus we look up to a special set of notebooks – the highly-voted notebooks users submitted to Kaggle competitions. We conducted a formative study with a sample of 80 of these notebooks, and our interative indepth coding analysis suggested these 80 notebooks have much better documentations in comparison to the corpus reported in previous literature [157]. Thus, we refer to them as “well-documented” notebooks.

Our coding process of these 80 notebooks also revealed a taxonomy of nine categories (e.g., Reason, Process, Result) for the documentation content, which reflects the thought processes and decisions made by the notebook owner. These findings together with the insights from related work motivate us to consider AI automation as a potential solution to support the human process of crafting documentation.

We propose Themisto, an automated code documentation generation system that integrates into the Jupyter Notebook environment. To support the diverse types of documentation content and to complement the AI limitations, Themisto incorporate three distinct approaches: a deep-learning-based approach to automatically generate new documentation for source code (fully automated); a query-based approach to retrieve existing documentation from online API websites for third party packages and libraries (fully automated); and a prompt-based approach to give users a start of the sentence and encourage

them to complete the sentence that serves as documentation (semi automated).

We evaluated Themisto in a within-subjects experiment with 24 data science practitioners. We found that Themisto reduced the time for data scientists to create documentation, reminded them to document code they would have ignored, and improved their satisfaction with their computational notebooks. Meanwhile, the quality of the documentation produced with Themisto are about the same as what data scientists produced on their own. Base on these findings, we re-imagine that the code documentation task can be conducted in a Human-AI Collaboration fasion in the future, where this joint effort may have unique advantages in comparison to the solo effort of a human alone.

Our paper provides a three-fold contribution to the HCI and data science practitioner communities:

- providing an empirical understanding of best practices of how humans document a notebook through an analysis of highly-rated Kaggle notebooks,
- demonstrating the design of a human-centered AI system that can collaborate with human data scientists to create high-quality computational narratives,
- reporting empirical evidence that Themisto can collaborate with data scientists to generate high quality and highly-satisfied computational notebooks in much less time.

5.2 Formative Study

In order to build a useful system that can support data scientists to create documentation and improve their computational narrative quality, we first need to explore and understand the characteristics of good documentation in high-quality notebooks. **What does a well-documented computational narrative look like?** We identify “well-documented” computational narratives with ratings from a broader data scientist community (Kaggle), and analyze their characteristics specifically around the documentation. We consider the community voting number is a good indicator to reflect a computational notebook’s quality for our research goal. Based on this premise, we then conduct a formative study to analyze the characteristics of a set of most voted computational narratives, and explore how the data scientists create documentations for these notebooks.

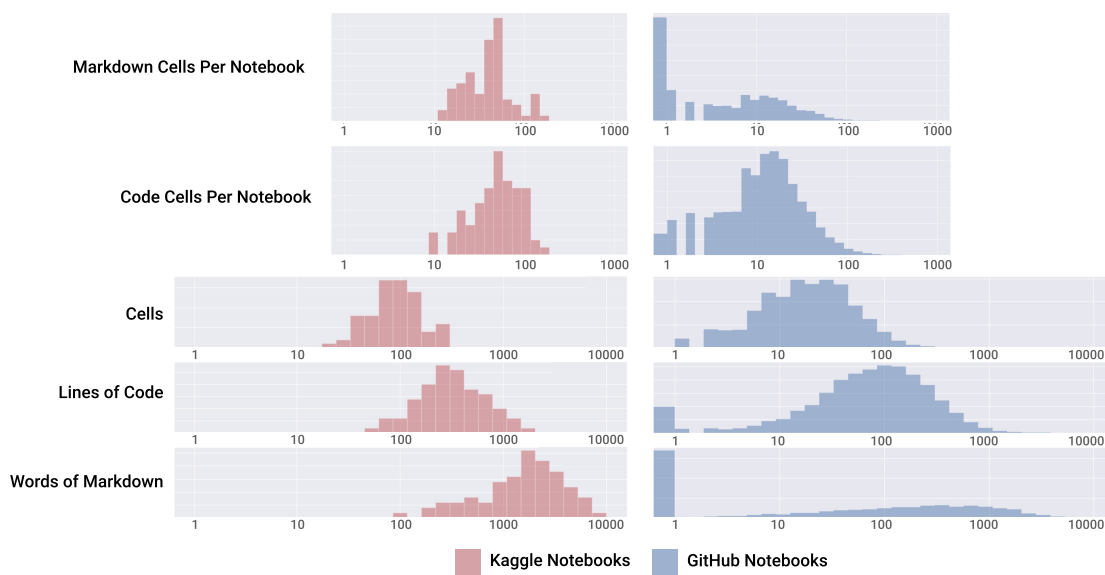


Figure 5.1: We replicated the notebook-level descriptive analysis by Rule et al. [157] to the 80 well-documented notebooks on Kaggle. The left side represents the descriptive visualization of the 80 well-documented computational notebooks from Kaggle (noted as Sample A) and the right side represents the descriptive visualization of the 1 million computational notebooks on Github (noted as Sample B). The highly-voted notebooks on Kaggle are better documented compared to the Github notebooks.

5.2.1 Data Collection

We collected notebooks from two popular Kaggle competitions — House Price Prediction¹ and Titanic Survival Prediction². We chose these two competitions because they are the most popular competitions (5280 notebooks submitted for House Price and 6300 notebooks submitted for Titanic Survival) and because many data science courses use these two competitions as a tutorial for beginners [23, 57].

We collected the top 1% of the submitted notebooks from each competition based on their voting numbers, which resulted in 53 for House Price and 63 for Titanic Survival. We then filtered out the notebooks that were not written in English and the ones that are not relevant to the particular challenge (e.g., a computational notebook as a tutorial on how to save memories can win lots of votes, but it is not a solution to the challenge), which returned 80 valid notebooks for analysis (39 for House Price and 41 for Titanic Survival).

¹<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>

²<https://www.kaggle.com/c/titanic/>

5.2.2 Data Analysis

Five members of the research team conducted an iterative open coding process to analyze the collected notebooks. Differing from [157], where their qualitative coding stopped at the notebook level, our analysis goes deep to the cell granularity: we code each cell’s purposes and types of content; and which step (stages) in the data science lifecycle that the cell belongs (e.g., data cleaning or modeling training [204, 187]). Our analysis covered 4427 code cells and 3606 markdown cells within the 80 notebooks. Each notebook took around 1 hour to code as we coded the notebook at the cell level.

Each coder independently analyzed the same six notebooks to develop a codebook. After discussing and refining the codebook, they again went back to recode the six notebooks and achieved pair-wise inter-rater reliability ranged 0.78-0.95 (Cohen’s κ). After this step, the five coders divided and coded the remaining notebooks.

5.2.3 Results

We found that these 80 well-documented computational notebooks all contain rich documentation. In total, we identified **nine categories for the content** of the markdown cells. In addition, we found the markdown cells covered **four stages and 13 tasks** of the data science workflow [187]. Note that a markdown cell may belong to multiple categories.

5.2.3.1 Descriptive statistics of the notebook.

We found that on average, each notebook contains 55.3 code cells and 45.1 markdown cells. We replicated the notebook descriptive analysis that Rule et al. used to analyze 1 million computational notebooks on Github [157]. As shown in Figure 5.1, the left side represents the descriptive visualization of the 80 well-documented computational notebooks from Kaggle (noted as Sample A) and the right side represents the descriptive visualization of the 1 million computational notebooks on Github (noted as Sample B). We found that the Sample A has more total cells per notebook (Median = 95) than Sample B (Median = 18). Sample A has roughly equal ratio of markdown cells and code cells per notebook, while Sample B is unbalanced with majority cells being code cells. Notably, Sample A has more total words in markdown cells (Median = 1728) than Sample B (Median = 218). This result indicates that the 80 well-documented computational notebooks are better documented than general Github notebooks.

Table 5.1: We identified 9 categories based on the purpose of markdown cells. Note that a markdown cell may belong to multiple categories of contents or none of the categories.

Category	N	Description	Example
Process	2115 (58.65%)	The markdown cell describes what the following code cell is doing. This always appears before the relevant code cell.	Transforming Feature X to a new binary variable
Headline	1167 (32.36%)	The markdown cell contains a headline in markdown syntax. The cell is used for navigation purposes or marking the structure of the notebook. It may be relevant to a nearby code cell.	# Blending Models
Result	692 (19.19%)	The markdown cell explains the output. This type always appears after the relevant code cell.	It turns out there is a long tail of outlying properties...
Background Knowledge	414 (11.48%)	The markdown cell provides a rich content for background knowledge, but may not be relevant to a specific code cell.	Multicollinearity increases the standard errors of the coefficients.
Reason	227 (6.30%)	The markdown cell explains the reasons why certain functions are used or why a task is performed. This may appear before or after the relevant code cell.	We do this manually, because ML models won't be able to reliably tell the differences.
Todo	202 (5.60%)	The markdown cell describes a list of actions for upcoming analysis. This normally is not relevant to a specific code cell.	1. Apply models 2. Get cross validation scores 3. Calculate the mean
Reference	200 (5.55%)	The markdown cell contains an external reference. This is also relevant to the adjacent code cell.	Gradient Boosting Regression Refer [here] (https://...)
Meta-Information	141 (3.91%)	The markdown cell contains meta-information such as project overview, author's information, and a link to the data sources. This often is not relevant to a specific code.	The purpose of this notebook is to build a model with Tensorflow.
Summary	51 (1.41%)	The markdown cell summarizes what has been done so far for a section or a series of steps. This often is not relevant to a specific code.	**In summary** By EDA we found a strong impact of features like Age, Embarked..

5.2.3.2 Data scientists use markdown cells to document a broad range of topics.

As shown in Table 5.1, our analysis revealed that markdown cells are mostly used to describe what the adjacent code cell is doing (Process, 58.65%). Second to the Process

category, 32.36% markdown cells are used to specify a headline for organizing the notebook into separate functional sections and for navigation purposes (Headline).

Markdown cells can also be used to explain beyond the adjacent code cells. We found that many markdown cells are created to describe the outputs from code execution (Result, 19.19%), to explain results or critical decisions (Reason, 6.30%), or to provide an outline for the readers to know what they are going to do in a list of todo actions (Todo, 5.60%), and/or to recap what has been done so far (Summary, 1.41%).

We observed that 11.48% markdown cells explain what a general data science concept means, or how a function works (Background Knowledge), while 5.54% markdown cells are connected with external references for readers to further explore the topics (Reference). We believe these are the extra efforts that the notebook owners dedicated, to attract a broader audience, especially beginners in the Kaggle community. In addition, we found that authors approached the story in different styles. For example, some authors want to leave their own signature, and so they spend spaces at the beginning of the notebooks to debrief the project, to add the author's information, or even to add their mottos (Meta-Information, 3.91%). Some authors prefer to use concise and accurate language to convey important information; while others write documentation in more creative and entertaining ways — for example, making analogies between data science workflow and starting a business.

5.2.3.3 Data science stages.

We coded markdown cells based on where they belong in the data science workflow [190]. As shown in Table 5.2, we identified four stages and 13 tasks. The four stages include **environment configuration** (4.50%), **data preparation** and exploration (37.05%), **feature engineering and selection** (10.40%), and **model building and selection** (27.57%). At the finer-grained task level, in particular, notebook authors create more markdown cells for documenting exploratory data analysis tasks (26.62%) and model training tasks (10.45%). The rest of the markdown cells are evenly distributed along with other tasks.

5.2.4 Design Implications

In summary, our analysis of markdown cells in well-documented notebooks suggests that data scientists document various types of content in a notebook, and the distribution of these markdown cells generally follows an order of the data science lifecycle, starting with

Table 5.2: We coded each markdown cell to which data science stage (or task) they belong. We identified 4 stages with 13 tasks out of the data science lifecycle [187]. Note that a markdown cell may belong to multiple stages or none of the stages.

Stage	Total	Task	N
Environment Configuration	162 (4.49%)	Library Loading	33 (0.92%)
		Data Loading	129 (3.58%)
Data Preparation and Exploration	1336 (37.05%)	Data Preparation	91 (2.52%)
		Exploratory Data Analysis	960 (26.62%)
		Data Cleaning	285 (7.90%)
Feature Engineering and Selection	375 (10.40%)	Feature Engineering	120 (3.32%)
		Feature Transformation	178 (4.94%)
		Feature Selection	77 (2.14%)
Model Building and Selection	994 (27.57%)	Model Building	247 (6.85%)
		Data Sub-Sampling and Train-Test Splitting	61 (1.69%)
		Model Training	377 (10.45%)
		Model Parameter Tuning	81 (2.25%)
		Model Validation and Assembling	288 (6.32%)

data cleaning, and ending with model building and selection. Based on these findings, we synthesize the following actionable design considerations:

- **The system should support more than one type of documentation generation.** Data scientists benefit from documenting not only the behavior of the code, but also interpreting the output, and explaining rationales. Thus, a good system should be flexible to support more than one type of documentation generation.
- **Some types of documentations are highly related to the adjacent code cell.** We found at least the Process, Result, Reason, and Reference types of documentations are highly related to the adjacent code cell. To automatically generate interpretations of results or rationale for a decision may be hard, as both involve deep human expertise. But, with the latest neural network algorithms, we believe we can build an automation system to generate Process type of documentation, and we can also retrieve Reference for a given code cell.

- **There are certain types of documentations that are irrelevant to the code.** Various types of documentations do not have a relevant code piece upon which the automation algorithm can be trained. Together with the Reason and Result types, the system should also provide a function that the human user can easily switch to the manual creation mode for these types.
- **For different types of documentation, it could be at the top or the bottom of the related code cell.** This design insight is particularly important to the Process, Result, and Reason types of documentation. It may be less preferable to put Result documentation before the code cell, where the result is yet to be rendered. The system should be flexible to render documentation at different relative locations to the code cell.
- **External resources such as URLs and the official API descriptions may also be useful.** Some types of documentation, such as Background Knowledge and Reference, are not easy to be generated with the NN-based models, but they are easy to retrieve from the Internet. So the system should incorporate the capability to fetch relevant web content as candidate documentation.
- **There is an ordinality in markdown cells that is aligned with the data science project’s lifecycle.** The system should consider that Library Loading types of cells are often at the beginning section of the notebook, and the Model Training type of content may be more likely to appear near the end of the notebook. In our system prototype, though, we did not take this design consideration into account, it will be our future work.
- **The notebook would be nice to have documentation with a problem overview at the beginning and a summary at the end.** We considered this design implication not in the system design, but our evaluation study design. For the two barebone notebooks we used in the experiment, we always provide a problem overview as a markdown cell at the top of the notebook.

5.3 Design and Implementation

Based on findings from the formative study and design insights from related works, we design and implement Themisto, an automatic documentation generation system that sup-

ports data scientists to write better-documented computational narratives. In this section, we present the system architecture, the user interface design, and the core technical capability of generating documentation.

5.3.1 System Architecture

The Themisto system has two components: the client-side UI is implemented as a Jupyter Notebook plugin using TypeScript code, and the server-side backend is implemented as a server using Python and Flask.

The client-side program is responsible to render user interface, and also to monitor the user actions on the notebook to edits in code cells. When the user's cursor is focused on a code cell, the UI will send the current code cell content to the server-side program through Hypertext Transfer Protocol (HTTP) requests.

The server-side program takes the code content and generates documentation using both the deep-learning-based approach and the query-based approach. For the deep-learning-based approach, the server-side program first tokenizes the code content and generates the AST. It then generates the prediction with the pre-trained model. For the query-based approach, the server-side program matches the curated API calls with the code snippets and returns the pre-collected descriptions. For the prompt-based approach, the server-side program sends different prompts (e.g., for interpreting result or for explaining reason) base on the output type of the code cell.

5.3.2 User Interface Design

Figure 5.2 shows the user interface of Themisto as a Jupyter Notebook plugin. Each time the user changes their focus on a code cell, as they may be inspecting or working on the cell, the plugin is triggered. The plugin sends the user-focused code cell's content to the backend. Using this content, the backend generates a code summarization using the model and retrieves a piece of documentation from the API webpage. When such a documentation generation process is done, the generated documentation is sent from the server-side to the frontend, and a light bulb icon appears next to the code cell, indicating that the there are recommended markdown cells for the selected code cell (as shown in Figure 5.2.A).

When a user clicks on the light bulb icon which appears next to any selected code cells, Themisto render all the three options in the dropdown menu: (1) a deep-learning-based

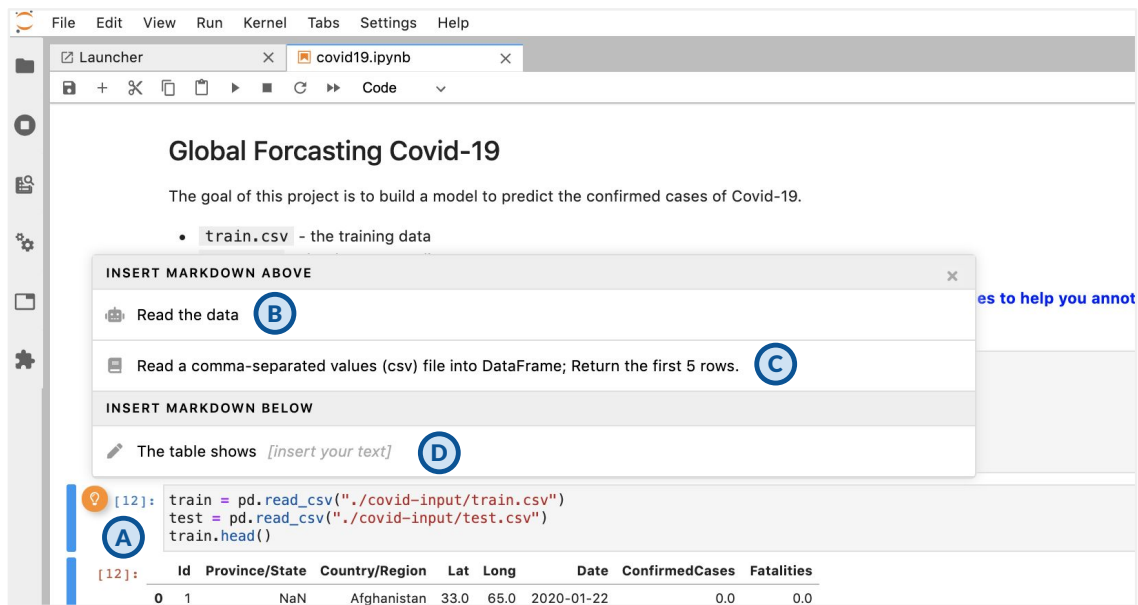


Figure 5.2: The Themisto user interface is implemented as a Jupyter Notebook plugin: (A) When the recommended documentation is ready, a lightbulb icon shows up to the left of the currently-focused code cell. (B – D) shows the three options in the dropdown menu generated by Themisto, (B) A documentation candidate generated for the code with a deep-learning model, (C) A documentation candidate retrieved from the online API documentation for the source code, and (D) A prompt message that nudges users to write documentation on a given topic.

approach to generate documentation for source code (Figure 5.2.B); (2) a query-based approach to retrieve the online API documentation for source code (Figure 5.2.C); and (3) a user prompt approach to nudge users to write more documentation (Figure 5.2.D). If the user likes one of these three candidates, they can simply click on one of them, and the selected documentation candidate will be inserted into above the code cell (if it is the Process, Reference, or Reason type), or below it (if it is the Result type).

5.3.3 Three Approaches for Documentation Generation

In this subsection, we describe the rationale and implementation detail of the three different approaches for documentation generation (Figure 5.3):

- Our formative study suggests that the system should be able to **generate multiple types of documentation** (e.g., Process, Result, Background Knowledge, Reason, and Reference).

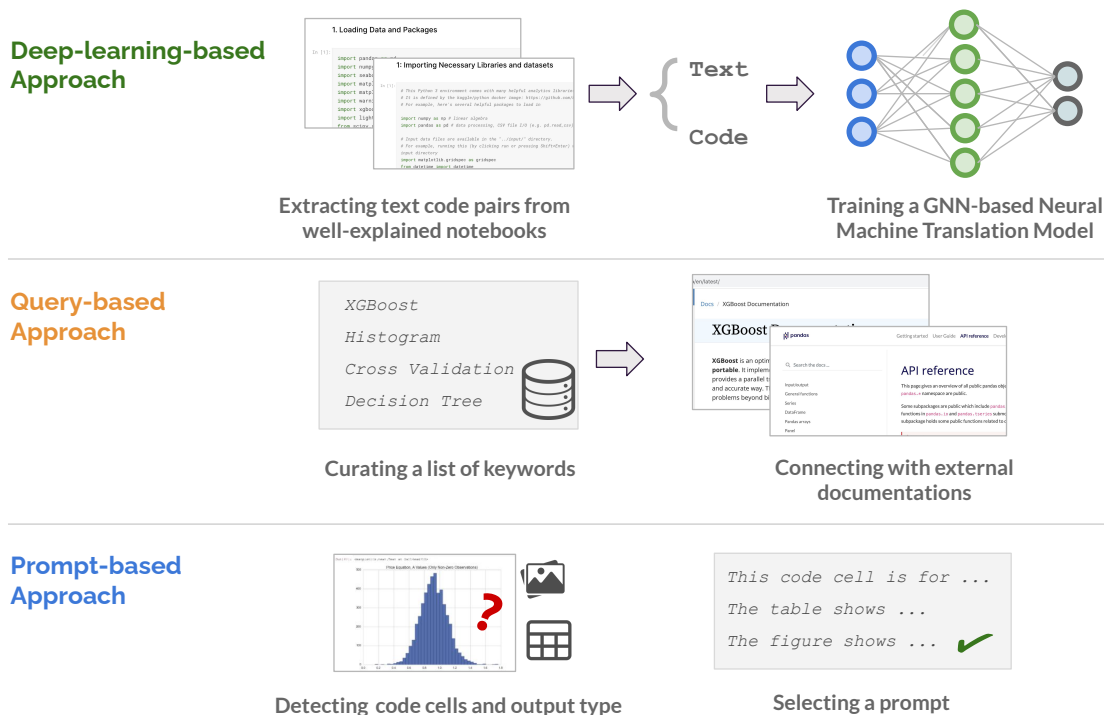


Figure 5.3: An illustration of the three different approaches for documentation generation in Themisto.

- Some types of documentation can be **directly derived from the code**, thus the automated approaches can help. The Process type of documentation directly describes the coding process, and existing ML literature suggests that the deep-learning-based approach is most suitable for generating it; The Reference type does not need a learning-based approach, it can be achieved with a traditional query-based approach, which locates and retrieves the most relevant online documentation as candidates;
- Some others types of documentation (e.g., Education, Result, and Reason) are **not directly related to the code**, thus the fully automated approaches are not capable of generating such contents. We design the prompt-based approach for users to complete the generation process.

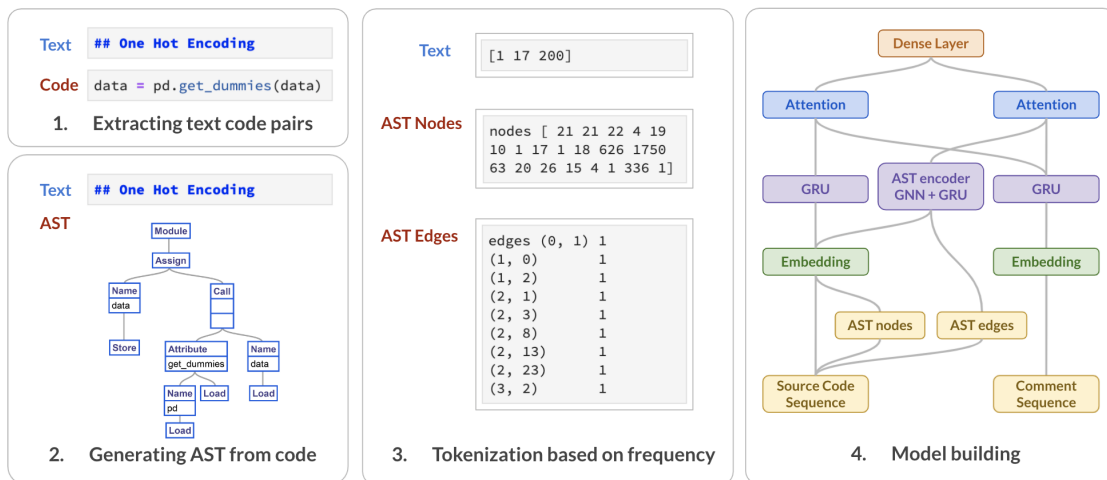


Figure 5.4: A code summarization model for the deep-learning-based documentation generation approach via GNN. There are three steps of data pre-processing (1) We first extract text code pairs from existing notebooks. (2) We generate AST from code. (3) We tokenized each word and translated them into embeddings. And (4), the GNN model architecture.

5.3.3.1 Deep-Learning-Based Approach

We trained a deep learning model³ using the Graph-Neural-Network architecture based on LeClair et al. [108]. These GNN models can take both the source code’s structure (extracted as AST) and the source code’s content as input. Thus, it outperforms the traditional sequence-to-sequence model architectures, which only takes the source code’s content as an input sequence, in source code summarization tasks for Python code⁴. We did not consider T5, BerT or GPT-3 architectures as these models can take minutes to make one inference (i.e., generate one summary) even with a cluster of GPUs (costing thousands of dollars per hour), whereas our GNN-based model can make an inference within a second with one GPU.

In order to fine-tune the model, we constructed a training dataset for our particular context. We collected the top 10% highly-voted notebooks from two popular Kaggle competitions – House Price Prediction and Titanic Survival Prediction (N = 1158). For each of the notebook, we first extracted code cells and the markdown cells adjacent above as a pair of input and output (similar to the data collection approach in [19]). If there is an inline comment in the first line of the code cell, we replaced the output of the pair using

³We release a larger dataset and a refined version of the model in a separate paper [109].

⁴All the collected data science notebooks are in Python.

the inline comment. In total, our dataset has 5,912 pairs of code and its corresponding documentation. Following the best practice of model training, we split the dataset into training, testing, and validation subsets with an 8 to 1 to 1 ratio.

Before feeding data into the training process, we have a three-step pre-processing stage, as illustrated in Figure 5.4. Step 1 removes the style decoration, formats, and special characters that are not in Python grammar (e.g., Notebook Magics). We also generate an AST for the source code input in step 2 with Python AST library⁵. The AST result is equivalent to the source code but with more contextual and relational information. In step 3, we tokenize source code to a sequence of tokens with an input dictionary, and parse the AST nodes as a sequence of tokens with the same input dictionary. We parse the relationship between AST nodes as a matrix of edges. Finally, we tokenize the output documentation as a sequence of tokens with a separate output dictionary. After this process, all the tokens are transformed into an array of word embeddings — vectors of real numbers. We use these data to train the network for 100 epochs, 30 batch sizes, and 15 early stop points on a two Tesla V100 GPU cluster. Out of all the epochs, we selected the model with the highest validation accuracy score.

To evaluate our model’s performance against baseline models, we conducted both quantitative and qualitative evaluations, as suggested by [151]. For the automated quantitative evaluation, we use BLEU scores [130] as the model performance metric. BLEU scores are commonly used in the source code summarization tasks. It evaluates the word similarity between the generated text and the ground truth text. We selected and trained Code2Seq model [20] and Graph2Seq model [197] with the same data split.

Our model achieves 11.41 (BLEU–a), which outperforms the baseline models Code2Seq (BLEU–a = 9.61) and Graph2Seq (BLEU–a = 11.05). These scores suggest that the data science documentation task is more difficult than the benchmark code summarization tasks in the software engineering field. For example, in data science, a notebook code cell can contain multiple code snippets and functions.

In addition to the automated quantitative evaluation, we also conduct a qualitative analysis of the generated documentation pieces. We found that despite the word-to-word similarity score is low, the general quality of the content is reasonable and satisfying for building a prototype system. As an illustration, we provide three examples with both input and model generated outputs, as shown in Table 5.3. In the Appendix, we provide full code cells and model-generated outputs for the two experimental notebooks that we

⁵<https://docs.python.org/3/library/ast.html>

Table 5.3: Example output from the model. (Example A) The generated text well describes the code. (Example B) The generated text vaguely describes the code. (Example C) The generated text is poorly readable, but still captures the keywords of the descriptions.

Example	Code Cell	Output from the Model
Example A	<code>train = pd.read_csv('./house-input/train.csv')</code> <code>test = pd.read_csv('./house-input/test.csv')</code>	Read the data
Example B	<code>all_data = pd.get_dummies(all_data)</code>	Convert all the data
Example C	<code>pred = Tree_model.predict(x_test)</code> <code>pred = pd.DataFrame(pred)</code> <code>pred.columns = ["ConfirmedCases_prediction"]</code>	Predicate to use a predict function for tests

used in the user study.

5.3.3.2 Query-Based Approach

Our formative study indicates that the well-documented Kaggle notebooks often have the description of frequently-used data science code functions for educational purposes. And sometimes data scientists directly paste in a link or a reference to the external API documentation for a code function. Thus, we implement a query-based approach that curates a list of APIs from commonly used data science packages, and the short descriptions from external documentation sites. In our system, we only cover Pandas⁶, Numpy⁷, and Scikit-learn⁸ these three libraries as a starting point to explore this approach. We argue that it can be easily expanded to include other packages in the future. We collected both the API names and the short descriptions by building a crawling script with Python. When users trigger this query-based approach for a code cell, Themisto matches the API names with the code snippets and concatenate all the corresponding descriptions.

5.3.3.3 Prompt-Based Approach

Lastly, the system also provides a prompt-based approach that allows users to manually create the documentation. Because our formative study found that a well-documented notebook not only documents the process of the code, but also interprets the output, and explains rationales. These types of documentation are hard to generate with automated

⁶<https://pandas.pydata.org/docs/reference/index.html>

⁷<https://numpy.org/doc/stable/reference/>

⁸<https://scikit-learn.org/stable/modules/classes.html>

solutions To achieve it, we implement a prompt-based approach. It detects whether the code cell has a cell output or not: if the cell outputs a result, Themisto assumes that the user is more likely to add interpretation for the output result, thus the corresponding prompt will be inserted below the code cell. Otherwise, the system assumes the user may want to insert a reason or some educational types of documentations, thus it changes its prompt message.

5.4 User Evaluation of Themisto

To evaluate the usability of Themisto and its effectiveness in supporting data scientists to create documentation in notebooks, we conducted a within-subject controlled experiment with 24 data scientists. The task is to add documentation to the given notebook. And each participant is asked to finish two sessions, one with the Themisto support and one without its support. The evaluation aims to understand (1) how well Themisto can facilitate documenting notebooks and (2) how data scientists perceive the three approaches that are used by Themisto for generating documentation.

5.4.1 Participants

We recruited 24 data science professionals as our evaluation participants in a multinational IT company. We used a snowball sampling approach to recruit participants, where we sent recruitment messages to friends and colleagues, various internal mailing-lists, and Slack channels. We then asked participants to refer their friends and colleagues. Our recruitment criteria are that the participant must have had experience in data science projects and they are familiar with Python and Jupyter Notebook environment. As shown in Table 9.1, participants reported a diverse job role backgrounds, including expert data scientists (N = 8), novice data scientists (N = 9), AI Operators (AIOPs) or Machine Learning (ML) engineers (N = 2), subject matter experts (N = 1), and application developer (N = 4).

5.4.2 Study Protocol

We conducted a within-subject controlled experiment with 24 data scientist participants. Their task was to add documentation to a given draft notebook, which only has code and no documentation at all. The participants were told that they were adding documentation for the purpose of sharing those documented notebooks as tutorials for data science

Table 5.4: Demographics of participants

PID	Gender	Job Role	Work Experience in Data Science
P1	M	Expert Data Scientist	5-10 years
P2	M	Application Developer	3-5 years
P3	M	Novice Data Scientist	less than 3 years
P4	M	Novice Data Scientist	0 year (just start learning data science)
P5	M	AI Operator or ML Engineer	3-5 years
P6	M	Novice Data Scientist	less than 3 years
P7	M	Application Developer	3-5 years
P8	M	Novice Data Scientist	less than 3 years
P9	F	Expert Data Scientist	3-5 years
P10	M	Expert Data Scientist	5-10 years
P11	M	Expert Data Scientist	more than 10 years
P12	F	Novice Data Scientist	3-5 years
P13	F	Expert Data Scientist	5-10 years
P14	M	Novice Data Scientist	0 year (just start learning data science)
P15	M	Expert Data Scientist	more than 10 years
P16	M	AI Operator or ML Engineer	3-5 years
P17	M	Subject Matter Expert	3-5 years
P18	M	Expert Data Scientist	more than 10 years
P19	F	Application Developer	3-5 years
P20	F	Expert Data Scientist	3-5 years
P21	M	Novice Data Scientist	3-5 years
P22	M	Novice Data Scientist	less than 3 years
P23	M	Application Developer	less than 3 years
P24	M	Novice Data Scientist	less than 3 years

students who just started learning data science. Each participant is asked to finish two sessions, one with the Themisto support (Experiment condition) and one without its support (Control Condition). We prepared two draft notebooks, one for each session, shown in the Appendix. The two experiment notebooks are adapted from winning notebooks from two Kaggle challenges, which are not included in the model training dataset: 1) House

Price Prediction⁹; 2) COVID Case Prediction¹⁰. The two notebooks have the same length (9 code cells) and a similar level of difficulty. Although the two notebooks are simplified versions from winning notebooks, they cover most stages in data science lifecycles. In addition, the length of the notebooks falls into the middle range of the notebook length distribution on the GitHub corpus (as referred to Figure 5.1). To counterbalance the order effect, we randomized the order of the control condition and the experiment condition for each participant, so some participants encountered Themisto in their first session, and some others experienced it in their second session.

Each participant was given up to 12 minutes (720 seconds) to finish one session. We conducted three pilot run sessions, and all the three pilot participants were able to finish a single task within 10 minutes, with or without the support of Themisto. Before the experiment condition session, we gave the participant a 1-minute quick demo on the functionality of Themisto. All study sessions were conducted remotely via a teleconferencing tool. We asked the participants to share their screen and we video recorded the entire session with their permission. After finishing both sessions, we conducted a post-experiment semi-structured interview session to ask about their experience and feedback. We had a few pre-defined questions such as “How do you compare the experience of the documenting task with or without the support of Themisto?” or “Did you notice the multiple candidates in the dropdown menu? Which one do you like the most?” In addition, participants were encouraged to tell their stories and experience outside these structured questions. The interview sections of the video recordings were transcribed into text.

5.4.3 Data Collection and Measurements

We have three data sources: the observational notes and video recording for each session (N = 48), the final notebook artifact out of each session (N = 48), and the post-task questionnaire and interview transcripts (N = 24).

Our first group of measurements are from coding participants’ behavioral data from the session recordings. In particular, we counted *the task completion time* (in secs) for all sessions. Then, for experiment condition only, we also counted the followings: how many times a participant clicked on the light bulb icon to check for suggestions (*code cells checked for suggestions*); how many times a participant directly used the generated documentation (*markdown cells created by Themisto*); how many times a participant

⁹<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>

¹⁰<https://www.kaggle.com/c/covid19-global-forecasting-week-1>

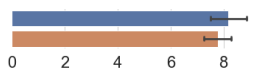
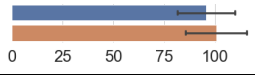
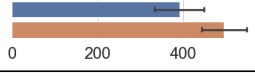



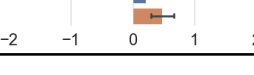
ignored the generated recommendations and manually created the documentation (*markdown cells created by human*); and how many time a markdown cell is co-created by human and Themisto(*markdown cells co-created by human and Themisto*). The result of this analysis is reported in Table 5.5 and 5.6.

Secondly, to evaluate the quality of the final notebook artifact, we define our second group of measurements by counting: the *number of added markdown cells*, and the *number of added words* as these two are indicators of the quantity and effort each participant spent in a notebook. Also, we asked the participants to give a self-reported satisfaction score to each of the two documented notebooks. We considered that score (-2 to 2) as a self-reported subject feeling of the notebook satisfaction. In parallel, we asked two experts to rate the notebook-level quality (N = 48) with a 3-dimensional rubric (based on [59]) to evaluate the documentation's *readability, accuracy, and informativeness* in a notebook. We considered these three scores (-2 to 2) as an objective quality of the notebook. Readability concerns whether the documentation is in readable English grammar and words, while accuracy concerns how the documentation matches the code content, and informativeness evaluates whether the documentation covers more information units. Two experts iteratively discussed and evaluated the notebooks until the independent ratings achieved an agreement ($\alpha = 0.76$, Krippendorff's alpha). The result of this analysis is reported in Table 5.5.

For the experiment session only, we conducted a cell-level expert rating (N = 194) using the same approach as in notebook-level expert rating. Two experts iteratively discussed and evaluated the notebooks until the independent ratings achieved an agreement ($\alpha = 0.88$, Krippendorff's alpha). The result of this analysis is reported in Table 5.6. In addition, we asked the participants to finish a post-experiment survey (5-point Likert Scale, -2 as strongly disagree and 2 as strongly agree, Figure 9.3) to collect their feedback specific on the system's *usability, accuracy, trust, satisfaction, and adoption propensity* (based on [193]).

Lastly, for the interview transcripts, four researchers of this research project conduct an iterative open coding method to get the code, theme, and representative quotes as the third group of data. They each independently coded a subset of interview transcripts, and discussed the codes and themes together. After the discussion, they when back and reiterated the coding practice to apply the codes and themes to their assigned notebooks. Some examples of the identified themes are: pros and cons of Themisto; preference of the three document generation approaches; future adoption, and suggestions for design

Table 5.5: Performance data in two conditions (M: mean, SD: standard deviation): the task completion time (secs), participants’ satisfaction with the final notebook (from -2 to 2), graded notebook quality, number of markdown cells, and number of words. In particular, participants spent less time to complete the task in the experimental condition than the control condition ($p = .001$); participants were more satisfied with the final notebook in the experimental condition than the control condition ($p = .04$).

	Condition	M	SD	
Number of Added Markdown Cells	Experiment	8.04	2.40	
	Control	7.79	1.91	
Number of Added Words	Experiment	95.75	50.56	
	Control	100.92	53.27	
**Task Completion Time (secs)	Experiment	391.12	200.15	
	Control	494.75	184.28	
*Satisfaction with the Final Notebook	Experiment	0.96	0.69	
	Control	0.54	0.83	
Expert Rating: Accuracy (-2 to 2)	Experiment	1.60	0.47	
	Control	1.62	0.52	
Expert Rating: Readability (-2 to 2)	Experiment	0.65	0.83	
	Control	0.90	0.57	
Expert Rating: Informativeness (-2 to 2)	Experiment	0.67	0.64	
	Control	0.75	0.63	

improvement. We will report the qualitative results as supporting materials together with reporting the quantitative results.

5.4.4 Results

In this section, we present the user study results on: how Themisto improved participants’ performance on the task, how participants perceived the documentation generation methods in Themisto, and how participants described the practical applicability of Themisto.

5.4.4.1 Themisto supports participants to easily add documentations to a notebook.

Our experiment revealed that Themisto improved participants’ performance on the task by reducing task completion time and improving the satisfaction with the final notebooks.

We performed a two-way repeated measures ANOVA to examine the effect of the two notebooks and the two conditions (with or without Themisto) on task completion time. As shown in Table 5.5, participants spent significantly **less time** ($p < .001$) to complete the

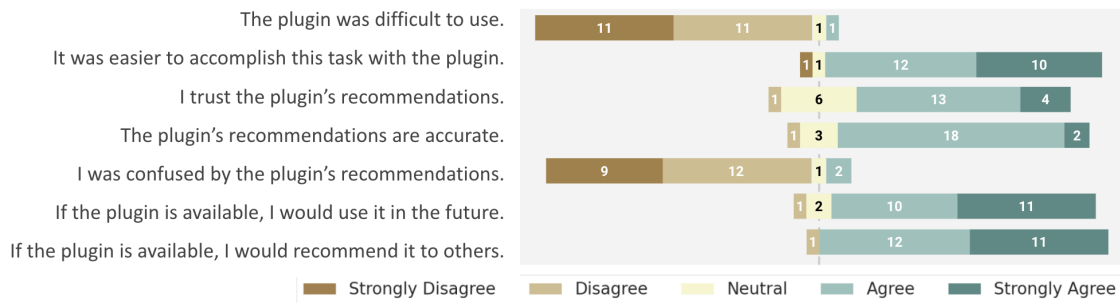


Figure 5.5: Results of the post-task questionnaire. Note that the disagrees are not from the same participant.

task using Themisto in the experiment condition ($M(SD) = 391.12 (200.15)$) than in the control condition ($M(SD) = 494.75 (184.28)$). In addition, there was not a statistically significant effect of notebooks on task completion time, nor a statistically significant interaction between the effects of notebooks and conditions on completion time.

The post-experiment survey result supported our findings. Most participants believed it was easier to accomplish the task with Themisto's help (22 out of 24 rated agree or higher), as shown in Figure 9.3. And the Themisto generated recommendation was accurate (20 out of 24 rated agree or higher).

Looking into the qualitative interview data, we can find some potential explanations for why participants believed so. Participants reported that Themisto provided them something to begin with, thus it was easier than starting from scratch: *“The plugin makes it easy to just pick it and have something simple. And then I got a couple of times where I went back and said, ‘Oh let me add a few more words.’”* (P21).

5.4.4.2 Co-creation yields longer documentation and improves accuracy and readability.

Through coding the video recordings for only the experiment-condition sessions, we were able to examine the following questions: while the Themisto was available, how did the participants use it? Did they check the recommendations it generated? Did they actually use those recommendations in their documentations added into notebooks?

As shown in Table 5.6, we found that while Themisto is available, for 86.11% of code cells, participants checked the recommended documentation by clicking on the light bulb icon to show the dropdown menu. Then, 46.90% of the created markdown cells were directly adopted from Themisto's recommendation; while 11.86% of the created

markdown cells were manually crafted by humans alone. **The most interesting finding is that 41.24% markdown cells were co-created by Themisto and human participants together:** Themisto suggests a markdown cell, human participants take it, and modify on top of it. This result suggested that most participants used Themisto in the creation of documentation, and some of them formed a small collaboration between humans and the Artificial Intelligence (AI). This finding inspires us to further explore how participants co-create the documentation with Themisto [119]. By looking at the log data, we discover several editing patterns. For example, many participants added supplemental details (e.g., expanding the steps into substeps) to Themisto’s suggested documentation. Participants also added stylistic edits, including modifying document hierarchies, polishing sentences, and changing conversational tones.

In order to explore the differences among documentation created by three methods (created by Themisto only, co-created by human and Themisto, created by human only), we conducted a cell-level expert rating ($N = 194$) along the dimension of accuracy, readability, and informativeness. We also calculated the word count of the documentation length. We performed a one-way ANOVA to examine the differences among the three groups. As shown in Table 5.6, markdown cells that are co-created by humans and Themisto have significantly more word count ($M(SD) = 15.45 (10.97)$) than markdown cells that are manually written by humans alone ($M(SD) = 10.26 (7.41)$) and the markdown cells that are directly adopted from Themisto’s recommendation ($M(SD) = 8.88 (7.14)$), with $F = 11.83, p < 0.001$. Markdown cells co-created by humans and Themisto also yield better results in terms of accuracy ($F = 9.43, p < 0.001$) and readability ($F = 3.28, p = 0.04$), while for informativeness, there is no significant differences across three groups. Our posthoc analysis suggested that no significant differences were found between markdown cells created by Themisto and markdown cells created by humans only along all dimensions (including word count, accuracy, readability, and informativeness).

5.4.4.3 Themisto increases participant’s satisfaction , while maintaining a similar quality of the final notebook.

The post-task questionnaire revealed that participants were **more satisfied** with the final notebook after using Themisto in the experiment condition than in the control condition ($p = .04$) (Table 5.5). The interview results also supported this finding. P14 believed that Themisto helped with wording: *“Sometimes I knew what the cells were doing but I did not know how to put things in a really good sentence for others.”*

Table 5.6: Usage data of the plugin in experimental condition. The results indicate that participants used the plugin for recommended documentation on most code cells (86.11%). For markdown cells in the final notebooks, 46.90% were directly adopted from the plugin’s recommendation, while 41.24% were modified from the plugin’s recommendation and 11.86% were created by participants from scratch.

	<i>N</i>	<i>%</i>	Word Count	Accuracy	Readability	Informativeness
Code Cells Checked for Suggestions	186	86.11	-	-	-	-
Markdown Cells Created by Themisto Only	91	46.90	8.88 (7.14)	1.36 (0.54)	1.94 (0.23)	1.34 (0.56)
Markdown Cells Co-created by Humans and Themisto	80	41.24	15.45 (10.97)	1.68 (0.47)	1.96 (0.17)	1.51 (0.55)
Markdown Cells Created by Humans Only	23	11.86	10.26 (7.41)	1.28 (0.69)	1.83 (0.36)	1.35 (0.65)

Themisto also motivates participants to document the analysis details. Although we did not see a difference in the number of markdown cells created in two conditions or the number of words in total, Themisto helps them overcome the procrastination of writing documentation and reminds them to document things that they might ignore.

I think I definitely overlooked some details when I was commenting without the tool, because I just made the assumption that people should know from the code... To be honest, I do not usually follow a good coding practice. My notebooks are really messy and I am the only person who can understand it. I feel sorry for anybody else that has to see it. (P19)

Moreover, participants believed that Themisto can help them form a better documenting practice in the long term: “*It very useful to remind me to always put some documentation in a timely manner.*” (P13).

The two experts’ gradings for the notebook quality suggest that there was not a significant difference for the three dimensions of the quality rubric (**accuracy, readability, and completeness**). In the post-task interview, participants mentioned that the accuracy of the generated recommendations plays a role in participant’s experience: “*My experience with the plugin is definitely better. For the most part, the suggestions are pretty accurate. Although sometimes I did make a few minor changes like rearranging the text.*” (P5). Some participants also mentioned that they needed to edit the format of the generated document to fit their context. We believe that while Themisto offers convenience to improve the data

scientists' productivity and saves their time, it may not provide the same level of readability as those notebooks well articulated by humans. Thus, data scientists may want to further revise the formatting and wording of the Themisto generated documentation.

In summary, our experiment indicated that Themisto improves participants' productivity for creating documentation. It also increases their perceived satisfaction with the final notebook, compared to the notebooks written by participants themselves.

5.4.4.4 The three approaches of generating documentation are suitable for different scenarios.

In this section, we have an in-depth analysis of how participants perceived the three different approaches that Themisto implemented to generate documentations: the deep-learning-based approach, the query-based approach, and the prompt-based approach. In the post-experiment interview, we explained how Themisto generated the documentation with these three approaches, and asked participants if they like or dislike one particular approach.

Participants reported that they felt the deep-learning-based approach provided concise and general descriptions of the analysis process: *"I think the AI suggestion gives me an overview. It is short, and has some useful keywords."* (P12).

Participants also suggested that the deep-learning-based approach sometimes generated inaccurate or very vague documentation: *"The first one gives me a very short summary, though it didn't always say what the cell is doing."* (P1). But the deep-learning-based approach is still perceived useful. As it is short and with only a couple of keywords, many participants believe it may be more suitable for some quick and simple documentation task, or for the analyst audience who can understand these short keywords.

In terms of the query-based approach, participants believed that the documentation generated from this approach contains has longer and more descriptive information. This approach is further perceived to be more suitable for educational purposes: *"This one gives you really good information. For some specific methods or calls, you don't have to come up with a high-level summary for others and you can directly use it."* (P14).

Participants also acknowledged that such a query-based approach may not work for some scenarios. For example, participants found that the query-based approach was not useful for summarizing the very fundamental level data manipulations, as there was no core API method in it. Some participants mentioned that the usefulness of this query-based approach depends on the audience.

The [deep-learning-based approach] was really useful. The [query-based approach]... it depends on the audience. It is much more appropriate for a novice programmer. (P18)

We observed in the video recordings that participants rarely used the prompt-based approach in the session. The interview data confirmed our speculation. Some participants said that they liked the idea of user prompts, but they did not use it because the deep-learning-based approach and the query-based approach already gave them the actual content. Other participants pointed out that the prompts were not intelligent enough, so they did not use it: *“It always asks the same thing and I just ignored the prompts.”* (P18).

Participants suggested that the prompts could be designed to better fit the context.

Perhaps the system can infer what the code cell was doing [from the deep-learning-based approach], and show prompts accordingly. Like if I delete a data point from the dataset, there is a prompt asking why I considered it as an outlier or something. (P5)

Last but not the least, many participants preferred a hybrid approach to combine the deep-learning-based approach and the query-based approach. For example, P12 mentioned,

The first one (deep-learning-based) tells me what the code cell is doing in general and the second one (query-based) tells me the details of the function. I would go with a hybrid approach. (P12)

5.4.4.5 Will participants use Themisto in their future data science project?

Most participants indicated that they would like to use Themisto in the future when answering the survey question as shown in Figure 9.3. The interview data provides more detail and evidence to elaborate on this result. Participants suggested various scenarios in which the Themisto could be useful in their future work, such as they need to add documentation during the exploration process for future selves, or they need to document a notebook in a post-hoc way for sharing it with collaborators, or they need to mentor a team member who is a novice data scientist, or they need to refactor an ill-documented notebook written by others:

When I am doing data analysis, I tend to write the code first because there is a flow in my head of what I need to do. And then I will go back afterward to use the plugin and add the comments needed. I will definitely do that before sharing that file or handing it over to others. (P12)

There was one participant who did not think Themisto could fit into his workflow: *“I always write documentation before writing code. Maybe Themisto does not work for people like me.”* (P3)

In our experiment, we provided the scenario as they were documenting the notebook as a tutorial for some data science students. In the interview, we asked how participants would document a notebook differently if they were created documentation for the notebooks for non-technical domain experts audience. Some participants suggested that computational notebooks may not be a good medium to present the analysis to non-technical domain experts. They would prefer to curate all the textual annotations in a standalone report or slide decks. Some others believed that the notebook could work as the medium but they would change the documentation by using less technical terminology, adding more details on topics that the non-technical domain experts would be interested in (e.g., how data is collected, potential bias in the analysis).

5.4.4.6 Participants suggest various design implications for automated code documentation.

In the interview, participants provided various design suggestions to improve Themisto and to design future technologies that can support data scientists to document the notebook.

Participants expected Themisto to have more functionalities than simply generating documentation for the code. For example, P13 proposed maybe Themisto can also create a description to document the changes of versions and the editing histories from different team members. P3 and P4 believed that the automatic generation of Reason is very much needed for explaining decisions such as why selecting a particular algorithm. P19 wanted the system to automatically add explanations to the execution errors. Participants also mentioned that Themisto should add more varieties into the generated content’s formatting. They would like to see suggested documentation with a better presentation.

And lastly, some participants suggested that maybe such a documentation generation system can take consideration of the purpose of the notebook, the domain-specific terminology, or the individuals’ habits for writing documentation.

5.4.4.7 Summary of the Results

In summary, our study found that Themisto can support data scientists in generating documentation by significantly reducing their time spent on the task, and improving the perceived satisfaction level of the final notebook. When Themisto is available, participants are very likely to check the generated documentation as a reference. Many of them directly used the generated documentation, a few of them still prefer to manually type the documentation, while many of them adopted a human-AI co-creation approach that they used the AI-generated one as a baseline and keep improving on top of it. Participants perceived the documentation generated by the deep-learning-based approach as a short and concise overview, the documentation generated by the query-based approach as descriptive and useful for educational purposes, but they rarely used the prompt-based approach. Overall, participants enjoyed Themisto and would like to use it in the future for various documenting purposes.

5.5 Discussion

5.5.1 The Documentation Practices in Data Science is Different from in Software Engineering

The practice of documentation in data science has both overlaps and strong contrasts in relation to the ones in software engineering in many facets. Software engineers write inline comments in their work-in-progress code to help collaborators understand the behavior of the code without the burden of going through thousands of lines of code; they document changes of their code for better version management and improving awareness of their collaborators; when others need to build upon their work, they write formal documentation and Readme files to describe how to use functions and API in their packages or services [112, 159, 204]. Data scientists write computational narratives as a practice of literate programming [89, 133], and as a way to think and explore alternatives. Thus, notebooks often have orphan code cells or out-of-order code snippets, which leads to lower reusability of the notebook and further highlights the importance of documentation in the notebook. As we found in our formative study, well-documented notebooks explain more than the behavior of the code. Notebooks cover various topics including describing and interpreting the output of the code, explaining reasons for choosing certain algorithms or models, educating the audience from different levels of expertise, and so on.

Thus, many interventions and lessons learned about documentation in software engineering may not apply in data science context. For example, how can we evaluate the quality of the documentation? Software documentation can be assessed based on attributes like completeness, organization, the relevance of content, readability, and accuracy [59]. Our experiment found that the quality scores assessed by these rubrics does not reflect users' satisfaction with the final notebooks. Despite many people's efforts to creating a standard documentation practice [154, 99], it remains questionable whether there is a one-size-fits-all solution. For example, Rule et al. [154] suggested ten rules for writing and sharing computational analysis in Jupyter notebooks. The first rule they proposed is to tell a story to the audience. However, this description is very general as people may approach storytelling differently. As we observed in Kaggle notebooks, some notebook authors prefer to use concise and accurate language while others use more colloquial and creative language. These creative notebooks stand out and receive many votes and compliments from the Kaggle community. As we recognize documentation in data science as a fluid activity, traditional template-based approaches to aid documentation writing may not work in data science because they can not capture a broader aspects of documentation, and limit the expressiveness of storytelling.

We argue that future work should recognize the difference between data science and software engineering, and tailor the documentation experience for data scientists. For example, Callisto [184] harnessed the fact that data scientists engage in synchronous work and discussion, and used contextual links between discussion messages and notebook content to aid the explanation of notebooks.

5.5.2 Human-AI Collaboration in Code Documentation in Data Science

We argue that AI-assisted code documentation process can be viewed as a co-creative process in which machine learning fits into the human workflow and collaborate together to create documents in a notebook. The notion of a “partnership relationship” between human data scientists and AI has been discussion by Wang et al. [190], and is part of a larger research discussion by many others (e.g., [28, 162, 114]). We consider this partnership as broadly defined where an AI system does not need an avatar or a conversational interface, but this AI system should be designed to fit into the existing human workflow and assist some parts of the human task to improve the quality or productivity. Human-AI collaboration, as opposed to human-AI competition (portrayed by AlphaGo or DeepBlue),

should be the ultimate goal of human-AI interaction research. Various human-centered AI design principles (such as human-in-the-loop) are means to get to this end goal. Our study demonstrated another means to achieve human-AI collaboration, where we combined the fully automated neural network approach and the less advanced rule-based or prompt-based approaches. This design is to acknowledge the limitation of today’s neural-network modeling. Our result showed that the combined human + AI effort produced a satisfied level of quality at a *more rapid pace* than what human or AI could achieve alone.

We also observed the user interaction pattern in which the AI creates “first draft” of the documentation, followed by human review and editing, resulted in a final artifact that not only met the bar for quality, but exceeded it for the level of satisfaction. Participants were happier with their code documentation when they were assisted by the AI system to create it, rather than when they worked alone. Thus, we conclude that the benefits of having an AI partner in this task stem from being able to produce the same high level of objective quality, but at a much more rapid pace (20% faster on average) and with a higher level of satisfaction with the end product.

We speculate that one of the contributing factors for why people were accepting of the AI’s suggestions is because the final decision of taking those suggestions was up to the human. As an alternative, we could have designed the system to always automatically produce a markdown documentation cell for each identified code cell, but we decided not to. Because this fully-automated design is an extreme in the framework of automation put forth by Parasuraman et al. ([131]; see also [79]), which people may feel being replaced. Our results confirmed our assumption — participants reported that they enjoyed being able to see multiple suggestions, created using different algorithms, and select the one that was the closest match to their intent in documenting a code cell. This level of interaction corresponds to “AI executes a selection only after a human has approved” in the Parasuraman et al. model [131].

Our result also shed light on the research question in [190] about the conditions under which human data scientists will enjoy working with AI partnership. In our case, maintaining control of the initiative and the final decision is an important aspect for people’s enjoyment and acceptance of the AI system. It remains to be studied whether people prefer both to control their own initiative *and* the initiative of a machine teammate, as proposed in Shneiderman’s recent two-dimensional model [163]. Also, we did not focus on the explainability or trust aspect of the designed AI system, such as how to visualize the connection between the generated documentaiton and the original code. In the future,

the explainability and trust aspects of the AI system in the data science context is a very critical research topic (e.g., [191, 52]), and should also be prioritized in the research agenda.

There are many other tasks in a data science project’s lifecycle that could use AI’s help, such as model presentation or feature engineering [187]. In the future, we plan to extend our work to design more human-centered AI systems to support users in these data science tasks as well.

In the future, we plan to explore whether the identified benefits and tradeoffs persist or not after a long period of adoption by users. One of the potential benefits could be: the human-AI collaboration work style helps users to learn more from the AI suggested/reminded documentations, thus they realize more of the value of adding code documentation to notebooks; in contrary, maybe users become over-reliance on AI systems thus they de-skill in this code documentation task, both hypotheses await future research to evaluate.

5.5.3 Design Implications

We offer designers and tool builders the following suggestions to encourage data scientists to write better documentation:

5.5.3.1 Towards Hybrid and Adapted Code Summarization

Our evaluation of Themisto indicates that instead of a fully automatic approach, data scientists prefer to use a hybrid method for helping them write documentation. We argue that future work for code summarization should investigate a hybrid and adapted approach. We suggest that *adaptive interactive prompting* may be a worthwhile research topic. For example, prompts could be based on the contents of the code cell which the user was trying to document. Another possibility is that prompts be based on the user’s own history of writing markdown cells, and could either appeal to the user’s strengths, or could anticipate and accommodate the user’s weaknesses. In a more socially-oriented approach, users within an organization might rate the initial set of prompts, voting some prompts up or down depending on their usefulness. An evolution of this idea might allow users to propose new prompts for use by selves and others (e.g., [48]). Furthermore, we argue that future code summarization tools would benefit from a reinforcement learning approach which learns from users’ modifications to the original proposed texts, and could

anticipate the users' preference in subsequent documentation.

5.5.3.2 Customizing the Recommendations based on Usage Scenarios

As prior work stated [204], data science workers engage in various collaborations during different stages of the data science lifecycle. Documentation plays an important role in many scenarios. For example, handing off work between data engineers, communicating results with stakeholders, or informal notes to future self. Data science workers may have different needs of the documentation for different usage scenarios. Designers and tool builders should take a user-centered approach to understand the purpose of documentation, the appropriate level of details, and the best way to present the documentation. For example, participants suggested that future versions of Themisto being able to document the changes of versions, errors, and related online forum posts. Participants also suggested that they would like to see more varieties into the generated narrative's formatting.

5.5.3.3 Inverting Themisto – Automatic Code Generation from Documentation.

The premise of Themisto was to generate descriptive material based on program code. Following some of the ideas in Seeber et al. [162], we might invert this strategy. We recall that P3 told us that he wrote documentation in advance of writing the code itself. If there are other people who use the same discipline as P3, could we generate code from the descriptive text? We suspect that this idea would not work for just *any* textual description. However, there could be certain stylized ways of writing descriptions that might be translatable into code; pseudocode could provide a starting point for the design of such a stylized type of description. We recognize that this kind of approach would need to have a representation of code packages and libraries, so that it could generate code that was appropriately structured for those packages. Of course, package documentation could be used to construct such a representation.

5.5.4 Limitation

Our formative study only explores notebooks from the Kaggle corpus, which may leave out some varieties of markdown cells that only exist in messy notebooks that can benefit from the support of documentation generation. However, notebooks on the Kaggle platforms are based on real data and real problems, and they aim for rich explanations

and narratives, whereas other places do not have high-quality notebooks with rich documentation. Future work should expand the exploration on the other notebook corpus, for example, notebooks published with scientific papers which contain fine-grained documentation.

Our experiment has several limitations: it focuses only on the documenting (instead of coding) process, it is a controlled experiment study, and participants did not work on notebooks created by themselves. Thus, for example, we do not know how participants would perceive the usefulness of the tool in realistic notebooks, which may be longer and more complicated (e.g., having out-of-order cells) than the notebooks we provided. However, we believe the result is still promising to shed light on future research and future system design. Future work can explore the generalizability of Themisto through a long-term deployment study.

As for the Human-AI Collaboration research initiative, our work only reports the findings on how human and AI collaborated at a coarse-grained level (Table 5.6). In future work, we will have an in-depth analysis to break down the level of individual cells, and further analyze the difference between automatically-generated, co-edited, or manually-produced cells. This detailed analysis will help us to understand how human behavior and perceive the fine-grained collaboration and interaction with the AI partner. And such findings and their derived design insights could also help researchers who are studying Human-AI Collaborations in other usage scenarios (e.g., in Healthcare or in Educational settings [198]) beyond the notebook documentation context in this paper.

5.6 Conclusion

In this paper, we have designed and built Themisto to support human data scientists in the notebook documentation task. This research prototype also serves as a prompt to explore the human-AI collaboration research agenda within the automated notebook documentation user scenario. The system design is driven by insights from previous literature, and also by a formative study that analyzed 80 highly-voted Kaggle notebooks to understand how human data scientists document notebooks. The follow-up user evaluation suggested that the collaboration between data scientists and Themisto significantly reduced task completion time and resulted in a final artifact that not only met the bar of quality, but also exceed it for the level of satisfaction.

5.7 Acknowledgements

We thank all of our participants for their help in the study, and the anonymous reviewers for their valuable feedback.

CHAPTER 6

DITL: Improving Awareness with Data Changes

Data science is characterized by evolution: since data science is exploratory, results evolve from moment to moment; since it can be collaborative, results evolve as the work changes hands. While existing tools help data scientists track changes in code, they provide less support for understanding the iterative changes that the code produces in the data. We explore the idea of visualizing differences in datasets as a core feature of exploratory data analysis, a concept we call *Diff in the Loop* (DITL). We evaluated DITL in a user study with 16 professional data scientists and found it helped them understand the implications of their actions when manipulating data. We summarize these findings and discuss how the approach can be generalized to different data science workflows.

6.1 Introduction

Data scientists try different transformations, aggregations, and filters until their data is in a state appropriate for the given task [89]. When producing models from their data, data scientists similarly iterate on different model features, architectures, and hyperparameters [21]. Existing tools for tracking changes typically only tackle half of the problem: differences in code. Development environments, for example, allow users to compare differences in notebook code cells between committed revisions [14], and Verdant reduces the burden of foraging code editing histories in Jupyter notebooks [88]. Yet comparing versions of *data* throughout an analysis is just as important [67]. Code differences do not always reveal data differences. For example, removing missing values from one column of a dataset may also affect the distributions of the dataset's other columns. To track the

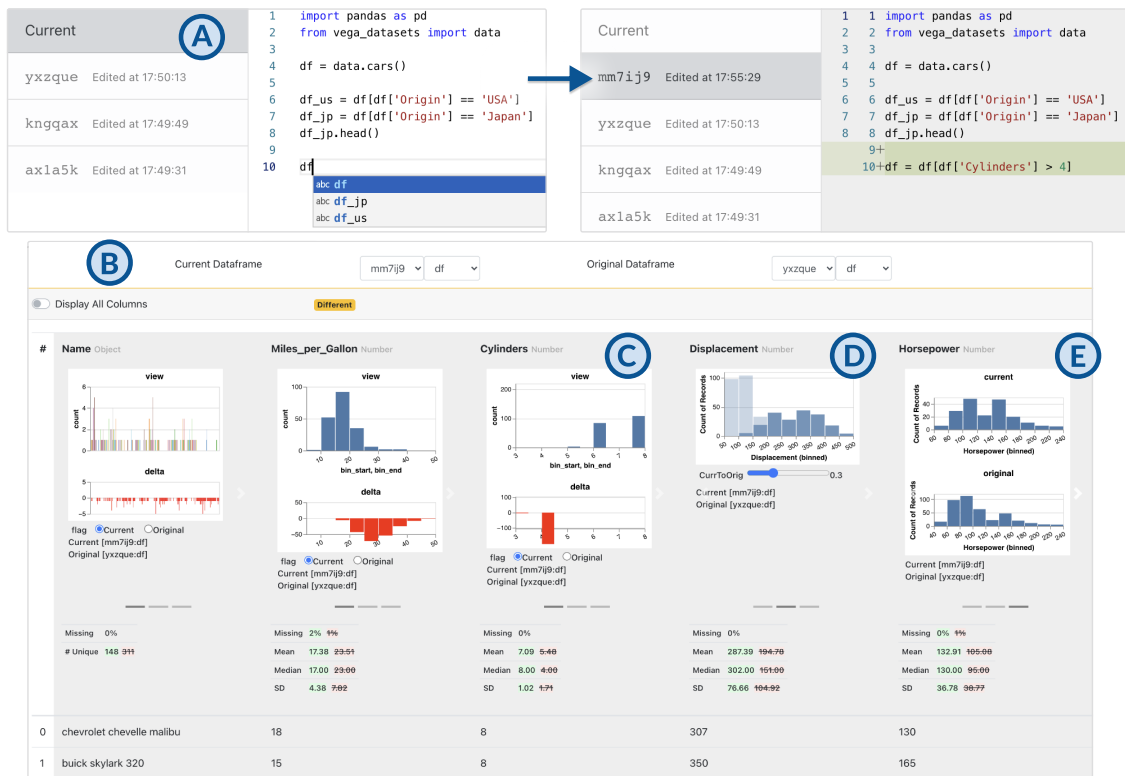


Figure 6.1: As users iterate on their data during analysis, they can use DITL to compare data snapshots. Every time users successfully execute code we save a snapshot (A). Users can compare the code using traditional code diffing tools. Users can also use DITL to compare data iterations with interactive visualizations, descriptive statistics, and data preview (B). User can choose three ways to visualize the differences in each column: delta view (C), opacity view (D), and parallel view (E).

effect that different lines of code have on the data currently requires data scientists to take the initiative to write additional code to browse or plot the data.

Recent work has begun to explore ways for analysts to understand and explain data iterations. Datamations uses animation to explain data transformation pipelines [145], and Chameleon allows analysts to compare data iterations simultaneously with model performance [78]. However, these approaches explain data differences *after* analysis has been done. In this paper, we explore adding visualizations of data differences as a core feature of tools for exploratory data analysis, a concept we call Diff in the Loop (DITL). Our DITL prototype stores a snapshot of the code and runtime variables as users make changes in the code editor. Using a table-based diff view (Fig. 6.1), users can either compare different datasets or compare the same dataset at different snapshots. When

comparing datasets A and B, the user can choose three ways to visualize the differences in each column: plotting histograms of A and B side by side; overlaying histograms of A and B, with cross-fading between them; or as a histogram of the user's chosen dataset (either A or B), plus a plot of the difference in the histogram bucket counts (either A subtracts B or B subtracts A). For each column, the DITL prototype also shows differences in descriptive statistics that are appropriate for that column's data (categorical or quantitative). The DITL view is designed to support both the explicit comparison of datasets and implicitly monitoring the evolution of a dataset as the user transforms it.

For example, Fig. 6.1 illustrates the effect of filtering a car dataset named `df` to those rows whose `Cylinders` column is greater than 4. The summary under the `Cylinders` column directly reflects this change: the top "view" plot shows a histogram of the current values (all above 4); the bottom "delta" plot shows that this step has the effect of removing rows with values 3 and 4, but keeping rows with values 5, 6, and 8. This column's summary confirms that the filter had the intended effect. Further, the DITL view also shows the effect this filtering step had on the other columns. For example, the distribution of `Miles_per_Gallon` lost the higher end of its distribution, with its median lowering from 23 to 17. Meanwhile, the columns `Displacement` and `Horsepower` lost the lower ends of their distributions. By having these data differences shown during exploratory data analysis, the user can maintain awareness of the effects that code has on the dataset as a whole, not just on columns mentioned in the code. Today, such awareness would require both the initiative and extra effort to write the code oneself to produce the plots and summaries.

We evaluated DITL in a user study with 16 professional data scientists, where participants were asked to finish typical data science programming tasks. They found DITL to be useful for tracking and understanding data changes. Furthermore, DITL improved their awareness of the side effects of some coding activities, guided them towards insights into the data, and reduced their workload for given data science tasks. We discuss the potential to integrate DITL in various data science programming tools and to generalize this approach for tracking changes in user-generated charts. To summarize, our contribution is twofold:

- A demonstration of the benefit of elevating data differences through visualizations to a core feature in a data science programming environment;
- Insights into users' needs and uses for leveraging both code and data differences during exploratory data analytic workflows through a user study with 16 data sci-

entists.

6.2 Design Motivations

To motivate the problem and guide the design, we present three typical usage scenarios that demonstrate how showing both code and data differences would be useful during exploratory data analysis.

6.2.1 Understanding the Impact of Code Changes in Debugging

In exploratory data analysis, data scientists write code to replace values in data tables, transform and combine data tables, or query subsets of data tables. Debugging data science code involves both ensuring that code changes compile, but that they also produce expected results [150]. However, existing data science code debuggers provide limited support for probing into the impact of code changes [30, 107]. Data science programmers often need to formulate temporary code queries to inspect data tables, which are likely to become stale or commented code that reduces the readability of the analysis scripts or notebooks [157]. Manual inspection is often performed on demand so analysts may miss unexpected impacts if they do not thoroughly explore the effects of code changes. Therefore, it is critical to inspect differences in both code and data throughout analysis. We believe that showing both code and data differences in data science programming environments can directly aid debugging.

6.2.2 Gaining Insights in Data Through Comparisons

Data scientists must make decisions throughout exploratory data analysis. Which features should be taken into consideration? How should null values be filled in? Does it matter if this part of the data is dropped? These decisions require making *comparisons* between whether or not a certain step improves the analysis. As opposed to comparing other types of variables, comparing data tables is exploratory and open-ended. Data scientists often need to tailor the comparison strategy according to the task. When tuning hyperparameters, data scientists must formulate a scoring function to compare the quality of the generated data tables. In model development data scientists must consider shifts in feature distribution, train test splits, and model performance when comparing data iterations [78]. In addition, understanding differences in model performance often requires



Figure 6.2: We integrate DITL into a simplified data science programming environment that allows data scientists to edit code, inspect data tables, and compare different data tables. This interface shows that the user is browsing a snapshot tagged 1k9i8j where the edit took place at 12:18:17. (A) Users are able to navigate among saved snapshots, compare code differences and output differences, or switch to the current code editor; (B) Users can edit code in the current code editor which automatically saves a new snapshot upon successful execution, or view code changes in a snapshot; (C) Users can switch between the output panel, the data panel, and DITL.

data scientists to consider beyond simply aggregating performance statistics. Comparing between data tables of the results can give them new insights on regions of impact on model changes. These examples demonstrate how comparison is an inherent task within exploratory data analysis.

6.2.3 Improving Awareness in Collaboration

Lastly, comparing data tables improves data scientists awareness of each others' work in collaborative settings. Data scientists rely on collaboration to improve the quality of their work [204]. Tracking and managing versions of scripts, artifacts, and documentation can help data scientists improve the efficiency of collaboration, reduce duplicated work, and avoid interference with each other [181]. Code versioning and editing sharing mechanisms (e.g., Git) in traditional software engineering can help data scientists managing code iterations when handing off work. However, it is not straightforward for data scien-

tists to interpret the impact of code changes unless they execute various versions of code and inspect the data tables thoroughly. We believe that showing and tracking both code and data changes can augment the existing data science collaboration tools by improving awareness of changes.

6.3 System Design

To address these use cases, we present DITL, a process of inspecting and comparing versions of data tables using interactive visualizations. We integrate the design into a simplified notebook experience so that we can examine how data scientists use it for comparing data tables. We choose to implement the simplified data science programming environment to highlight the utility of incorporating data table differences during exploratory data analysis without the distraction of other programming features included in existing tools.

6.3.1 Overview of DITL Study Apparatus

Figure 9.1 shows an overview of DITL study apparatus. As opposed to Jupyter notebooks, it has a single code editor for editing and running code. Users can make changes in the code editor (Figure 9.1B) or view the historical contents in previous edits. A snapshot (Figure 9.1A) is saved upon successful execution, which tracks the code content, output, runtime variable values, and a timestamp. Each snapshot is marked with a unique hashtag to aid in history navigation. Below the code editor, users can switch between the output panel, the data panel, and DITL. The output panel shows the results of users' consoles. The data panel (Figure 6.3) allows users to inspect the value of a single data table, using a design inspired by existing data table inspectors [170, 129]. As shown in Figure 6.3A, users can select saved data tables across different code snapshots. For each column, the data panel displays a visualization of the distribution (Figure 6.3B), summary statistics (Figure 6.3C), and sample rows (Figure 6.3D). Next, we elaborate on DITL and explain how the diff views are generated.

6.3.2 Tracking Runtime Variables

The programming prototype we built is able to collect runtime variable values upon every successful execution. The web-based interface executes Python code and stores the names and values of variables that are dataframes. This approach allows us to create snapshots

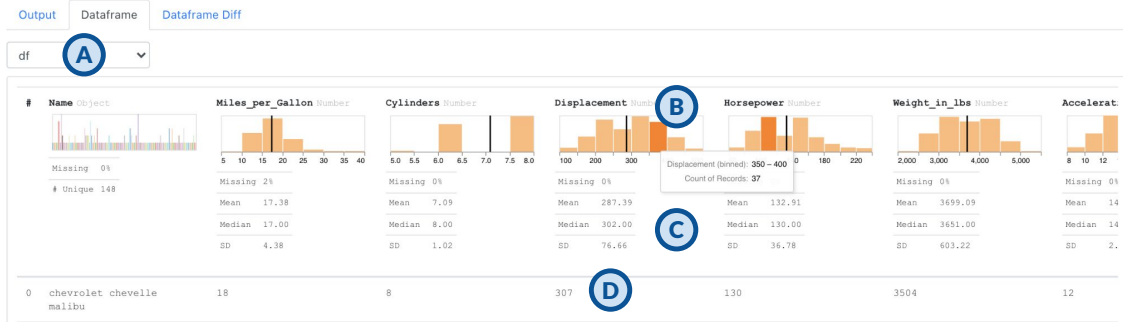


Figure 6.3: The data panel allows users to inspect a single data table. (A) Users can select saved data frames from the current code snapshot; (B) The data panel shows the distribution of each column; (C) The data panel shows the summary statistics for each column; (D) The data panel shows a sample of rows from the selected data frame.

during each code iteration which are later used to create the dataframe diff visualizations. This approach to tracking runtime variable iterations could be easily generalized to other data science programming tools like Google Colab [66], DeepNote [16], or Jupyter Notebooks [11].

6.3.3 Comparing Changes in Data Frames

In order to visualize the differences between data tables, we must first identify correspondences between two tables. Our notation and method for calculating data table correspondences is inspired by the notation used in the visualization library D3 [10]. In order to align the two data tables, we use the heuristics of comparing data frame index provided by the Pandas package [127]. Given an original (old) data table and a current (new) data table, we use five labels to describe their correspondences. **Both** corresponds to a point that is exactly the same in both the original and current data table. Updates occur when a point has the same primary key but some other column value has changed. **Update-Enter** refers to the newly updated point in the current data table, while **Update-Exit** refers to the old point in the original data table. Lastly, **Enter** corresponds to new points while **Exit** refers to deleted points. The labels are appended as an additional column in the joint data table.

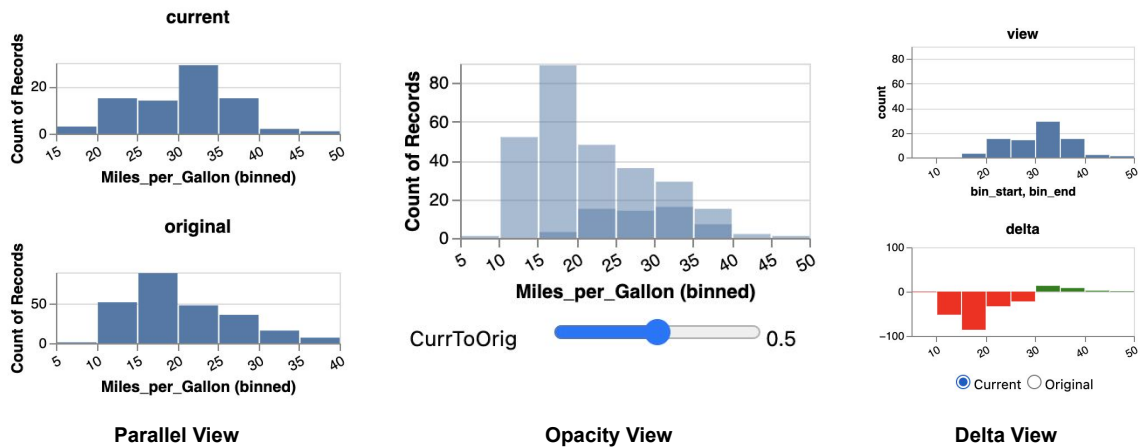


Figure 6.4: DITL uses three approaches for rendering data differences: parallel view, opacity view, and delta view.

6.3.4 Rendering Data Frame Diffs

As shown in Figure 6.1, we use this correspondence information to visualize the differences between two data tables. To eliminate information overload, by default, we only display the columns that have been changed. Users can click on a toggle button to display all columns. Through early pilot testing, we decided on three views for DITL: parallel view, opacity view, and delta view (Figure 6.4). The design of these views follows the best practices from the visualization community for supporting comparisons [63]. We implemented the interactive visualizations in Vega-Lite.

6.3.4.1 Parallel View

The parallel view shows the original and current distributions of the column side-by-side. We enable the tooltip to show detailed information about the distribution. While we purposely chose to implement the parallel view in a fashion consistent with the way these are currently displayed in comparison views in ReviewNB [149] or VSCode [14], the parallel view can be easily augmented with the option to display on common axes scales. We are aware of the potential issues with not unifying the axes and we included the view to validate these issues.

6.3.4.2 Opacity View

While the parallel view allows users to directly inspect and compare distributions of the columns, it is hard to visually compare the shape of the distributions since they may come in different scales. Thus, we designed an opacity view which overlays the distributions on the same axes. This design implements Gleicher’s guideline [63] that correspondences can be easier to track when the data is overlaid. We then use the opacity channel to map the “diff-label” information. Users can move the opacity slider to cross-fade between the current and original distributions.

6.3.4.3 Delta View

In our pilot user testing, users demonstrated the desire to visualize not only the distributions of the current and original data tables, but also the distributions of their differences. Thus, we introduce the delta view to explicitly show the subtraction results. As shown in Figure 6.4, the delta view contains two parts. The top view shows either the current or original distribution. On the bottom, the delta shows how the current distribution differs from the original by computing $N_{delta} = N_{enter} - N_{update_enter} - N_{exit} - N_{update_exit}$. Red bars represent negative values of delta, indicating the decreased counts of the data points falling under the bin, while green bars represent positive values of delta, indicating the increased counts of the data points falling under the bin. We purposely tweak the scale for the delta distribution to help amplify small changes.

6.4 Usability Study

We conducted a 60-minute long virtual usability study with each of 16 professional data scientists to understand the support that DITL can provide for common data science programming tasks. In particular, we sought to answer the following research questions:

- Do data scientists find DITL useful for comparing data tables?
- How might DITL provide them with additional insights into the differences between programming iterations?

6.4.1 Method

6.4.1.1 Recruitment

We randomly selected 200 data scientists at a large software company based on their job titles and sent them recruitment emails. To be eligible for the study, participants had to self report at least basic experience with Python programming. We recruited 16 participants altogether (3 females, 12 males, 1 prefer not to say). Three of our participants had less than 1 year of professional data science experience, 11 had 1-5 years of professional experience, and two had more than 5 years of professional experience. Their job titles included data scientist (9), senior data scientist (4), principal data scientist (1), research scientist (1), and senior machine learning scientist (1). We compensated participants with a US\$25 Amazon gift card.

6.4.1.2 Study Setup

The usability study was conducted remotely with participants sharing their screens over a video conferencing tool. Since the DITL study apparatus is a web-based programming environment, participants were able to use the tool on their computers within their own choice of browsers and configurations. Each study consists of three sessions — a training session and two experiment sessions. After a brief walkthrough of the prototype, we gave participants a trial task to get familiar with the tool. We presented them with an ongoing code session to explore a dataset about cars [9]. The trial task is scaffolded into four activities: using the data panel to inspect a given data frame; using DITL to compare the differences between two data frames; understanding historical edits to the code and the data frame; and, modifying the current code to include an additional step for exploratory data analysis.

After the training session, we gave participants two existing data science tasks modified from online data science challenges. One task is about customer satisfaction (noted as T1), which is modified from Kaggle [12]. The other task is about salary analysis (noted as T2), which is modified from TidyTuesday projects [13]. We chose these two tasks because they are shared on popular data science communities [164, 35] and are perceived to be representative of real-world data science tasks. Since the original challenges are open-ended and time-consuming, we scaffolded the tasks into three subtasks: one for cleaning duplicate presentations in data (noted as S1), one for exploring subsets of the data (noted as S2), and one for evaluating two model prediction results (noted as S3). To maximize

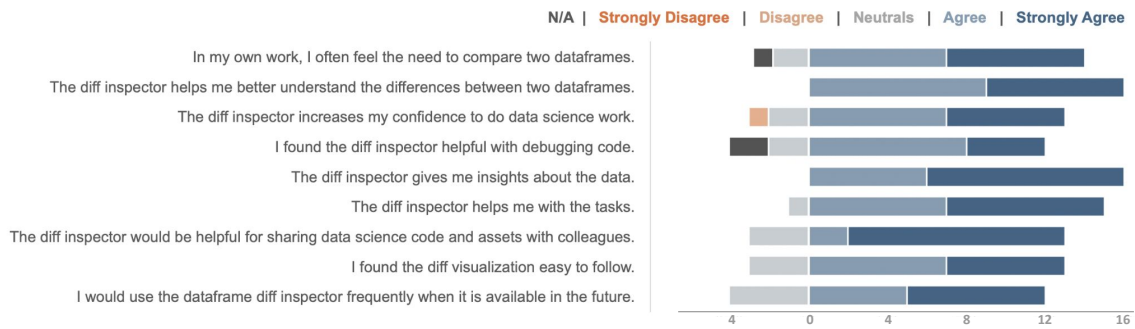


Figure 6.5: Participants’ responses to the likert scale questions in the post-task questionnaire.

the time on experiencing DITL, we provided participants hints and code cheatsheets for the given tasks, and allowed them to ask API-related questions. We counterbalanced the order of the tasks between subjects. For each task, participants are randomly assigned to solve it with or without the DITL. We encouraged participants to think aloud throughout the tasks.

Lastly, we asked participants to fill out a post-study questionnaire and reflect on their experience with the tasks. Two members from the research team observed each study session and took notes. We recorded the screen sharing of the study sessions and transcribed the audio recording.

6.4.2 Results

6.4.2.1 The need to compare data tables

Our evaluation showed that comparing data tables is a common activity in various data science tasks. During the study tasks, we frequently observed participants comparing data tables for various purposes. For S1, many participants inspected and compared the data tables before and after changes to validate whether the code edits worked as they expected (14/16). Comparing data tables is also an essential step for generating insights for exploration purposes. For example, for S2, all participants (16/16) compared the subset of the dataset with the original dataset in order to understand the side effects of the filtering query and come up with their next step. Participants also reported the need to compare data tables to make decisions between solutions. For S3, all participants (16/16) evaluated the performance of the models by comparing the model prediction results either using DITL or writing code for inspections. In the post-task questionnaire, most partic-

ipants (14/16) agreed that they often need to compare two data tables in their own work (Figure 6.5). For example, P14 mentioned that their work involved collecting new data during model deployment:

One thing we do is comparing the original training data with the scoring snapshots of the weekly changing data. This is something [comparing data iterations] we should do but we did not do as often. (P14)

6.4.2.2 DITL makes comparison easier

We observed several different strategies for comparing data tables. When DITL was not available, participants wrote code to manually understand and compare data tables. For instance, they printed summary statistics, previewed the first five rows of the data tables, manually created distribution plots, or formulated customized queries for examining a specific attribute (e.g., the ratio of female respondents to male respondents). Most participants used DITL (15/16) when it was available during the analysis. As shown in Figure 6.5, participants agreed (15/16) that DITL helps them with the tasks. They explained why DITL makes comparison easier.

One advantage of DITL is that besides reducing the amount of code that participants wrote, it also eliminated code that was there just for verification or validation purposes. When DITL was not available, participants wrote code for logging and querying attributes. This process would produce additional code, reduce the readability of the analysis, and potentially lead to the rabbit hole of debugging code that was not part of the primary analysis efforts. For example, P4 suggested that DITL helped her maintain a cleaned code space:

This is useful for quick visual inspection across data frames. I find this helps to avoid intermittent logging and debugging during the development process.
(P4)

We counted and compared the total lines of code that participants produced for completing the tasks. Unsurprisingly, participants using DITL wrote significantly fewer lines of code (17.08 lines vs 25.38 lines, $p < 0.001$ two-sample t-test). Some participants mentioned that this tool could be helpful for novice data scientists who are less familiar with relevant APIs (P9) or for explaining changes to people who are not on the technical side (P10):

That [DITL] is way easier; The diff is really helpful for analytical purpose;

I think this would help people like me to show the changes to other people who might not know the technical side. (P10)

Next, participants perceived that DITL helps them discover insights about the data (16/16). Participants described the tool “*directly explains what is going on*” (P10), “*allows me to instantly look at the differences*” (P16), “*gives me a big picture*” (P7), and “*is helpful for formulating the next steps*” (P8). In addition, participants mentioned that the visualizations helped them understand the side effect of code edits: “*I was not aware of the changes in column ‘Horsepower’ when I applied a filter on the column ‘Cylinders’ until I used the tool.*” (P6)

6.4.2.3 Feedback on the Visualizations

Overall, 13 out of 16 participants found the visualizations easy to follow (Figure 6.5). Two participants mentioned that their lack of familiarity with interactive visualizations “*is getting the way*” (P4) and wished to “*have more practice to use the visualizations*” (P9). Participants also made comments on the usefulness of the three different approaches for visualizing the changes. As expected, there was not a single “best” view. Rather, the three views are complementary depending on the task:

Not necessary every view was useful for different tasks. It is hard to say which one is the best for all. It kind depends on the task. (P6)

The parallel view is perceived to be “*straightforward*” (P1, P6). One participant described the parallel view as the default approach they would use when manually comparing two distributions (P6). This corresponds to our rationale to include the parallel view — to simulate the go-to approach for comparing the distributions by plotting them side-by-side. However, other participants critiqued that this approach “*seems not that helpful*” (P13) and even “*misleading*” (P3, P13). They raised concerns that this view was not intuitive for understanding changes and could be misleading due to the inconsistent axes and scales (P3, P13).

Participants had split attitudes towards the opacity view and the delta view. Five participants (P1, P2, P5, P8, P14) were in favor of the opacity view most, and described it as “*intuitive*” and “*easy to understand*”, particularly for observing shifts in distributions. For the delta view, six participants (P4, P7, P9, P13, P15, P16) explicitly mentioned it being most helpful. They found it particularly useful upon slicing and selecting subsets (P4), providing the exact differences in counts in the tooltip (P15). Yet, some participants

reported that it requires more time for them to understand the delta view than the two other views (P6, P8).

6.4.2.4 Preferences for integration

Participants' feedback on future integration helps us validate the design motivations. Overall, 12 out of 16 participants responded in a positive manner that they would frequently use DITL if it were available in the future (Figure 6.5). For debugging and understanding the impact of code changes, 12 out of 16 participants were positive on the usefulness of DITL. Participants explicitly mentioned future usage of “*debugging customers' data over time*” (P5), “*validating results of cleaning*” (P15), “*time travel debugging*” (P2), and “*debugging during the dev process*” (P4). For supporting decision making, all participants agreed that DITL gives them insights about the data. In particular, P3 described how DITL can be useful to compare A/B experiments:

Looking for distributional shifts between A/B experiments. Where the distributional information is hard to summarize into a neat hypothesis test, the visual chart really helps. (P3)

Lastly, 13 out of 16 participants agreed that DITL would be helpful for sharing data science code and assets with colleagues.

Participants described how they see DITL working in their own data science workflows. They mentioned integrating DITL in existing data science IDEs like PyCharm (P15), Jupyter notebooks (P6, P14), RStudio (P10, P13), and VS Code (P7, P8, P9) for tracking and comparing data tables. Some participants mentioned data science collaboration tools, for example, integrating DITL as part of the git versioning experience (P1), or augmenting real-time collaborative editing tools like Google Colab (P16) with DITL.

In addition, participants provided suggestions to further improve the comparison feature. Participants wanted tailored comparisons over certain data types. For example, P15 suggested adding visualizations to demonstrate text attributes, such as word length, number of characters or character sets, different topics. Participants also mentioned the need to compare visual outputs beyond data tables:

Maybe in the future, users can compare other kinds of graphs than distribution plots. (P16)

6.5 Discussion and Future Work

6.5.1 Towards a Design Space for Visualizing Data Comparisons

Since the goal of our project is to investigate the idea of data comparison in exploratory data analysis, we did not extensively explore the design space for these visualizations nor did we evaluate these possible designs. What we learned at this stage suggests empirical evidence for the utility of using DITL in exploratory analysis. Participants felt that the effectiveness of the visualizations themselves greatly depended on the task at hand and that there was no single visualization that fits all circumstances. For example, the opacity view might be more suitable for observing the trend of shifting in distributions (e.g., correcting skewed distributions through log-transforms); while the delta view might be more suitable for showing slicing and filtering to highlight the changes on individual data bins. In addition, our approach of encoding the diff information in additional channels can be extended to create other types of visualizations, for example, a grouped bar chart rendering the current and original distributions along the same axes, or a facet view showing the distribution of data points marked as “new” or “absent”. Future work can continue to explore this design space and evaluate the usefulness of the views for various data science tasks.

6.5.2 Generalizing from Comparing Data Tables to Comparing Arbitrary Charts

DITL demonstrates the idea of tracking and visualizing changes in data tables in data science programming environments. We further argue that the same techniques used for visualizing the differences in iterative changes of data tables can be generalized to visualizing changes on a wide variety of charts. Typically, data scientists make two types of changes on charts: changing the underlying data or changing the visual representations. If the visual representation and data schema remain the same while only the underlying data changes, a similar approach can be used to first combine the original and current data tables to encode the diff information for each data point. This diff information can subsequently be rendered with an unused channel (e.g., opacity, color, facet, or z-axis) in the visual representations. Interactions such as sliders, selections, or brushes can be used to switch between original and current charts. In addition, we can filter the combined data table and explicitly render the subtractions. For example, the delta view can be

generalized in charts to represent the visual differences in the visualization. On the other hand, if the visual representation or the data schema changes (e.g., table pivoting), there is an opportunity to combine the stateful interactive visualizations with animations (e.g., SandDance [161], Datamation [145], Gemini [94]) to explain the transitions. Lastly, if the changes in charts are multifold, future work can look into ways to break the changes into the combination of data changes and visual representation changes.

6.5.3 Integrating DITL in Data Science Programming Environments

In this paper, we demonstrate the idea of DITL in a customized data science programming tool. To integrate DITL in existing data science programming environments, both scalability and task complexity must be considered. In particular, the timescale for creating snapshots of the data iterations should be tailored to the context. For programming IDEs that allow execution of script files (e.g., PyCharm), tool designers can leverage built-in debuggers to track variable values upon each execution and map versions of variable values to the snapshot of the scripts. For REPL-based programming environments that allow interactive execution of code snippets (e.g., Jupyter Notebook), mapping the versions of variable values with the execution orders and the state of the notebooks can be a challenging task. Future work can explore how approaches used in foraging code versions (e.g., Verdant [88], Gather [75]) can be extended for foraging data iterations. In collaborative data science programming environments, the timescale for creating snapshots should be tailored towards tracking data iterations and hand-offs between collaborators. For example, versioning tools like Git or real-time editing tools like Google Colab can support the diffing of the data tables and charts when synchronizing collaborators' edits. Lastly, the idea of comparing data changes can be helpful in live programming environments. Live programming is a programming paradigm recently emerging in data science communities (e.g., Observable Notebook [18], Glinda [47]). Compared to REPL-based programming, live programming updates the execution immediately upon editing [45]. Although live programming is favored for providing a responsive and consistent experience for exploratory data analysis, the live experience hides history and may result in mismatched expectations for the automatic execution [45]. Future work can explore the idea of showing both code and data iterations in live programming environments for browsing and resurrecting histories.

6.5.4 Limitations

6.5.4.1 Limitations of DITL

DITL is tailored towards comparing data tables with changes to the column values without altering the schema. DITL is able to detect small changes to the schema such as adding, deleting, and renaming column names, while not able to handle full schema transformations like data pivoting. Recent work [145] has used animations to explain operations such as pivoting that might be incorporated into future work. In addition, DITL only compares two data tables. Future work can explore ways to make comparisons between multiple data tables.

6.5.4.2 Limitations of the Evaluation

Our user study has several limitations. First, in order to control the complexity of the tasks and the duration of the study, we gave participants data science tasks modified from online challenges instead of evaluating the tool with their own tasks. Second, we scaffold the tasks to ensure that novice data scientists were capable for performing the required tasks. To further prevent diluting the focus of the study, we provided immediate, verbal assistance to them when they got stuck on the programming tasks. We did not evaluate performance in terms of time as we expected this might be affected by participants' familiarity with the tool. Most statements in the post-task questionnaire are positively framed, which could cause a priming effect. Future work should consider long-term deployment to further examine the usefulness of the tool in open-ended, real-world data science tasks.

6.6 Conclusion

This paper presents the idea of Diff In The Loop (DITL), integrating data differences through visualizations as a first class citizen in data science programming environments. We illustrate the usage of comparing data tables in three usage scenarios grounded in prior literature. We implement a prototype that incorporates DITL and show how comparing data tables through visualizations can help in exploratory data analysis. The evaluation of this system confirmed the needs and benefits of showing both code and data differences during exploratory data analytic workflows. In particular, DITL helps data scientists understand the implications of their actions when manipulating data.

6.7 Acknowledgements

We thank all of our participants for their help in the study, and the anonymous reviewers for their valuable feedback.

CHAPTER 7

PADLOCK: Resolving Editing Conflicts in Real-Time Collaboration

Real-time collaborative editing in computational notebooks can improve the efficiency of teamwork for data scientists. However, working together through synchronous editing of notebooks introduces new challenges. Data scientists may inadvertently interfere with each others' work by altering the shared codebase and runtime state if they do not set up a social protocol for working together and monitoring their collaborators' progress. In this paper, we propose a real-time collaborative editing model for resolving conflict edits in computational notebooks that introduces three levels of edit protection to help collaborators avoid introducing errors to both the program source code and changes to the runtime state. Through a set of evaluations, we found that these three levels of edit protection make collaborators more satisfied with their collaboration experience compared with notebooks without these features and are perceived to be useful in a variety of collaborative settings.

7.1 Introduction

“You work on *this* section, and I’ll work on *that* one” is a familiar refrain for authors who work in teams. Working on different portions of the same document is a natural way to combine collaborators' work while preventing conflicts [142]. In the context of data science programming, collaborators use a variety of collaborative strategies including “divide and conquer” (splitting work between team members), “competitive authoring” (working on the same sub-problem simultaneously), and more [181].

However, computational notebooks like Jupyter, which are often used by data scientists, introduce new challenges for collaboration. Although some version control tools (e.g., Git) can be used for computational notebooks, they mostly support the collabora-

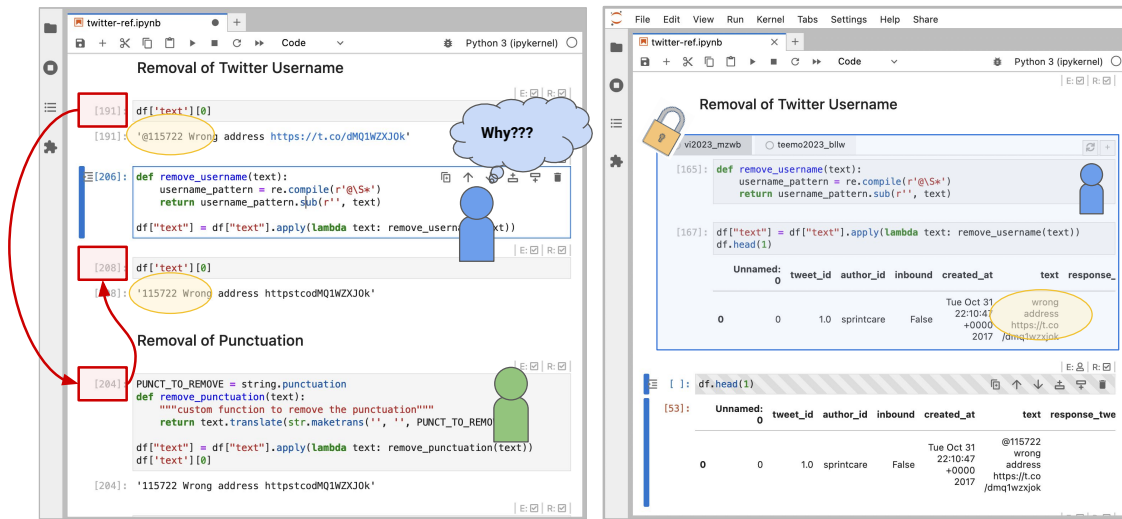


Figure 7.1: Editing conflicts in real-time collaborative notebooks can be implicit. As shown on the left, one can get an unexpected execution result because the collaborator accidentally changed the shared variable. As shown on the right, PADLOCK helps data scientists resolve editing conflicts in real-time collaborative editing in computational notebooks.

tion strategies for dividing work (e.g., working in separate, interdependent files). Further, data scientists sometimes collaborate *synchronously*, with tools like JupyterLab¹, Google Colab², Deepnote³, and RStudio Cloud⁴ that broadcast code and runtime updates to collaborators in real-time [181]. Consequently, dependencies between different authors' code can hinder collaboration, as upstream changes can break downstream dependent code [65, 64]. This includes changes to code that defines variables that will be subsequently referenced or even direct changes to the shared runtime state (as shown in Figure 7.1). Finally, data science work is often exploratory, so the notebooks produced might be unorganized, fragmented, and poorly documented draft code [157].

Computational notebooks also introduce new opportunities for improving collaboration between data scientists. Synchronized collaborative computational notebooks allow data scientists to see each other's edits as they happen and share documentation, code, and an interpreter runtime state. Real-time collaborative editing improves data science teamwork by creating a shared context, encouraging more explanation, and reducing commu-

¹<https://jupyter.org/>

²<https://colab.research.google.com/>

³<https://deepnote.com/>

⁴<https://www.rstudio.com/products/cloud/>

nication costs [181]. Real-time collaborative editing may also benefit presenting and reproducing results in the workplace or educational settings [103]. However, editing these shared computational narratives together comes with unique challenges [181]. To address these issues, data scientists establish social protocols similar to those found in other collaborative writing situations [122] but contextualized to the computational narrative workflow. For instance, an author may take ownership of a code cell and use formatted Markdown or code comments to indicate to collaborators they don't want others to edit the contents. Depending on the workflow of collaboration, this might result in multiple variants of an analysis which then need to be managed and potentially reduced to a single narrative. For instance, after two data scientists work together on a problem, they may decide to only keep the most efficient solution found and delete one. Alternatively, two competing solutions have unique characteristics which warrant keeping them both. For instance, if different machine-learned models highlight unique aspects of the underlying data, the authors might wish to integrate both into a single narrative.

Inspired by these challenges and opportunities, we propose a set of interactive techniques to minimize collaboration friction while maintaining the readability of the shared notebook. We instantiate these techniques in PADLOCK⁵, an extension to the open source JupyterLab platform that provides three different mechanisms to support conflict free collaboration. PADLOCK leverages the context of data science development to provide three domain-relevant mechanisms to improve collaboration on computational narratives. The first, *cell-level access control*, prevents collaborators from viewing or editing a collection of cells. This mechanism aims to allow ad-hoc locking of code cells to support volatile collaboration patterns. The second mechanism, *variable-level access control*, extends the access control from cell-level to shared variables. This mechanism is designed to prevent implicit editing conflicts and allow collaborators to protect important shared variables. The third, *parallel cell groups*, leverages the familiar programming concept of encapsulation through scoping with the Jupyter cell user interface control. This mechanism allows individuals to pursue exploratory solutions while not having to be concerned about interference with others. Our evaluation of PADLOCK has shown that these mechanisms can effectively prevent editing conflicts in shared notebooks; and they support a wide range of ad-hoc and volatile collaborative workflows.

This work makes several contributions that advance the state of the art for collaborative data science tools:

⁵PADLOCK is short for “Parallelization And Data Locks Offset Collaboration Kinks”

- We introduce a mechanism (cell-level access control) that can support collaborative data science work by giving authors more control over who can view or edit sensitive cells
- We define variable-level access control mechanisms that give data scientists more control over the runtime state of shared notebooks
- We enable “parallel cell groups”, designated areas where data scientists can manipulate and share their own ideas
- A system (PADLOCK) that instantiates all these features in a JupyterLab plugin
- We present three evaluations of PADLOCK (paired sessions, planning sessions, and group sessions) to better understand how these features can be used in collaborative data science.

Further, to the best of our knowledge, PADLOCK is the first system to:

- Give users the ability to specify access control constraints at the level of individual cells in computational notebooks
- Allow programmers to specify which collaborators (as opposed to which code fragments) can access or overwrite specific variables
- Allow data scientists to work in “parallel cell groups”, that are scoped in a way where they can access and reference each other’s work without worrying about introducing conflicting code

7.2 Design Motivations

To motivate our design, we identified three typical real-time collaboration scenarios in data science that would cause conflict, synthesized by the challenges in real-time collaboration by Wang et al. [181].

7.2.1 Implementing the Same-Purpose Code at the Same Time

Data scientists would adopt different collaboration styles in their work, including competitive authoring, divide and conquer, single authoring, and pair authoring [181]. Editing

conflicts can arise when collaborators are adopting the competitive authoring collaboration style, editing cells that attempt to solve the same problems to explore different alternatives. When multiple people are attempting to edit the same cell, they must be in close collaboration and frequent communication to avoid conflicts. Thus, in competitive authoring scenarios, some people would choose to make copies and only edit their own copy as a way of claiming ownership of some cells. For example, Alice and Bob are experimenting with different ways to pre-process the data. They each created a few cells to finish their explorations. Although they are working on separate cells and “claim” these cells, it does not eliminate the possibility of other users accidentally making edits. Although some commercial platforms such as Deepnote and Hex prevent two users from concurrently editing the same cell, the solution is not perfect - once the “cell owner” stops editing the cell, other users can start editing, even when the owner was only pausing their activity in that cell and would come back shortly after, not expecting changes from other people. In many situations, after each collaborator generates a solution, they would want to save previous exploration results and keep a clean computational narrative. However, these two needs are often in conflict.

7.2.2 Using or Changing the Same Variable at the Same Time

Even when data scientists are adopting collaboration styles such that the tasks they are working on are different, such as divide and conquer, conflicts can happen when multiple users are using or changing the same variable at the same time. In the second scenario, we demonstrate that when people are running code that affects the same variable at the same time, unexpected conflicts can happen.

Imagine Alice and Bob are doing different analyses on the same variable that contains the dataframe. In the working process, it is easy for collaborators to forget they are sharing the same variable, and start to make edits that are unexpected to other collaborators. When a person changes the variable, it could cause conflict in other people’s exploration. In a more extreme situation, they might both give a new variable the same name, unaware that their actions on the variable would influence other collaborators.

Aside from protecting variables from being changed unexpectedly, there are also occasions where data scientists need to sync the changes made by their collaborators working upstream. In this case, Carol might be working on data cleaning that is upstream of Alice and Bob, and they would need to sync the variable from Carol once they finish. In terms of preventing conflict in variable change, we would also need to provide flexibility

in syncing from other collaborators.

7.2.3 Other Needs for Access Control in RTC

The need for access control not only comes from preventing collaborators from unconsciously messing up other collaborator's code or variable, but can also come from other aspects, such as concern for social image or project management.

For example, in a team with different expertise, people may shy away from attempting exploratory work when they are aware that their progress is visible to other collaborators, especially more senior ones. In an educational scenario, novices might be self-conscious about their half-finished code in a collaborative tool being viewed by peers, and might choose to explore their code in a separate environment and paste the results back after they finish; or instructors might want to hide the solution from students before they finish the exploration.

Another reason that calls for access control is from the management perspective. In a larger team with multiple collaborators and a leader, the leader might want to manage the tasks for individuals and give different levels of access. For example, they might want to freeze the code that is ready and does not want any change, such as importing data. Or they might only want a subset of people to edit a certain part of the notebook, such as letting machine learning experts work on training models, and domain experts work on exploratory data analysis to provide more insights. Although many commercial real-time collaborative platforms support notebook-level access control, more fine-grained access control would be appreciated in many scenarios.

7.3 System Overview

Our system uses three complementary techniques for conflict-free protection: cell-level access control, variable-level access control, and parallel cell groups.

7.3.1 Cell-Level Access Control

Computational notebooks consist of *cells*. Each cell typically represents a conceptual unit within the larger notebook. For example, a notebook might consist of one cell to fetch data from a remote API, another to clean those data, and other cells for various transformations and visualizations of the data. In PADLOCK, we leverage the structure

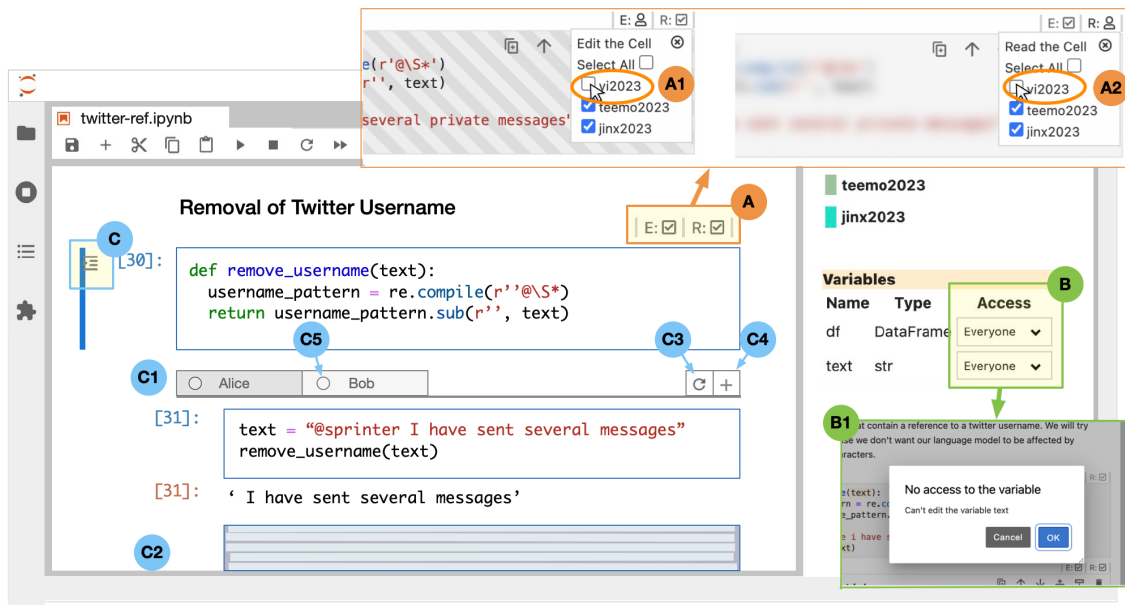


Figure 7.2: Overview of the three conflict-free mechanisms in PADLOCK

of cells in order to allow collaborators to claim ownership of parts of larger collaborative notebooks. This helps address the challenges of synchronous editing in traditional text-based programming tools where there are no clear “dividing lines” between different parts of the shared codebase and unclear how to localize the scope and effects of collaborators’ changes.

Specifically, PADLOCK enables *cell-level access control* where users can prevent collaborators from viewing or editing a collection of cells. As Figure 9.1.A shows, users can select a code cell and specify who can read or edit the code. As prior studies have found, there are many collaboration styles [181], and cell-level access control benefits multiple collaboration styles. In a “single authoring” style [181]—where one collaborator contributes the majority of ideas and code—setting cells to be only editable by the main contributor can prevent others from accidentally introducing errors. In a “divide and conquer” style [181]—where collaborators split up work—restricting view access might ease feelings of self-consciousness that authors sometimes feel when collaborators can see their writing in real-time (which might otherwise lead them to work in a private editor and then copy its contents to the main notebook, as prior work found in collaborative writing [189]).

When an author is restricted from editing a cell, the background of the cell (grey striping) indicates that edit access is not permitted. When an author is restricted from reading

a cell, the content of the cell is blurred but activity (and thus awareness of contributors' location in the narrative) is supported.

Thus, the view control of the cell can allow them to focus on early explorations of ideas while let others aware of where they are working on.

7.3.2 Variable-Level Access Control

Restricting cell access gives collaborators control of how the runtime state is determined (as determined by the code that computes variables' values). However, it does not prevent other cells from subsequently modifying the runtime state. For example, a user might create a cell that defines `df` as a data frame (data in a table-like structure) and restrict write access to the cell. Other collaborators cannot edit the cell that declares `df`. However, they could create a new cell that either re-declares or mutates the value of `df` and breaks downstream code that references the variable.

Thus, PADLOCK also introduces *variable-level access control*. Variable-level access control extends the idea of access control from cells to shared variables—authors can determine if collaborators' code can view or modify the values of runtime variables. PADLOCK tracks the runtime state of the notebook kernel and extracts the variable information. Users can specify the access control of every variable in a side panel (as shown in Figure 9.1.B). On the other collaborators' side, the protected variable is highlighted throughout the notebook. When an individual attempts to execute a code cell a static analysis on the abstract syntax tree (AST) of the program is done to determine whether the execution would impact the value of protected variables and, if so, the execution is halted with an error.

Variable-level access control is especially beneficial for scenarios where there is a lead collaborator in charge of managing important data tables. Setting variable-level access control can encourage collaborators to either make a copy or use parallel cell groups before they do any risky explorations.

7.3.3 Parallel Cell Groups

Data science work is often exploratory. Authors might write code to explore an idea or approach. In the context of teams, multiple team members might simultaneously work through different approaches for the same problem [181]. In these situations, authors might want to write code that manipulates *their own version of* some subset of variables

in the notebook as they explore.

PADLOCK thus also introduces *parallel cell groups* (which we will call “parallel cells”). Parallel cells define a designated area where changes of the code and runtime state stay inside its own scope. As Figure 9.1.C shows, users can split a regular code cell into parallel cell groups. Collaborators can create new cell groups to branch off and explore alternatives; add multiple cells to a cell group to write larger and more complex alternative code; and work individually in each cell group. The parallel cell groups are folded together into the same area in the notebook, helping collaborators to maintain an overall coherent structure of the narrative. In addition, when collaborators are settled on a solution, they can mark a cell group as “primary”, which merges the execution result into the main runtime state. Note that the parallel cell groups designed in PADLOCK are different notions than the forked cells in [192] in several ways. In terms of the usage scenario, [192] is designed for a single developer to explore alternative ideas, whereas PADLOCK is designed for synchronous computational notebooks. For the implementation, PADLOCK uses a scoping mechanism instead of spawning multiple kernels, making it easier for managing different versions of the same variable.

A key difference from prior tools for branching and managing local versions [87] is that each cell group has its own execution scope, which means changing the variables in cell group A would not affect the value in cell group B. For example, suppose there is a parallel cell group named `pl1`, and within those cells, code creates variables named `x` and `y`. Inside of the cell group, `x` and `y` are defined as normal (referencing them returns the value that they were set to in `pl1`). Outside of the cell group, code that references variables `x` and `y` get their ‘old’ values (or an error if they were not set outside of the cell group). However, these variables can be referenced outside of the group if the user explicitly specifies which scope they want to reference. So while `x` and `y` are not affected outside of `pl1`, collaborators can refer to `_pl1.x` and `_pl1.y` to access the values that were set inside of `pl1`.

Parallel cell groups allow collaborators to flexibly split the notebook for exploring alternatives. It is particularly designed for the “competitive authoring” collaboration style [181] where team members competitively write code for the same purpose and reach consensus when an acceptable solution is found. This allows collaborators to work independently while making concurrent edits and executions, preventing costly mismatches between programmers’ mental state and the actual state of the runtime. It also provides collaborators with the shared context so they do not work too “far” away from one an-

other, thus supporting awareness of the others' actions (e.g., working on an individual notebook for exploration). Finally, this feature preserves the structure of the narratives by grouping and folding parallel alternatives together.

In the PADLOCK user interface, parallel cell groups are represented as indented cells. Conceptually, this matches the semantic meaning of indentation in Python (specifying the bounds of a code block and potentially creating a new scope), which makes the UI more intuitive and easier to remember for Python programmers.

7.3.4 Implementation

Although the three features of PADLOCK are conceptually related, the implementation of each feature's implementation is substantially different, as detailed below.

7.3.4.1 Cell-Level Access Control

Cell-level access control restricts users' ability to view or edit cells and is thus implemented primarily in the JupyterLab UI. In order to store and sync access control specifications (who can edit or view which cells), PADLOCK stores access control permissions in the metadata for each cell. This information is automatically shared in real-time with every collaborator of the notebook through JupyterLab's Conflict-free Replicated Data Type (CRDT) syncing mechanism. When appropriate, PADLOCK prevents edits by adding a 'read-only' flag to the CodeMirror editor (the underlying Web-based editor used by each JupyterLab cell). When appropriate, PADLOCK prevents viewing a cell by modifying the cell's CSS style to add a "blur" gaussian blur filter to the cell element.

One limitation of our implementation of cell-level access control is that it would not prevent technically savvy bad actors from bypassing the PADLOCK UI (for example, using a browser's debugging panel) and violating the notebook's access control specification. However, we believe it is reasonable to assume that collaborators *intend* to act in the best interest of the larger team and thus would not go out of their way to sabotage the larger project. Otherwise, there would be many other attack vectors to address, such as text spamming or executing computationally expensive code.

7.3.4.2 Variable-Level Access Control

The variable-level access control feature stores and syncs access specifications in the metadata for the notebook (similar to how the cell-level access control specifications are

stored). However, unlike cell-level access control, which is implemented primarily as a UI feature, variable-level access control is implemented primarily as a backend feature. PADLOCK implements variable-level value locks by performing a static analysis on each code cell before it is executed. This static analysis searches the Abstract Syntax Tree (AST) of the code to detect if any restricted variables appear in code and what context they appear in (assignment vs. reference). If the code uses the variable in a way that it should not have access to, PADLOCK prevents the entire cell from executing (before any part of the code runs) and throws an error. If the user has permission to execute the cell, PADLOCK performs a post-execution query to track every variable and its value. This allows every variable to appear in the UI for specifying access control rules. After every cell execution, PADLOCK performs a query to track every variable and its value.

As is the case with cell-level access control, a savvy bad actor could bypass PADLOCK's variable-level access control rules by directly inspecting and executing custom code outside of the JupyterLab UI. However, this is acceptable under the assumption that collaborators are trustworthy in their intentions. Although variable locks are currently implemented through static analyses, future work could consider instead using dynamic analysis to avoid only the portions of code that would actually reference or modify restricted variables.

7.3.4.3 Parallel Cell Groups

Parallel cell groups are implemented by dynamically transforming each cell's code before it is executed. PADLOCK takes a different approach from ForkIt [192], which spawns multiple kernels for its variations. PADLOCK instead executes parallel cell groups within the same kernel, but uses a scoping mechanism that limits the scope of any variables that are declared in the cell. The benefit of PADLOCK's approach is that collaborators can cross-reference variables in other parallel cells (for instance, they can reference the variable `test` in `fragmentA` using `_fragmentA.test`). Cross referencing variables allows users to directly compare different versions of the variables as needed. However, a limitation of PADLOCK's single kernel approach is that computationally expensive code can block the kernel for other users if it takes too long to run.

Through dynamic code execution, PADLOCK converts a parallel cell's code into code that is functionally equivalent but limits the scope of any variables that are declared in the cell. Specifically, PADLOCK defines a `Fragment` class that executes code in a parallel group privately. When a new parallel cell group is created, an instance of `Fragment`

is created and the variables in the main notebook are deep copied through pickle serialization. When a user executes a cell, PADLOCK will send the code content of the cell through the `Fragment` execution function. The execution function runs the code using Python dynamic execution (`exec`). We create an execution scope that merges the global scope with the local scope of variables inside the `Fragment` instance. This allows the execution function to use the cloned version of the variables under the scope of the `Fragment` instance. Lastly, PADLOCK updates variables under the scope of the `Fragment` instance to variables from the dynamic execution.

Finally, there are subtle aspects to how JupyterLab executes cells that PADLOCK handles. For example, JupyterLab typically displays the value of the last expression in the cell as the ‘output’ of the cell. This would be lost in PADLOCK’s rewritten `Fragment` instances so PADLOCK re-routes system IO in a way that matches JupyterLab’s ‘standard’ behavior.

To illustrate on a high-level how this works, consider the following example, where the ‘main’ notebook defines a variable named `foo` and a parallel cell group named `plel` re-assigns `foo` and declares a new variable named `bar`:

```
# main notebook  
foo = 123
```

```
# parallel cell group named 'plel'  
foo = foo + 5  
bar = 'hello'  
foo # note: in "standard" Jupyter, this line outputs '128' when  
↪ the user executes this cell
```

PADLOCK first uses a hidden “cell magic”⁶ command (`%%_privateCell`) to convert the second cell’s text (omitting the original comments in subsequent code samples):

```
%%_privateCell plel  
foo = foo + 5
```

⁶<https://ipython.readthedocs.io/en/stable/interactive/magics.html>


```
bar = 'hello'  
foo
```

The `_privateCell` cell magic function further transforms the code to new code that:

- Creates a new `Fragment` instance for this cell group, if it does not exist
- Copies all the global variables into this `Fragment` instance
- Executes a transformed version of the cell code

This is illustrated in the code below. Note that several function calls (`_copyglobal` and `_execute`) are expanded to demonstrate what they do:

```
# STEP 1. create an instance of the _Fragment class if it does  
↪ not exist  
# =====  
if not '_plel' in dir():  
    _plel = _Fragment('_plel')
```

```
# STEP 2. deep clone the global variables through pickle  
# =====  
_plel._copyglobal() # this method does the functions described  
↪ below (2.1 & 2.2)  
# 2.1. get all the global variables by dir(),  
# the _filter_var function omits variables with some  
↪ prefix (default: '_')  
_global_vars = _filter_var(dir()) # note: this  
↪ happens inside _copyglobal()  
# 2.2. loop through all the global variables  
# _vars = ['foo']  
_plel.foo = pickle.loads(cPickle.dumps(foo, -1)) # note: this  
↪ happens inside _copyglobal()  
# ... repeat for other global variables in  
↪ _global_vars
```

```

# STEP 3. execute the code in local scope
# =====
_plel._execute(''' # note: this example ignores spacing for
↳ clarity
    foo = foo + 5
    bar = 'hello'
    foo
''')
# the _execute method does the functions described below (3.1,
↳ 3.2, & 3.3):
#     3.1. Construct an appropriate scope
_global_scope = globals() # note: this happens inside
↳ _execute
_local_scope = {'foo': _plel.foo } # note: this happens inside
↳ _execute
# use the *cloned* version
↳ of foo
_merged_scope = dict() # note: this happens inside
↳ _execute
# add in global and local variables:
_merged_scope.update(_global_scope) # note: this happens inside
↳ _execute
_merged_scope.update(_local_scope) # note: this happens inside
↳ _execute

#     3.2. Execute a transformed version of the original cell
↳ code
#         within the constructed scope using exec()
exec(code, _merged_scope, _merged_scope) # note: this happens
↳ inside _execute

#         this is equivalent to executing (through exec):
#         _plel.foo = _plel.foo + 5

```

```
#         bar = 'hello' # note that this assigns 'bar' within
→  _merged_scope
#         display(_plel.foo) # A call to display() needed to be
→  added for consistency

#     3.3. Clean up and store variables that were declared.
#         _merged_scope.foo has been updated but _plel.foo has
→  not.
#         (same with .bar) so store results back in _plel
#         _plel.foo = foo # note: this happens inside
→  _execute
#         _plel.bar = bar # note: this happens inside
→  _execute
#         ... merge anything else in _merged_scope
```

7.4 Evaluation Overview

To validate the effectiveness of PADLOCK, we designed a three-stage evaluation—a laboratory study with individual participants working with a paired collaborator, a laboratory study with individual participants planning for various collaboration scenarios, and a case study with groups of participants. In the first stage (paired session), participants joined a laboratory study to work on a structured data science task with designed situations of editing conflicts. After the first stage, participants are optionally allowed to proceed to the second and third stages. In the second stage (planning session), participants were given three hypothetical collaboration setups and work on planning the notebook for these collaborations. In the third stage (group session), participants were paired with each other into groups and worked on open-ended data science tasks. Through this three-stage evaluation, we aim to answer the following question: How do features of PADLOCK assist or hinder real-time collaboration among data scientists?

7.4.1 Participants

We recruited data science students and alumni from data science programs and interest groups in a university. We required participants to have experience with Jupyter and

Python, and preferred participants to have experience using real-time collaborative notebooks. To match the task difficulty, we required participants to be familiar with Pandas, but not necessarily with Regex. As Table 9.1 shows, the paired session contains 14 participants (3 undergraduate students, 9 graduate students, 2 alumni), where all of them have used real-time collaborative notebooks. 8 participants volunteered to join the additional evaluation for planning notebooks for various collaboration scenarios, while 7 of them signed up for the group session. Based on their time availability for the group session, we assigned them to two groups, where G1 has 4 participants and G2 has 3 participants.

7.5 Paired Session: Handling Situations of Editing Conflicts

We first conducted a laboratory study on handling situations of editing conflicts with a paired collaborator. We are interested in observing how participants use the features in a common conflict editing scenario. This conflict editing scenario is synthesized from literature [181] and reproduced by pairing participants with a member of the research team who plays the role of a “clumsy collaborator”.

7.5.1 Study Setup

The study session was conducted remotely through a video-conferencing application. We deployed the extension on a JupyterHub instance to support multiple users accessing the collaborative editing infrastructure of JupyterLab. In addition, we implemented a basic chat interface for the clumsy collaborator to communicate with the participant. We chose to use text chat instead of voice communication because it was easier for the clumsy collaborator to control how they talked to the participants.

Each session lasted 60 minutes. When participants joined the remote meeting, we first showed them how to use the real-time collaboration feature provided by JupyterLab. We then introduced them to the clumsy collaborator and asked them to greet each other through the chat tool. Participants were informed that not all the study procedures would be explained until the end of the study, and we did not reveal the clumsy collaborator being a member of the research team. The clumsy collaborator would then introduce his background as a data science student who knew Python and regex, but was not experienced in Pandas. Next, we explained the task and the dataset. We divided the task into three

sub-goals (noted as T1, T2, and T3), which we will discuss later in the task description. We then asked the participant to work with the clumsy collaborator to solve T1, where the clumsy collaborator will not disturb the participant's work. This is to get participants familiar with the built-in collaborative editing feature and the clumsy collaborator. Next, we asked the participant to solve T2 without the conflict editing feature; the clumsy collaborator would follow a script to disturb the participant's work. We would observe how participants reacted to the unexpected execution results. After T2, we would conduct a debrief to ask participants what went wrong in the previous session and how they handled it. Followed by a live demo on PADLOCK, we asked the participant to solve T3 with the conflict editing feature; the clumsy collaborator would follow the participants' suggestion to use the notebook. Lastly, we would debrief the whole research process, reveal the study setup about the clumsy collaborator, and discuss the conflict editing features with the participants in a reflective interview.

7.5.2 Task Description

The task and the dataset were adapted from a Kaggle challenge to preprocess customer support twitter contents. The reference solution on Kaggle contains several steps, including lower casing, removing twitter user name, removing frequent words, etc. We chose lower casing as T1 for warm-up, removing twitter user name as T2, and removing URL as T3. In the notebook, we also inserted sections on removing punctuation and removing frequent words after T3, with code already implemented.

7.5.3 The Clumsy Collaborator

A member of the research team played the role of clumsy collaborator followed by the following heuristics. First, the collaborator would greet the participant in the chat message at the beginning of the study. For T1, the clumsy collaborator would first ask participants how they want to solve the problem. Then, the clumsy collaborator would tell the participant that she is going to google some API. If the participant got stuck on the task for more than 5 minutes, the collaborator would come back and send a reference code or documentation in the chat (not a direct solution).

For T2, the clumsy collaborator would inform the participant that she would explore the regex in a different cell. Then, if the collaborator got stuck on the task, the collaborator would message an example regex string (not directly how it can be applied to the

Table 7.1: We recruited students and alumni from data science programs in our institution. There are 14 participants who finished the paired session, 8 of them chose to participate in the individual session (S2), and 7 of them chose to participate in the group session (S3). Grads. refers to graduate students; Ugrd. refers to undergraduate students.

PID	S2	S3	Bg.	RTC Exp.	PID	S2	S3	Bg.	RTC Exp.
P1	-	-	Grads.	Deepnote	P8	-	-	Grads.	Deepnote
P2	-	-	Alumni	Google Colab	P9	✓	G2	Grads.	Google Colab
P3	✓	G2	Ugrd.	Deepnote	P10	✓	G1	Ugrd.	Deepnote
P4	-	G1	Grads.	Deepnote; Google Colab	P11	-	-	Grads.	Deepnote
P5	-	-	Grads.	Google Colab	P12	✓	G1	Alumni	Deepnote; Google Colab
P6	✓	-	Grads.	Deepnote	P13	✓	-	Grads.	Collaborative JupyterLab
P7	✓	G1	Grads.	Deepnote	P14	✓	G2	Ugrd.	Deepnote

dataframe) to the participant. Next, the clumsy collaborator would remove the @ symbol at a reasonable time — before the participant executes the code to remove the Twitter username. This operation would interfere with the participant’s action to remove the Twitter username since the @ symbol was no longer in the tweet to indicate the Twitter username.

For T3, the clumsy collaborator would first ask the participant which cell she should work on. Then, the clumsy collaborator would follow the participant’s suggestion to work together. The clumsy collaborator would still pretend to “accidentally” execute the removal punctuation code, which would disturb the regex matching for URLs.

7.5.4 Data Analysis

For each study session, two members of the research team took notes on how participants responded to the clumsy collaborator in T1, T2, and T3. The research team also used screen recording to reflect on the observations. For the reflective interview, one member of the research team took an inductive approach to identify common feedback and representative comments.

7.5.5 Result

7.5.5.1 Conflict editing is hard to notice and prevent

After the second task, most participants (13/14) were not able to correctly find out what caused the code cell not to return the expected results until we explained it to them. This aligned with our observations that many participants (12/14) switched their browsers to search for API documentation and did not stay on the shared notebooks all the time. Moreover, there are several participants (P5, P10) who did not even notice that the output was wrong. There is an exceptional case where P9 ran the data loading cell right before executing the cell for removing the twitter username, leaving no chance for the clumsy collaborator to modify the shared variable. P9 noted,

Yes, I do prefer to reload the data every time before running a new cell, unless the data frame is very large, in which case it takes a lot of time to load. So then I would avoid doing it, but otherwise yes, I do.

Interestingly, although several participants (4/14) were able to recover from the issue by reloading the dataframe, they still did not find the source of the problem. The majority of participants (12/14) did not doubt their collaborators' actions or question what they did. Instead, they blamed themselves and looked into their own code to debug. For example, P3 said:

I am familiar with Jupyter Notebook, but I just don't have the confidence... I felt like I had the correct code. But I assumed something was wrong with it. I just didn't even think that it could have been the collaborator's code (that causes the issue).

7.5.5.2 Perceptions of PADLOCK for preventing conflict editing

After debriefing the second task and walking through the three features of PADLOCK, participants were asked to finish the third task with the clumsy collaborator. All participants (14/14) chose to create parallel cell groups and suggested the clumsy collaborator to write their code in a parallel cell. After they finished the task, some participants (8/14) cleaned up the notebook by unindenting the parallel cell groups. Several participants (2/14) chose to keep the clumsy collaborator's parallel cell and merge their solution into the notebook by marking their solution as main.

Overall, participants reported that they felt confident about not messing up with the shared notebook. For example, P12 reported:

The parallel cells are very useful. In the case of removing punctuations and removing twitter username, as long as I check the value of `df` when I start the indent, that would be okay.

Participants also mentioned that the parallel cell groups made the shared notebook “neat” (P4), “organized” (P11), and “structured” (P6).

Although participants did not use the cell-level access control and variable-level access control, they described scenarios where these features could be useful. P10 mentioned that both features could be helpful in the large classroom setting, and she recalled an experience of messing up shared notebooks:

I definitely think the cell-level access control and variable-level access control can be useful in a classroom setting where maybe you have an instructor with a sample notebook. Maybe they’d want to obscure and not have you be able to read like a possible solution that they have, or be able to accidentally edit and screw up some steps that they had put in just to show everyone. Because I remember that happened a couple of times in my class. Last semester students would sometimes accidentally edit the wrong thing, and then the professor would have to backtrack and just make sure his starter code was fixed before we could continue...

P5 said that variable-level access control can be useful when the cost of restarting the kernel and running previous code cells is expensive. He described the scenario where a data science manager would not want interns to accidentally modify large-scale data tables and had to restart the kernel to recover the results. In addition, P10 mentioned that she would use the cell-level access control on finished code cells, and use parallel cell groups on work-in-progress cells. Noticeably, several participants mentioned that read access in cell access control was not necessary for themselves, but they could see it being used by other people. For example, P4 said:

For blurring the cells, some of my friends are shy so I could see that this would be very useful for them. But I personally would not use it. I think being able to see what your collaborator is doing is a part of that collaboration experience.

7.5.5.3 Improvement of the Parallel Cell

Participants shared several ideas on improving the parallel cell feature in PADLOCK. Several participants (P6, P9) mentioned adding notifications or activity histories to track if others have unindented a code cell. P9 illustrated this need by comparing the experience with git:

When you merge the selected tab with the main thread, that's like a commit to the main repository in GitHub. So then, you know, you need to also tell others that I have launched this, maybe a notification. I was hoping there would be some way to track that, like GitHub provides a history of commits that somebody has made to other changes.

In addition, P2 asked for a merging process where she could pull cells from various fragments:

I wish there is an option to maybe merge different parts of the cell, like maybe one collaborator has one cell, and then you merge the second part of another collaborator.

7.5.5.4 Resonate with prior experience

The instance of editing conflict in task 2 resonated with participants' prior experience with real-time collaborative editing. P1 mentioned a different collaborative setting in a data science classroom. The data science classroom had around 100 students and the instructor asked everyone to join the same notebook in Deepnote. However, the instructor asked students to not directly run code cells in the notebook. Instead, students typed out solutions and commented at the same time. Several participants (P9, P13) mentioned that their prior experience with shared notebooks was mostly asynchronous collaborating. To further understand how PADLOCK may improve the issues that participants mentioned in their prior experience, we conducted an additional study where we asked participants to plan for a future collaboration scenario, as discussed in the next section.

7.6 Planning Session: Planning for Various Collaboration Scenarios

Noticeably, none of the participants used the cell-level access control and variable-level access control features when working with the clumsy collaborator in the paired session, although they have mentioned the potential of using these features in other collaboration setups. To better understand the usefulness of PADLOCK under various collaboration scenarios, we conducted an additional evaluation where we asked participants to plan a future collaboration session by configuring a collaborative notebook. This additional evaluation allowed us to explore the usefulness of the PADLOCK features in different collaboration scenarios. By asking participants to configure a collaborative notebook for planning a future collaboration session, we were able to see how they would use the collaborative features in a more proactive manner. This can provide valuable insights into how users might use the features of PADLOCK in a variety of collaborative environments.

Overall, our additional evaluation showed that the PADLOCK features can be useful in various collaboration scenarios, including working on an asynchronous paired programming session, working on tasks with high cost of error recovery, and using collaborative notebooks as lecture notes. By providing a way to prevent conflicts and improve organization, PADLOCK can help make collaboration more efficient and effective.

7.6.1 Study Setup

In the individual study sessions, we used the same deployment as in the paired sessions. Based on participants' responses from the previous study, where they mentioned their prior or future use of collaborative notebooks, we synthesized three collaboration scenarios designed to be representative of various realistic situations. For each scenario, we gave participants an initial notebook containing skeleton code and a written description of the collaboration scenario. We asked participants to plan for the collaboration by leveraging the features of PADLOCK and modifying the notebook content as necessary. When they finished planning, we asked participants to verbally describe their plans to the study coordinators, who would then ask a set of semi-structured interview questions to gain further insight into the participants' plans and the features they used or did not use, as well as any potential problems that might arise during the collaboration and any additional features that they felt would be useful. Participants were given as much time as they needed to think about their plans, with most individual sessions lasting around 30 minutes.

7.6.2 Collaboration Scenarios

In the individual session, participants were asked to plan for the following three collaboration scenarios:

7.6.2.1 Scenario 1: Asynchronous Collaboration with a Peer

For the first scenario, participants are asked to plan a paired programming session with a friend named Bob who is inexperienced with libraries like Pandas and Numpy, similar to the clumsy collaborator case in the previous study. However, due to conflicting schedules, Bob and the participant are unable to find a synchronous time to work together. This means that both of them may join and leave the session with incomplete code, and the participant cannot guarantee how Bob will use the collaborative editing features without being able to observe and intervene in Bob's actions.

7.6.2.2 Scenario 2: Working with Trainees and a High Cost of Error Recovery

In this scenario, we asked participants to plan a collaboration session as a full-time employee distributing tasks to interns on visualizing different aspects of the data. We provided participants with a notebook skeleton where the data takes a long time to load. In this scenario, participants needed to avoid the shared dataframe being polluted by any accidental changes, which would be costly to recover from. To do this, they could leverage the features of the PADLOCK system to ensure that their collaboration was organized and efficient, and that the data was protected from any unintended changes. This scenario simulates the case where error recovery could be costly in a collaboration setting.

7.6.2.3 Scenario 3: Classroom Sharing with Hierarchical Permissions

In this scenario, we provided participants with an educational notebook on the topic of linear regression, taken from a data science handbook. The first half of the notebook demonstrated the concepts, while the second half contained an exercise for students to practice what they learned. Participants were asked to plan the use of the notebook as an instructor, taking into account the needs of the entire class during a lecture. They needed to ensure that they could effectively explain the concepts to the students, while also providing them with an opportunity to practice individually on the exercise. Additionally, the exercise included a standard solution that the instructor may want to go over with the class after they have explored their own solutions. Participants were asked to plan how

they would like to set up the collaborative notebook at the beginning of the lecture, and how they would modify the configurations as the lecture progressed.

7.6.3 Data Analysis

In each of the above scenarios, participants were asked to configure a shared notebook. We used three metrics to understand and evaluate the effectiveness of the participants' specified notebook configurations. First, we summarize the patterns in how participants set up collaborative notebooks for each scenario. The second data source was a list of potential editing conflicts that may occur in each scenario, which was identified by the research team. We used this list to test participants' configurations against the potential conflicts and analyze their reliability in addressing them. Lastly, we recorded and transcribed participants' post-task reflections, in which they discussed their choices, potential problems, and suggestions for additional features. These data sources gave us a comprehensive view of the effectiveness of the configuration in different collaboration scenarios.

7.6.4 Results

7.6.4.1 Usage of the Collaborative Features for Each Scenario

In Table 7.2, we summarize the strategies that participants described for each scenario and listed the features of PADLOCK that are involved. For the first scenario of working in an asynchronous paired programming session, three participants chose to ask the collaborator to use the parallel cells for exploration, while the other participants chose to use cell-level or variable-level access controls. P12 explained why their strategy changed compared to the clumsy collaborator scenario in the previous paired session:

I can't really trust that Bob is going to use parallel cells because we are not working together at the same time. I want to set up everything for him so he can only access the things he need[s].

For the second scenario, most participants except P10 chose to lock the shared variable to prevent it from being polluted. Participants suggested that the interns could use parallel cells or make a copy of the dataframe if needed. In addition, most participants (6 out of 8) chose to restrict the editing access of the code cell for loading the data, while two of them also chose to set up the editing access for the code cells that are assigned to each individual intern.

Session	Action or Strategy	Feature	PID
Paired	Nudge the clumsy collaborator to create parallel cells	Parallel Cell	P1-14
Paired	Merge parallel cells	Parallel Cell	P1, P2, P3, P5, P9, P11, P12, P14
Planning (1)	Lock cells for loading package and data	Cell-Level Access Control	P3, P7, P10, P12, P13, P14
Planning (1)	Lock future cells	Cell-Level Access Control	P6, P10
Planning (1)	Lock the original or a copied dataframe	Variable-Level Access Control	P3, P10, P12, P13, P14
Planning (1)	Create a copy of the dataframe	–	P3, P7, P12
Planning (1)	Ask Bob to use parallel cells	Parallel Cell	P6, P9, P13
Planning (2)	Lock cells for loading data	Cell-Level Access Control	P3, P6, P10, P12, P13, P14
Planning (2)	Change cells' edit access so that interns cannot change each others' code	Cell-Level Access Control	P9, P12
Planning (2)	Lock the shared dataframe	Variable-Level Access Control	P3, P6, P7, P10, P12, P13, P14
Planning (2)	Ask the interns to use the parallel cells	Parallel Cell	P3, P6, P9
Planning (2)	Create a copy of the dataframe if needed	–	P7, P10
Planning (3)	Lock editing access for code cells in lecture notes	Cell-Level Access Control	P3, P6, P7, P9, P10, P12, P13, P14
Planning (3)	Lock reading access for code cells in lecture notes that are not covered yet	Cell-Level Access Control	P6, P10
Planning (3)	Lock reading access for the standard solution code cell in the exercise	Cell-Level Access Control	P3, P6, P9, P10, P12, P14
Planning (3)	Lock access for variables generated in lecture notes	Variable-Level Access Control	P13
Planning (3)	Lock access for the variable for storing results in the exercise	Variable-Level Access Control	P14
Planning (3)	Create parallel cells for students to work on the exercise	Parallel Cell	P3, P7, P9, P10, P12, P13, P14
Planning (3)	Ask students to use parallel cells if they want to explore the code (e.g., change parameters) in lecture notes	Parallel Cell	P9, P12
Group	Create parallel cells	Parallel Cell	P3, P4, P7, P9, P10, P14
Group	Merge parallel cells	Parallel Cell	P4, P7, P9
Group	Sync parallel cells	Parallel Cell	P3

Table 7.2: Features that participants have used for tasks in each study.

In the last scenario, participants described a mixed strategy for planning the collaborative notebook for a data science classroom. All participants decided to turn off the editing access for code cells in lecture notes, as P14 explained:

I would turn off the cell editing for the class. Otherwise, if there's so many students, it is easy for someone to accidentally hit a backspace somewhere or something and mess things up.

Some participants (P9 and P12) mentioned that they would create a copy of the cell below each code cell for lecture notes and make them into parallel cells, in case students want to explore the code cells in lecture notes. For the same consideration, P13 wanted to restrict the access for variables generated in lecture notes, in case students modify the shared runtime in the lecture. Other participants did not worry about protecting the shared variables, as one participant explained (P6):

Since the content of the code cell is locked, I can always restart the kernel and run the notebook from the beginning if anything goes wrong.

In addition, two participants (P6 and P10) mentioned that they would like to change the code cells' reading access as the lecture progresses so that students can stay focused. For the exercise part, most participants planned to hide the reference solution code (6 out of 8) and ask students to work on parallel cells for their own practice (7 out of 8).

7.6.4.2 Effectiveness of the Collaboration Plan

For each scenario, we solicited three potential conflicts that may arise during collaboration: two that we expect to be common and one that is less likely to occur. We then evaluated each participant's use of the collaborative notebook to determine if their configuration could effectively address these conflicts. Two members of the research team carefully calibrated and discussed the ratings until they reached a consensus. The results, shown in Table 7.3, indicate that most participants (more than 6 out of 8) were able to utilize the features in PADLOCK to successfully address common collaboration issues. Even for the less likely problems (e.g., Bob missing the instruction and starting work on the last code cell in the first scenario), a small number of participants (1–3) were still able to successfully handle these rare cases.

Scenario	Editing Conflicts	Category	Pass Rate
Scenario 1	Bob dropped all the NA values in the dataset.	Common	0.75
Scenario 1	Bob changed the importing package cell to only include a subset of a package.	Common	0.75
Scenario 1	Bob missed the instruction and started to work on the last code cell.	Rare	0.25
Scenario 2	Intern A changed the shared dataframe.	Common	0.88
Scenario 2	Intern A run the code cell for loading the data frame.	Common	0.88
Scenario 2	Intern A and intern B have the same naming of a variable.	Rare	0.38
Scenario 3	A student edited the code cells for demonstration.	Common	1
Scenario 3	Students directly assigned the prediction results to the shared data frame.	Common	0.88
Scenario 3	Students executed a code cell multiple times.	Rare	0.13

Table 7.3: For each scenario, the research team solicited three potential cases for editing conflicts and run through participants’ notebooks through these cases.

7.6.4.3 Improving Access Control

In the reflective interview, participants mentioned several potential problems and suggested improvements for the collaboration scenarios in the current system design.

First, participants brought up the need for better access control on the notebook level. For example, P13 mentioned that restarting or interrupting the notebook kernel in the third collaboration scenario could cause problems when the instructor is demonstrating concepts. Other types of access control mentioned by participants included preventing collaborators from executing a code cell (P6) or copying and pasting the content from a code cell (P13).

Participants also suggested ways to improve the process of configuring access control. For example, in the first scenario, P6 and P10 mentioned that they would like to restrict cell edit access for their collaborator for every new code cell that they create under a section. One participant also suggested adding a “run and lock all cells and variables above” button (P13) to avoid the need to manually lock all code cells in the lecture notes in the third scenario.

Lastly, participants (P9 and P12) mentioned the potential benefits of combining cell access control with parallel cells. This would be particularly useful in the third scenario, where the instructor may not want students to see each other’s solutions in the parallel

cell.

7.7 Group Session: Case Study on Open-Ended Collaboration

The paired session and the individual session demonstrated the usefulness of PADLOCK in designated collaboration setups. To further understand how PADLOCK would be used in open-ended and less-structured collaboration tasks, we conducted two case studies by observing participants as they worked together in groups on a collaborative task.

Overall, our case studies showed that PADLOCK can be useful in open-ended and less-structured collaboration tasks. By providing a framework for organizing the collaboration, it can help them divide up the work and ensure that each person is working on a specific part of the task. This can help make the collaboration more efficient and reduce the risk of overlap or duplication of effort. Additionally, PADLOCK can help prevent conflicts between collaborators, save time and reduce frustration, leading to a more productive collaboration.

7.7.1 Study Setup

The group study sessions were also conducted remotely, and used the same deployment as the individual sessions. In the group sessions, participants communicated by talking to each other through the video-conferencing application.

As all participants had already used the extension in individual sessions, we started with a brief reminder about its features. Then, one researcher would explain the collaborative task to the participants, and ask participants to introduce themselves to their collaborators. Next, we explained the open-ended tasks and dataset. In the first group (4 participants), we assigned the participant who is most experienced in data science as the team leader to mitigate collaboration during the task. We did not assign a group leader in group two, which only had 3 participants. Each session lasted 60 minutes in total, and participants were given 40 minutes to complete the task. We would remind the participants to clean up the notebook to provide final results when there were 5 minutes left. Finally, we asked each participant to fill out a post-task survey regarding their collaboration experience.

7.7.2 Task Description

In the group task, participants were asked to conduct an exploratory analysis of papers accepted by CHI 2022. We provided the participants with two datasets: the first one contains the paper name, link, and author list of full papers; the second one contains the name, link, and affiliations of authors. To make the task more authentic, we collected the raw data by scraping the actual conference program from 2022. Participants were given a list of open-ended tasks regarding ranking authors and institutions by the number of full papers they published. To complete the task, they would need to conduct a series of data pre-processing and cross-referencing between two datasets, and the process is highly open-ended. For example, the format for authors' names in two datasets are different; authors from different institutions might have the same name; an author may have multiple institutions; or even the name of the same institution can be put differently in the system. We made participants aware of the complication in the dataset, but did not provide detailed instructions to encourage open-ended exploration.

7.7.3 Data Analysis

During each group session, two members of the research team closely monitored collaboration activities and took detailed observational notes. We also recorded participants' screens and captured key activity logs from the Jupyter Notebook to help us understand the collaborative workflow of each group. In addition, we conducted a second pass on the video recordings to identify instances of collaboration challenges and undesired behaviors. This allowed us to carefully analyze the collaboration process and identify areas for improvement.

7.7.4 Result

Overall, we observed different collaboration patterns among the two groups. We report how the two groups leveraged PADLOCK to support their workflow.

7.7.4.1 G1: Starting from Pair Authoring

We observed that participants in G1 used a mixed of collaboration styles, including pair authoring, divide and conquer, and competitive authoring. With a group leader moderating the discussions, the team naturally started the task by a pair authoring mode. Figure

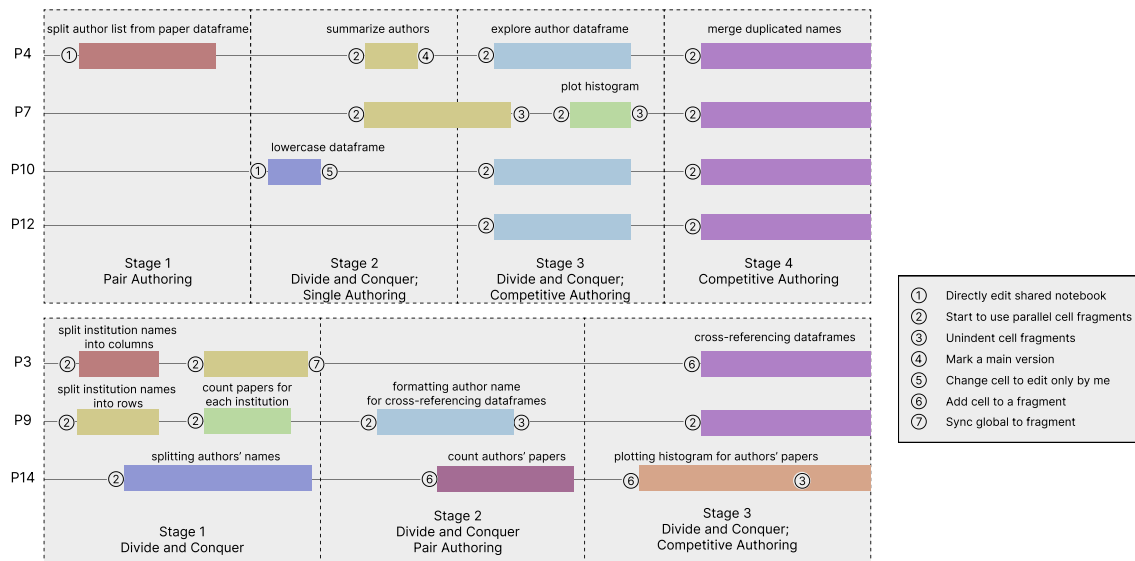


Figure 7.3: The collaborative workflow for G1 and G2.

7.3 illustrates the collaborative workflow of G1. We broke down their exploration process into four stages.

- Stage 1:** To begin with, the four participants (P4, P7, P10, P12) discussed the task and decided to split paper authors first. Then, P4 proposed to begin work and they used a pair authoring mode where P4 directly wrote code in the notebook. The rest of the team closely watched him coding and helped him with debugging.
- Stage 2:** Next, the team leader P12 proposed two tasks – lower casing the authors' names and summarizing authors' paper counts. The team then split into two groups where P10 worked on lower casing, P4 and P7 worked on summarizing the counts, and P12 observed the process. P10 quickly implemented lower casing by directly working on the code cell while the rest of the team is still discussing how to summarize the counts. P10 then turned on the cell editing lock to her finished code cells. In the meanwhile, P4 and P7 created two parallel cell groups and started to work on summarizing the counts in competitive authoring style. After P4 figured out the correct solution, he reported to P12 and they decided to merge his solution into the main notebook. However, noticing that P7 was still working on his solution, P4 chose to mark his cell group as the main version in the parallel groups and moved on to the next task with P12.
- Stage 3:** Moving on to the next stage, P4, P10, and P12 started to examine the

author—affiliations table using parallel cell groups. Then, P7 finished his implementation of summarizing author counts and reported to the team. P7 proposed to unindent the parallel cell group to keep P4’s solution. After they merged the cells for summarizing authors count, P7 proposed to branch off and work on creating a histogram visualization of the authors’ paper count while the rest of the team kept exploring the author—affiliations table. After P7 finished the histogram visualization in a parallel cell, he notified the rest of the team and merged his exploration code.

- **Stage 4:** Lastly, all the team members worked together on merging duplicates in institution names since they realized that there was not much time left. They created a parallel cell group with four tabs where everyone worked on their own tab. They were not able to finish this task towards the end of the study.

7.7.4.2 G2: Starting from Divide and Conquer

G2 started with the divide and conquer style where the three members decided to work on three different tasks — authors’ paper count, institutions’ paper count, and institutions’ paper count without duplicates. The team also discussed and agreed on using parallel cell groups whenever they started to work, and merging them to the main notebook after they verified the results. P3 proposed to lock the shared variables as well. The team decided to stick with the protocol of using parallel cell groups because they agreed that “If we are all working in indented cells, we don’t need to lock the dataframes” (P3). We broke down their exploration process into three stages:

- **Stage 1:** The three participants started by indenting three cells and working individually on the three tasks. However, the tasks for P3 and P9 both required splitting institution lists. P3 and P9 took two different approaches for the splitting, where P3 splitting the institutions into columns and P9 splitting the institutions into rows. P9 realized it and started a discussion with P3. In the meanwhile, P14 was working individually on splitting the list of authors for each paper.
- **Stage 2:** In this stage, P3 and P9 turned into pair authoring mode after realizing that their tasks need the same pre-processing. After the discussion, P3 was convinced to split the institution list into rows and she attempted to do it in her previous cell. After she finished, she continued to discuss with P9 and observed P9 coding. P9 first attempted to count institutions’ papers. After realizing the need to cross reference

two tables, P9 and P3 turned to work on formatting authors' name in the author-affiliations table. At this stage, P9 did most of the programming and P3 actively participated in discussions. On the other hand, P14 continued to work individually on counting authors' paper numbers.

- **Stage 3:** In this stage, P3 and P9 decided to both try to create a cross-reference between the two tables. P3 asked P9 to merge his previous code into the main notebook, and performed a sync to update her parallel cell group. Then, they began to work competitively on cross reference. Meanwhile, P14 continued to work individually on creating a histogram visualization of the authors' papers.

7.7.4.3 Reflections on the Collaborative Session

By observing two groups attempting the task with different collaboration strategies, we found that PADLOCK was stable and effective in preventing conflict edits while being flexible to support various ad-hoc combinations of collaboration models. We did not observe instances of the abused copying of data frames. We also did not observe instances where participants messed up with the shared notebook and needed to restart the kernel. In the reflection questionnaire, participants commented that “it’s a very nice tool for collaboration” (P7); “it helps a lot when we are exploring the dataset and trying to test some functions.” (P10). In particular, P9 mentioned:

This tool is very helpful. It allows us to split the tasks amongst the team, work independently without having to worry about interference, but still be able to discuss problems with each other’s solutions if needed.

In addition, participants also reported things that did not work well in the collaboration session. For example, P14 in G2 who spent most of the session working independently on the authors' paper count complained that the pre-processing approach in her task (e.g., split author list) is similar to her collaborators' tasks (e.g., split institution list). But they did not collaboratively work on the pre-processing:

It helped us avoid running cell blocks that would influence each other’s work and allowed us to work on our own parts. However, it also made us communicate less and made us focus on our own parts when a lot of the questions could’ve been solved with the majority of the data cleaning work being the same.

7.8 Discussion and Future Work

7.8.1 Lightweight Collaboration Support for Data Science

Compared to collaboration in traditional software engineering, collaborative data science is more exploratory, ad-hoc, and volatile. Thus, data scientists benefit from real-time collaboration to quickly exchange ideas and plan the next step. Real-time collaborative editing improves data science teamwork by creating a shared context, encouraging more explanation, and reducing communication costs [181]. Although data scientists benefit from writing code that is exploratory, experimental, and messy, they need to be careful about their programming practices when working on a shared notebook. This limits data scientists from harnessing the advantage of computational notebooks as they have to manage the invisible state of the shared kernel through careful practices. We believe that data science collaboration benefits from lightweight and ad-hoc support in computational notebooks. Our evaluation of PADLOCK shows that such a design can effectively help data science teams avoid both implicit and explicit editing conflicts.

7.8.2 From Small Groups to Collaboration at Scale

Collaborative editing in computational notebooks can benefit not just small team collaboration, but also collaboration at scale. For example, instructors can share a collaborative notebook with a classroom of students; researchers can share a collaborative notebook with a broader audience for open collaboration; data science hobbyists can make their live streaming session more engaged by sharing the collaborative notebook session. Our work suggests exciting design opportunities for supporting collaborative editing at scale. For example, our current design of the parallel cell would horizontally display the parallel tabs as the number grows. Future work can use mechanisms like searching, tagging, and filtering for managing multiple parallel cells. Our current design of cross-referencing allows participants to computationally compare versions of variables from different parallel cells, which could be improved by integrating visualization techniques to compare data changes [179], or clustering techniques to explore variance [62]. Additionally, we are interested in incorporating domain-specific features and needs for large-scale collaborative editing, such as allowing students to test their code cells with shared test cases provided by peers [177].

7.8.3 Blending Sync and Async Collaboration in Long Terms

One area of interest in our future work on PADLOCK is exploring how the system might be adapted to support different levels of synchronicity in collaborative editing. For example, in asynchronous collaboration, collaborators may not work on the same shared kernel even with a shared notebook. In this case, cell access control may still work to protect the ownership of the code cell; parallel cell groups may be used to improve the readability of the notebook and make it more structured. However, managing access control in a hybrid synchronous-asynchronous collaboration presents unique challenges, such as deciding who should be in charge of setting and enforcing access rules. The current design of PADLOCK allows any collaborators to change the cell-level and variable-level access control. It remains to be explored who should be in charge of managing the access control. Lastly, it is worth exploring how PADLOCK would affect the presentation of the shared notebook. In a hybrid synchronous and asynchronous collaboration, collaborators may leave staled configuration over a code cell or shared variable, making it difficult for others to understand the context or history of the notebook.

7.8.4 Improving Awareness of Collaborators' Activities

PADLOCK focused on the perspective of editing conflicts in real-time collaborative computational notebooks. We believe that effective collaboration can also benefit from combining conflict-free mechanisms with awareness design. For example, the use of parallel cell groups allows multiple users to work on different versions of the same document concurrently, but makes it difficult for users to see each other's cursor movements or edits. By highlighting the active tab for each collaborator, PADLOCK can improve awareness and help users understand who is working on what. Similarly, notifications can alert users when others make changes that affect their work, such as unindenting a cell group or removing tabs. In addition, the design of PADLOCK brings up the unique challenges in helping collaborators track and forage editing history. With the notebook structure being not linear anymore, it is worth exploring how notebook history foraging designs [88] can be extended to support the awareness of complex cell editing.

7.8.5 Limitations

7.8.5.1 Limitations of the Evaluation

It is worth noting that the results of the evaluation may not be generalizable to all collaboration scenarios. The specific tasks in the three study sessions of working with a clumsy collaborator, configuring a collaborative notebook for planning a future collaboration session, and working in small groups to solve an exploratory analysis task together, may not be representative of all collaboration scenarios, and further research would be needed to explore the usefulness of PADLOCK in a wider range of collaboration contexts. Additionally, the results of this evaluation may have been influenced by other factors, such as the participants' individual preferences and experiences with collaborative tools. As such, it is important to interpret the results of this evaluation with these limitations in mind.

7.8.5.2 Limitations of PADLOCK

While PADLOCK is designed to address editing conflicts and support flexible explorations between small-size teams during synchronous editing sessions, it may be limited in its ability to support the needs of large-scale and long-term collaborations. For example, PADLOCK may not provide the necessary tools and features for comparing variance across multiple parallel cells, or maintaining the readability of notebooks and keeping the access controls updated over long periods of use. However, we are interested in understanding how features of PADLOCK are effective for large-scale and long-term usage, as well as ways to extend and adapt PADLOCK to better support these types of collaborative work in the future.

7.9 Conclusion

Real-time collaborative editing in computational notebooks requires strategic coordination between collaborators. We investigated common obstacles in real-time notebook editing and proposed a set of access control mechanisms to support conflict-free editing: cell-level access control (which restricts collaborators' ability to see or edit cells), variable-level access control (which protects runtime variables from being referenced or modified, and parallel cell groups (which allow collaborators to work in their own space while staying connected to the larger notebook). As we found in our user studies with PADLOCK, these features can improve collaboration within data science teams.

Part 3

Facilitate Knowledge Sharing for Future Data Scientists

CHAPTER 8

PuzzleMe: Leveraging Peer Assessment for In-Class Programming Exercises

Peer assessment, as a form of collaborative learning, can engage students in active learning and improve their learning gains. However, current teaching platforms and programming environments provide little support to integrate peer assessment for in-class programming exercises. We identified challenges in conducting such exercises and adopting peer assessment through formative interviews with instructors of introductory programming courses. To address these challenges, we introduce PuzzleMe, a tool to help Computer Science instructors to conduct engaging in-class programming exercises. PuzzleMe leverages peer assessment to support a collaboration model where students provide timely feedback on their peers' work. We propose two assessment techniques tailored to in-class programming exercises: live peer testing and live peer code review. Live peer testing can improve students' code robustness by allowing them to create and share lightweight tests with peers. Live peer code review can improve code understanding by intelligently grouping students to maximize meaningful code reviews. A two-week deployment study revealed that PuzzleMe encourages students to write useful test cases, identify code problems, correct misunderstandings, and learn a diverse set of problem-solving approaches from peers.

8.1 Introduction

Collaborative learning actively engages students to work together to learn new concepts, solve problems, and provide feedback [166]. Programming instructors often use various collaborative learning activities in teaching, such as group discussion, project-based work [115], pair programming [143], code debugging [69], and peer assessment [167]. In par-

ticular, peer assessment through reviewing and testing each other’s solutions can improve students’ motivation, engagement, and learning gains [167, 152, 105, 106], while reducing the effort required for instructors to provide scalable personalized feedback [106].

Despite the benefits of peer assessment, current programming and teaching environments provide little support to conduct peer assessment for **in-class programming exercises**—small scale programming exercises for students to practice during lectures or labs. As a result, peer assessment is typically conducted asynchronously rather than in a live classroom setting [167]. Prior research has made it easier for instructors to share and monitor code with multiple students in real time [31, 72]. However, designing real-time systems to enable both student-student interactions and student-instructor interactions in a live setting is still a challenge [33]. Moreover, students often struggle to give each other high-quality feedback or even start a fruitful conversation without proper moderation and effective grouping [29, 121]. In a needs analysis, we also found that it is difficult for instructors to effectively break students up into groups with appropriate balances of expertise in physical classroom situations. Further, because of the overall lack of expertise, peers can find it difficult to assess whether a given piece of code would fail unknown edge cases even if it generates the desired output for the test cases given by instructors.

In this paper, we present PuzzleMe, a web-based in-class programming exercise tool to address the challenges of peer assessment. PuzzleMe consists of two mechanisms: **live peer testing** and **live peer code review**. Live peer testing helps learners assess their code through moderated collection of test cases from peers. Inspired by the notion of the “sweep” [140], live peer testing seeks essential examples only for illustrating common and interesting behaviors rather than writing comprehensive test suites. PuzzleMe automatically verifies valid test cases by referencing an instructor-provided solution and shares valid test cases with the whole class. Live peer code review aims to provide personalized feedback at scale. It does this by automatically placing students in groups where they can discuss and review each other’s code. PuzzleMe introduces several features to encourage meaningful code review, including a matching mechanism to balance student groups based on the number of correct answers and the diversity of those answers. PuzzleMe also includes mechanisms that allow instructors to create improvised in-class programming exercises, monitor students’ progress, and guide them through solutions. Our design is inspired by formative interviews where we investigated the obstacles instructors face when conducting in-class programming exercises and encouraging peer activities.

To validate PuzzleMe’s effectiveness, we deployed it to an introductory programming

course for two weeks and conducted several exploratory studies. Our results show that the peer testing feature can motivate students to write more high-quality tests, help identify potential errors in their code, and gain confidence in their solutions. Further, the peer code review feature can help students correct misunderstandings of the course materials, understand alternative solutions, and improve their coding style. We also report on the use of PuzzleMe in an online lecture¹ and demonstrate its potential to be used at scale in synchronous online education. We found that PuzzleMe is perceived to be useful in a wide range of programming classes, reducing the stress of providing near-immediate feedback, helping instructors to engage students, and providing opportunities to explore different types of pedagogy.

The key contribution of this work is the design lessons learned from a series of mixed methods studies, which add to the body of work on personalized feedback, peer assessment, and in-class exercises. We believe these lessons can guide future interface design exploration in similar contexts (e.g., live workshops and programming education via live streaming [33]). shows the potential for increasing learning outcomes via in-class peer support without increasing teaching costs. Specifically, our contribution includes:

1. an articulation of the needs and challenges that instructors have when conducting in-class exercises for introductory programming courses based on formative interviews with five instructors,
2. two techniques—live peer testing and live peer code review—that enable peer assessment during in-class exercises, and
3. PuzzleMe, a web-based system for instructors to carry out in-class programming exercises with the support of live peer testing and live peer code review.

8.2 In-class Programming Exercise Challenges

To better understand the current practices and challenges for conducting in-class programming exercises, we conducted formative interviews with instructors of introductory programming courses. We chose to focus on introductory programming courses because (1) they typically have larger enrollments, (2) students in introductory courses often need more support, and (3) large knowledge gaps between students are more likely, making it difficult for instructors to accommodate all students' needs.

¹During our deployment, this course migrated to a fully online setting due to the outbreak of COVID-19.

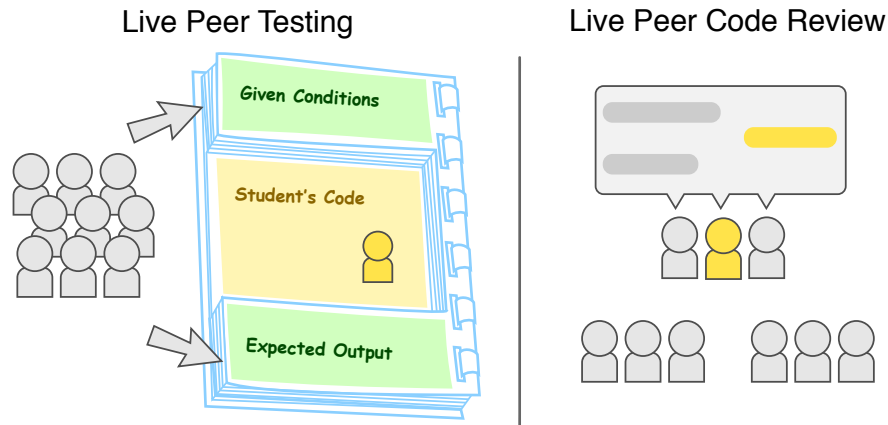


Figure 8.1: PuzzleMe implements two mechanisms for peer assessment: live peer testing and live peer code review. Live peer testing allows students to create and share test cases with peers in real time. Live peer code review intelligently groups students to encourage meaningful code discussions.

8.2.1 Method

We recruited five instructors from different introductory programming courses at the authors' university. Table 8.1 presents an overview of the courses that the five interviewees taught. They include three undergraduate- and two graduate-level courses across three departments, with the same session time (80 minutes, twice per week). Each interview lasted 30 minutes. We asked instructors to recall the most recent introductory programming courses they had taught and explain what types of in-class exercises they conducted and what processes and tools were involved to facilitate the exercises. In addition, instructors were encouraged to tell us about any challenges they had encountered with conducting in-class programming exercises. Two authors separately conducted iterative coding to identify reoccurring themes using inductive analysis. We then merged similar codes to infer important findings. During the process, the codes of common practices for conducting in-class exercises, such as the types of exercises and tools, were merged smoothly. However, those of challenges in conducting programming exercises were rather difficult because of the authors' different perspectives (e.g., process vs. roles involved). By dividing the in-class exercise activity into different stages and identifying the major roles of each party, the authors discussed and finalized the codes.

Table 8.1: Instructors' course demographics in formative interviews.

PID	Size	Name	Language(s)
S1	260	Intro. to Prog. I (G)	Python
S2	250	Intro. to Engr. (U)	MATLAB
S2,S3	300	Intro. to Prog. II (U)	Python
S4	260	Intro. to UI Prog. (G)	HTML,CSS,JS
S5	260	Intro. to Prog. I (U)	Python

8.2.2 Findings

8.2.2.1 In-class exercises are common

On average, interviewees spent a third of the lecture time on programming exercises. Interviewees often conducted two types of in-class exercises: multiple-choice questions (where students respond using audience response systems such as i-clickers) and programming exercises (where students write code on their laptops). In some cases, students needed to download starter code from code collaboration applications [201, 65]. None of the interviewees mentioned using assessment tools to collect and validate students' solutions. Instead, they often provided an example of an acceptable solution (e.g., on a projector) and asked students to self-check their code. All interviewees mentioned asking students to discuss with their peers or providing personalized help during office hours.

8.2.2.2 Impromptu exercises are valuable

To conduct effective exercises, instructors get feedback from students on what they understand and then improvise exercises based on that feedback. Some instructors would live code in class and call on students to verbally describe what code they should write (S2, S5). They encountered cases of *“a lot of people making similar misconceptions that I did not expect”* (S1), so they would often choose to let students vocally explain them to the rest of the class (S1, S4). Although vocal feedback has a low overhead cost, interviewees were concerned that *“you always get the same people participating”* (S4). S1 wished to leave more lecture time for students to *“share their thoughts”*, or *“learn from the person sitting 20 feet away.”* Additionally, vocal communication is often not accessible (e.g., hard to hear) for students and is not archived for students to revisit.

8.2.2.3 Hard to scale support for exercises

When conducting the exercises, all interviewees reported that they would walk around the classroom to observe students' progress (S1–S5), provide help (S1–S5), “*get teaching feedback*” by asking students “*what’s hard about this*” (S1), or “*hearing about low level problems that we might not have thought about*” (S4). Four interviewees emphasized the physical challenge of navigating the classroom and interacting with students sitting in hard-to-reach spots: “*there are 130 people, you’re going to run into backpacks, walking between different chairs. It’s not the best way of walking around*” (S4). Moreover, because exercises were often short, instructors did not have time to gauge students' mistakes and adapt their teaching later on: “*I want to get a sense for how everyone is reacting so I can decide what I’m going to talk about next*” (S1).

8.2.2.4 Difficulty in connecting students with each other

All interviewees encouraged students to talk to each other while performing exercises: “*I don’t care if they get the answer right. As long as you have the discussion you learn from it*” (S4). Four interviewees applied peer instruction [39] to multiple-choice questions in their lectures (S1, S2, S3, S5) and found it encourages students to monitor themselves in learning and build connections with each other. However, instructors reported a lack of intrinsic motivation for students to interact with their peers (S1–3). Instructors reported ineffective grouping as an important reason. Due to the physical distance between students in classrooms, instructors often pair students with their neighbors. Pairs are matched without regard to their diverse backgrounds, solutions, and levels of knowledge, which does not ensure that students in a pair would have meaningful conversations with each other (S3, S5). In addition, instructors mentioned the social hurdle for nudging peer interactions. Students would feel hesitant to engage with others socially without proper prompting. This corresponds to prior work on structuring roles and activities for peer learners to increase student engagement and process effectiveness [160, 101]. Instructors also reported that students would feel less comfortable asking for help from peers than instructors (S3, S4).

8.2.3 Summary of Design Goals

Driven by the findings, we formed three design goals for tools that scale support for in-class programming exercises:

- **Improvising in-class exercises and synchronous code sharing:** Instructors need a synchronous code-sharing platform for improvising in-class exercises. Tools should support various teaching needs, including creating and sharing exercises, verifying students' solutions, monitoring students' progress and activities, and walking through answers.
- **Scale student support:** Students need to get timely and on-demand support during in-class programming exercises.
- **Encourage live peer interaction:** Tools should encourage students to interact with their peers in real-time to engage them and motivate active learning.

8.3 PuzzleMe Design

We designed PuzzleMe as a platform to improve in-class programming exercises for instructors and students. PuzzleMe allows instructors to easily create and share exercises, monitor students' progress, improvise exercises as needed, and demonstrate correct solutions through live coding. On the student side, PuzzleMe supports live peer testing and live peer code review to scale support and encourage peer interaction.

8.3.1 In-Class Programming Exercises

PuzzleMe supports the types of exercises that instructors described in our interviews: programming exercises², multiple-choice questions, and free-response questions. For a programming exercise, an instructor can provide a problem description (Fig.8.2.A) and starter code (Fig.8.2.C) for students to build on. The editor includes a read-only code area with instructor-specified input variables (Fig.8.2.B1) and assertions to evaluate program output (Fig.8.2.B2). Fig.8.2.E shows the code output and error messages.

PuzzleMe also supports creating impromptu exercises in response to students' feedback and performance. Instructors can import exercises they prepared in advance or improvise and modify exercises during class. As soon as the instructor modifies existing exercises or creates a new exercise, their modifications are propagated to students. PuzzleMe propagates character-by-character edits, as we found in pilot tests that these more frequent updates kept students engaged if they had to wait as instructors wrote the exercise.

²PuzzleMe currently supports Python but could easily be extended to other programming languages.

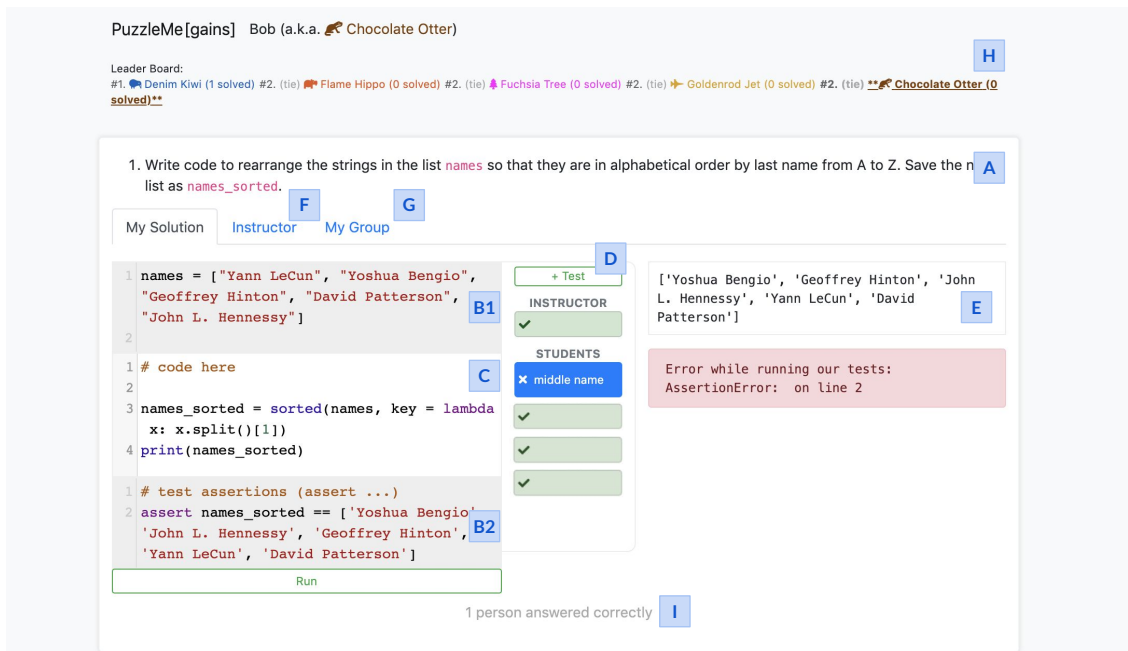


Figure 8.2: PuzzleMe is a peer-driven live programming exercise tool. The student view shows: (A) a problem description; (B) an informal test that consists of the given conditions (see B1) and the assertion statement (see B2); (C) a code editor where students can work on their solutions; (D) a test library that contains valid tests shared by instructors and other students; (E) the output message of the current solution; (F) access to the instructor’s live coding window; (G) access to peer code review; (H) a leaderboard of the number of problems that students have finished; and (I) the number of students who have finished the current problem.

8.3.1.1 Live coding for answer walkthrough

Prior work has found that students prefer when instructors write out solutions in front of the class—known as “live coding” [153]. Live coding also allows instructors to teach through experimentation (for example, by demonstrating potential pitfalls as they go along), narrate their thoughts, and engage students by asking questions [153]. PuzzleMe enables live coding and propagates instructor’s code changes to students (Fig.8.3), which allows students to easily copy and experiment with the instructor’s code. PuzzleMe also enables free-form sketches and annotations to allow instructors to draw explanatory diagrams to augment their code.

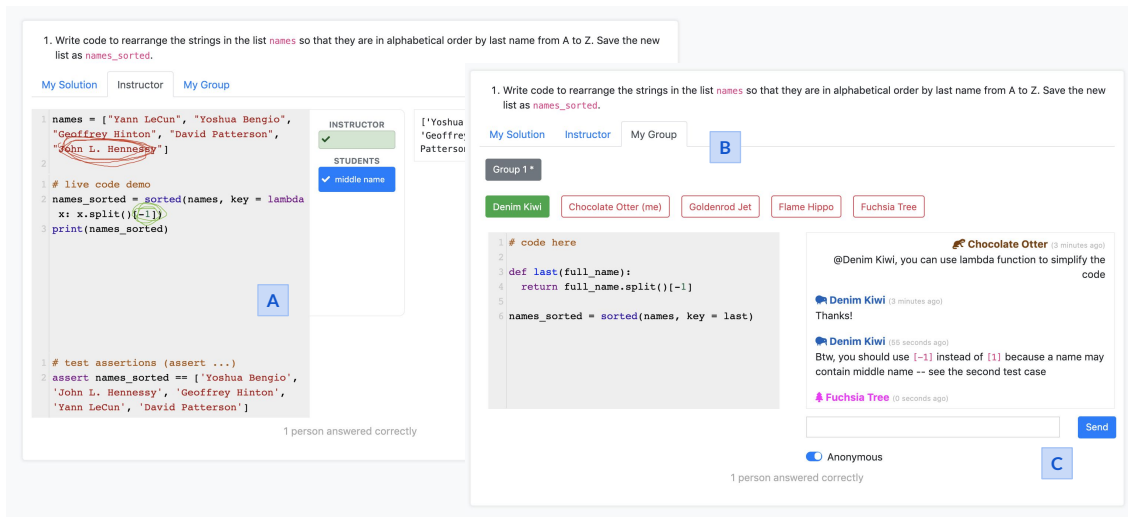


Figure 8.3: The live code view and the peer code review view. (A) Instructor can enable live coding mode to demonstrate coding in real-time, and use the built-in sketching feature to assist their demonstration; (B) Live peer code review allows students to check other group members' solutions and provide reviews in a chat widget (as shown in C).

8.3.1.2 Monitoring class progress

To monitor students' progress, similar to Codeopticon [72], PuzzleMe allows instructors to examine an individual student's response in real time. In addition, PuzzleMe presents an anonymous leaderboard (Fig.8.2.H) based on exercise completion time. PuzzleMe also displays how many students have written working solutions, to give a sense of progress relative to their peers (Fig.8.2.I).

8.3.2 Live Peer Testing

To give students timely feedback during in-class programming exercises, we designed **live peer testing**—a practice of writing and sharing lightweight tests in real time—in PuzzleMe.

8.3.2.1 Conceptual model of testing

One of the challenges of enabling live peer testing in introductory programming courses is that many testing frameworks require advanced knowledge [54]. For example, Python's `unittest` module requires an understanding of classes, inheritance, user-defined func-

tions and more. By contrast, students in introductory courses often do not learn how to define functions or intermediate control flow mechanics until several weeks into the course. Students in introductory courses benefit more from identifying interesting and representative examples rather than constructing comprehensive test suites [140]. We thus designed a model for live peer testing that would be flexible enough to test any number of configurations while still being conceptually simple enough for students in introductory classes. In our model, the code editor is split into three parts: 1) setup code that specifies pre-conditions, 2) the student's code, and 3) assertion code that tests whether the student's code produces the correct output given the pre-conditions. The setup (Fig.8.2.B1) and assertion code (Fig.8.2.B2) are "paired", independent of the student's code (Fig.8.2.C). This can be visualized as a "flipbook" that flips through pairs of pre-conditions and expected post-conditions.

8.3.2.2 Creating test cases

Students can create a new test case by clicking the "+ Test" button (Fig.8.2.D) and editing the setup code and assertion code. Instructors can specify whether tests are disabled, enabled, or enabled and mandatory. In addition, instructors can choose to require that students write a valid test case before they can start writing their solution, which can be useful for adapting different types of pedagogy like test-driven learning [83].

8.3.2.3 Verifying and sharing test cases

To help guide students to create effective test cases (and avoid sharing invalid test cases), PuzzleMe includes mechanisms for verifying test cases. For this to work, instructors write a reference solution that is hidden from students. To be considered acceptable, a student test must pass the reference solution (to prove that it is valid) and fail an empty solution (to prove that it is non-trivial, such as empty tests).

PuzzleMe notifies students of their test case status and students can always update and resubmit their unverified test cases. PuzzleMe shares verified test cases with all students as a test library (Fig.8.2.D). When students execute their code (pressing the 'Run' button), PuzzleMe uses all the cases in the library to examine their answers.

8.3.3 Live Peer Code Review

Live peer testing provides feedback on students' test cases and creates a test library to help them write more robust code. However, passing test cases does not ensure high-quality code. Valid solutions may contain unnecessary steps or bad coding practices. This might not affect the output of the program but may harm students' long-term coding ability [117, 61]. Thus, it is important for students to get feedback on both the correctness and the quality of their solutions. Prior work suggests that providing comparisons can help novice learners better construct feedback [29, 144]. PuzzleMe supports **live peer code review**, a feature that allows students to see and discuss each other's code. As Fig.8.3 shows, students can check other group members' solutions and provide reviews in a chat widget.

We originally designed the discussion widget (Fig. 8.3) as a "peer help" tool where students could press a "help" button to request assistance from another student. However, in pilot testing, we found that struggling students were hesitant to ask for help, even when they could do so anonymously. By framing this discussion as peer code review rather than peer help, however, PuzzleMe removes students' stigma around asking for help while still enabling valuable discussions between students.

8.3.3.1 Matching learners

By forming students into groups after working individually on the problem, our goal is to encourage meaningful conversations around everyone's solutions—for students who have incorrect solutions to clarify misconceptions and for students who have correct solutions to learn from others who use a different approach to solve the problem. Therefore, matching learners effectively is crucial for encouraging meaningful discussions among group members. Based on instructors' needs in the formative study to leverage peer help and match peers with diverse solutions so that they could learn from each other, we propose two heuristics for matching learners. First, students who have incorrect solutions should be paired with at least one student who has a correct solution. Second, if a group has multiple students who have correct solutions, they should have different approaches to or implementations of the problem. We determine group sizes by the proportion of students with incorrect solutions so that students who have incorrect solutions are paired with at least one student who has a correct solution. Next, we calculate the Cyclomatic Complexity Number (CCN) of the correct solutions as a proxy measure for approach. We then allocate the correct solutions by as many different approaches as possible. In addition,

students may feel hesitant to start conversations because of social barriers or feel stressed when their solutions are put together with others [53]. To address this issue, PuzzleMe keeps students anonymous [121] when sharing their solutions and reviews.

8.3.4 Implementation

We implemented PuzzleMe as a web application. We used ShareDB [2], a library that uses OTs to keep instructors' and students' data updated in real time. To ensure that it can scale to large numbers of students, PuzzleMe executes all code client-side, using Skulpt [8], a library that transpiles Python code to JavaScript. We have published our source code³ for researchers to evaluate and build on.

8.4 Evaluation

In total, we conducted three studies to evaluate the effectiveness of PuzzleMe for scaling support (Studies 1 and 2) and its general applicability (Study 3). To evaluate its effectiveness, we deployed PuzzleMe in an introductory programming class for two weeks. For the first week, we compared students' performance with and without the support of PuzzleMe in four lab sessions. For the second week, we collected and analyzed the PuzzleMe usage data for an online lecture. We then conducted an interview study with teaching staff from other programming classes to understand the broader applicability of PuzzleMe.

8.4.1 Course Background (Studies 1 and 2)

The introductory programming course consisted of a weekly lecture and two weekly lab sessions where students were assigned to one of nine smaller lab sections for hands-on practice of coding. There were 186 undergraduate students enrolled in the class, one full-time instructor (Professor) and five Graduate Student Instructors (GSIs). Most students did not have prior experience in programming. Each GSI individually led one or two lab sections with 10–30 students. The class format changed to online with optional participation in lectures and lab sessions during the second week of the deployment.

³<https://github.com/sonoy/puzzlemi>

8.4.2 Study 1: Using PuzzleMe in Face-to-Face Lab Sessions

To gain a holistic understanding of how PuzzleMe can be used to improve students' learning experience in in-class programming exercises, we conducted a user evaluation in four lab sections during the first week of the deployment. In particular, we aimed to answer the following questions:

- Q1 Compared to conventional methods, would *live peer testing* encourage students to write higher quality test cases? Are test cases shared by peers helpful for students to assess their code?
- Q2 Compared to conventional methods, how would *live peer code review* affect students' willingness to seek feedback from others? Would live peer code review yield more meaningful and constructive feedback?

We chose self-assessment and face-to-face discussion as a representation of conventional methods because they were widely applied by instructors in our formative studies.

8.4.2.1 Study setup

The GSIs gave students a set of problems to work on based on the material they were learning at the time. We used one of the programming exercises to evaluate live peer testing (noted as E1 for answering Q1) and another programming exercise to evaluate live peer code review (noted as E2 to answer Q2). For E1, the GSIs gave students 8–10 minutes to work individually and encouraged them to create additional test cases. For E2, the GSIs gave students around 5 minutes to work on the solution individually before placing them into groups. They then gave another 5 minutes to discuss with peers and continue working on their solutions. We used a between-subjects design where the four lab sections were randomly assigned to use PuzzleMe with or without the live features.

(Treatment) Using PuzzleMe with the Live Features: For E1, students were encouraged to write, verify, and share test cases using PuzzleMe, and use others' test cases to assess their code. For E2, PuzzleMe assigned students into groups to perform live peer code review after working individually on the problem.

(Control) Using PuzzleMe without the Live Features: Both live features in PuzzleMe were disabled in this case. Instead, students were encouraged to write test cases in their standard code editor for E1. For E2, students were asked to show their computer screens to people sitting next to them and discuss each other's solutions after working individually.

Table 8.2: The four lab sections were randomly assigned into the treatment condition or the control condition.

Session ID	Condition	GSI	Total Students
T1	Treatment	I1	16
T2	Treatment	I2	16
C1	Control	I2	12
C2	Control	I1	19

8.4.2.2 Data collection

We gathered the usage log of PuzzleMe (which tracks and timestamps every student submission attempt), the test cases students created in live peer testing, and the feedback students sent one another through live peer code review. In addition, we engaged in a follow-up interview with two GSIs and six students, where we asked about their learning or teaching experience of the lab sections and how they perceived the usefulness of PuzzleMe. Finally, we made observational notes during the lab sessions, where two of the authors sat at the back of each lab session and collected data on students' participation in the class, overall performance on the programming exercises, and usability issues with PuzzleMe.

8.4.2.3 PuzzleMe encourages students to create and share test cases.

Table 8.3 shows that with the live features, students wrote 0.72 test cases on average ($\sigma = 0.73$). Otherwise, students wrote 0.26 test cases on average ($\sigma = 0.44$). In both conditions, students wrote less than one test case on average. We believe this reflects realistic use of the tool with novice programmers in an in-class setting where students were given less time and were less motivated to engage in the exercises as compared to writing test cases as an assignment. To evaluate test quality, two authors manually coded student-written test cases into three levels (Figure 8.4), assigning a 0 if the test case was invalid or duplicated the default case, a 1 if the test case only performed additional checks on the output without changing the given conditions, or a 2 if the test case checked assertions under new conditions. We found that the average quality of the test cases in the treatment condition was 0.96 ($\sigma = 0.71$). The average quality of the test cases in the control condition was 0.63 ($\sigma = 0.74$). We did not find any incorrect tests that slipped through the validation procedure. We also found that 37.5% of students improved the code

quality after the group discussion in the treatment condition, while only 12.9% improved the code in the control condition.

Our comparison suggests that the number of test cases ($p = 0.002$, Mann-Whitney U test with power = 0.98) is significantly different in the two conditions. We do not find significant differences between the test quality in the two conditions ($p = 0.13$, Mann-Whitney U test with power = 0.353).

In the follow-up interviews, the students and instructors explained why they felt the live peer testing feature was useful. Live peer testing gives students feedback on their test cases, ensuring the quality of the shared test pool because PuzzleMe verifies the test cases against a known correct solution before sharing across the student body. In contrast, students in the control condition were hesitant to write new tests because they “*don’t know if my code is being tested correctly*” (C2). Second, the participants felt that writing tests improved their understanding of the learning materials overall. Students commented that the “*writing test was helpful to practice the new coding skill learned from class readings*” (T2). Similarly, I2 mentioned that “*I think [writing tests] might be helpful for students to think about what they should expect from their program*”. Lastly, both students and instructors reported that the live peer testing feature helped the former gain confidence in

<pre>1 names = ['Alice', 'Bob', 'Charlie'] 2 # code here 3 4 assert names_sorted == ['Charlie', 'Alice', 'Bob']</pre>	Score: 0
<pre>1 names = ['Alice', 'Bob', 'Charlie'] 2 # code here 3 4 assert len(names_sorted[0]) > len(names_sorted[1])</pre>	Score: 1
<pre>1 names = ['Alice', 'Bob', 'Mark'] 2 # code here 3 4 assert names_sorted == ['Alice', 'Mark', 'Bob']</pre>	Score: 2

Figure 8.4: An example of three different levels of test cases for one exercise (Problem description: alphabetically sort the given array, names, and assign the output to a variable named, names_sorted). Test cases were manually coded into three levels: 0 if the test case was wrong, meaningless, or duplicated the default case; 1 if the test case did not create new examples of names but added additional checks on the output names_sorted; 2 if the test case contained new examples of names and names_sorted.

Table 8.3: The number and quality of test cases students wrote in E1 (mean: \bar{x} , standard deviation: σ). The number of students who improved code (calculated by completion status) after group discussions in E2 (total: N). Our comparison suggests that the number of test cases is significantly different in the two conditions ($p = 0.002$, Mann-Whitney U test with power = 0.98); the number of students who improved code is also significantly different in the two conditions ($p = 0.02$, proportions z-test given the binary data type).

E1 Writing Test	Condition	\bar{x}	σ
Number of Test Cases*	Treatment	0.72	0.73
	Control	0.26	0.44
Test Quality	Treatment	0.96	0.71
	Control	0.63	0.74
E2 Code Discussion	Condition	N	Total
Number of Students Who Improved Code*	Treatment	12	32
	Control	4	31

their solutions, and both instructors indicated that PuzzleMe might help identify problems in students' solutions. Moreover, the instructors reported that the live peer testing feature reduced their teaching stress as students gained confidence:

(PuzzleMe) takes off some stress from me to check students' code. A lot of times students don't have a lot of questions, but they want me to check their code and make sure their code is correct. Students always have more faith in other students than themselves. If they pass their own assertion, they will still be unsure. If they pass others' assertions, they will be definitely more sure.
(I2)

8.4.2.4 PuzzleMe scaffolds group discussions.

We observed that the face-to-face group discussions were largely affected by the layout of the classroom in the control condition—“(Face-to-face discussion) depends on the physical setting and how many people are sitting. One of my lab session[s] is smaller while the other session is more spread out” (I2). In addition, some students found it difficult to have meaningful conversations with their neighbors in the classroom setting. One student mentioned in the follow-up interview, “Sometimes, neither of us know the solution. It's a little awkward” (a student from C2). Instructors reported that the matching mechanism overcomes the physical limitation of face-to-face discussion:

In my class, students who sit in the front always finish their code, so talking to neighbors didn't really work. I like the matching in PuzzleMe because it can really pair students based on their solutions. (I2)

8.4.2.5 PuzzleMe encourages students to explore alternative solutions.

PuzzleMe encourages students to explore and discuss alternative solutions, which may help them better apply the concepts they learned. For example, one session covered sorting and advanced functions. Students could solve a problem by either passing a lambda expression or a traditional named function to the `sorted` function. Students found it useful to see others' solutions and understand both ways to solve the problem—“*PuzzleMe is very helpful especially my classmates ask questions that I didn't think to ask*” (a student from T1); “*Live peer code review is good because we get to see the other ways people do their code.*” (a student from T2).

In addition, the results suggest that the live peer code review feature helps students improve their completion status. As shown in Table 8.3, the number of students who improved code ($p = 0.02$, proportions z-test given the binary data type) after group discussions in E2 significantly improved with the live peer code review feature. However, we suspect that multiple factors may lead to an improvement in completion status. For example, students may directly copy and paste peers' solutions without thinking, which is not our intention when designing live peer code review. To understand how the live peer code review feature helps students complete the code exercise, we continued with Study 2 to collect the PuzzleMe usage logs from an online lecture.

8.4.2.6 Limitation

Although we designed the experiment to balance the multiple confounds (e.g., instructors, room size, as shown in Table 8.2), it is difficult for us to conduct a rigorous comparison between the two conditions given that we deployed PuzzleMe in an authentic usage scenario. Factors like total students who showed up to each session were hard to control and may reduce the external validity of the results. Thus, we focused on empirical evidence of how PuzzleMe can be used to conduct in-class programming exercises. A well-controlled exhaustive study—including use of the application during a complete course and evaluation of the students' grades or the instructors' perceived workload—will be needed to explore the pedagogical benefits of PuzzleMe.

8.4.3 Study 2: Integrating PuzzleMe in an Online Lecture

Study 1 indicates that the live peer code review feature is related to the improvement of code completion. To further explore whether students were using the tool as we expected, we collected and analyzed the PuzzleMe usage data for a lecture. By the time of the deployment, the class was changed to an online format where students attended lectures synchronously using video conferencing tools. This allows us to additionally test the benefits of using PuzzleMe in online classrooms.

8.4.3.1 Study setup

During the live-streamed lecture, the instructor gave students an in-class programming exercise through PuzzleMe to practice the concepts they learned about that week (higher order functions—`map`, `filter`, and list comprehensions). There were multiple ways to solve the problem and students were encouraged to explore and find the most concise solution. After initial exploration for about five minutes, the instructor turned on live peer code review mode and asked students to discuss with their peers. We collected and analyzed the events log from the usage data. In total, we collected data from N=48 students who participated in the programming exercise.

8.4.3.2 Results overview

Figure 8.5 shows the event logs where each row represents a student and the x-axis represents the timeline. We sorted students based on the first time they passed the default test. On average, each student had 13.75 attempts with incorrect solutions (meaning they ran their code and failed at least one test case) and 2.63 attempts with correct solutions.

PuzzleMe connected students who were struggling with the problem with their peers for help. Before the live peer code review started, 17 students passed the problem. After the peer code review activity, an additional 10 students were able to pass the problem. Finally, 6 students passed the problem after the instructor revealed the correct solutions, and the remaining 15 students did not finish the problem within the given time (1000 seconds). We observed an additional 8 students passing the problem after the given time.

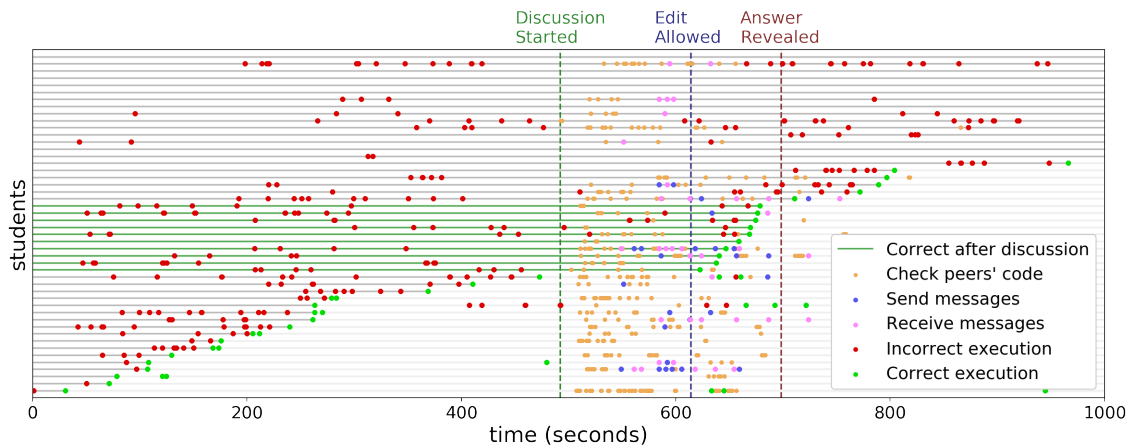


Figure 8.5: Usage logs from the online lecture. With the help of peers, 10 students who had incorrect solutions were able to pass the problem (as indicated in green horizontal lines). Live peer code review helps students identify cavities in their code and inspires them to explore alternative approaches.

8.4.3.3 Code review is correlated to better completion status

For students who were not able to complete the problem before peer code review, we ran a proportions z-test to identify whether there is a correlation between using the code review feature (as indicated by blue and orange dots in Figure 8.5) and solving the problem. The result shows that there is a strong correlation between using the code review feature and eventually solving the problem ($p = 0.02$). We examined the editing histories of the code and did not observe students directly copying and pasting others' solutions. Our interpretation is that students are self-motivated to work out their own solutions rather than having a correct solution since the programming exercise is voluntary and not associated with grades. This corresponds to the observation that students who failed to improve the code were inspired by their peers' code while continuing to work on their original solutions.

8.4.3.4 Who initiates the talk? Conversations need nudging.

Although students reported in Study 1 that they felt more comfortable talking to peers in PuzzleMe, we observed that there was no discussion going on in 8 of the 16 groups. Most members in the 8 groups “shied away” from talking to peers, though they still actively engaged with the tool to check their peers' code or make code execution requests.

<pre> 1 common_words = ['of', 'a', 'the', 'an'] 2 3 def acronymBuilder(sentence, to_ignore = common_words): 4 return ''.join([i[0] for i in sentence.split() if i not in common_words]) </pre>	Solution A
<pre> 1 common_words = ['of', 'a', 'the', 'an'] 2 3 def acronymBuilder(sentence, to_ignore = common_words): 4 return ''.join([i[0] for i in sentence.split() if i not in to_ignore]) </pre>	Solution B
<pre> 1 common_words = ['of', 'a', 'the', 'an'] 2 3 def acronymBuilder(sentence, to_ignore=common_words): 4 words, acro = [], [] 5 for word in sentence.split(): 6 if word not in to_ignore: 7 words.append(word) 8 for word in words: 9 acro.append(word[0]) 10 return ''.join(acro) </pre>	Solution C

Figure 8.6: Three example solutions that pass the default test. Solution A is a false positive because students did not use the function arguments correctly. Solution B is the most elegant way to solve the problem. Solution C is correct but does not demonstrate an understanding of advanced list operations.

In half of the other groups, students who already had the correct solutions (in the helper role) initiated the conversation; in the other half, students who sought help initiated the conversation. This result corresponds with the general challenge of nudging peer-driven conversations in formative study. Next, we looked into the content of the conversations and found that the common topics included making comparisons between various solutions, providing suggestions on variable naming and formatting, seeking help on debugging, and clarifying the lecture content. In addition, we examined the group discussions and did not find any propagation of misconceptions through peer code review.

8.4.3.5 Code sharing improves engagement.

Lastly, we observed that students who used the live peer code review feature tended to stay engaged with the exercise. For students who passed the default test, PuzzleMe inspired them to explore alternative approaches and think about issues in their code that they might have missed. We manually examined students' code and found some solutions contained a similar bug that the default test case did not catch (as shown in Figure 8.6, Solution A). This bug was caused by students' misunderstanding of a critical concept in previous

lecture sessions—using default values for function arguments. Among the 17 students who initially passed the problem, 8 students made this mistake. Through talking to peers and checking their code, three students were able to identify the bug in their original code. In addition, several students passed the problem with correct yet naive solutions (as Figure 8.6, Solution C shows). PuzzleMe informed these students of the existence of other solutions and encouraged them to explore them further. In total, we found 13 students who had additional attempts after they passed the problem the first time.

8.4.4 Study 3: The Practical Applicability of PuzzleMe

To explore the practical applicability of PuzzleMe, we conducted an exploratory study with four programming course instructors from the authors’ university. Participants had varied experience teaching a wide range of programming topics in both in-class and online settings (as Table 8.4 shows). We first gave participants a walkthrough of PuzzleMe and asked them to interact with the features. Then we asked them to brainstorm the possible use cases of PuzzleMe in their courses. We also encouraged participants to envision other features they would like to have in a future design.

8.4.4.1 The use case of PuzzleMe in various programming topics

Table 8.5 summarizes the use cases for PuzzleMe. For live peer testing, most of the use cases (6/7) relate to specific topics (e.g., UI testing), and the rest (peer challenge) are class activities that can increase students’ engagement (P3). The six specific topics are categorized as the functionality of a system (e.g., security testing, UI testing, data visualization), different modalities (e.g., in-circuit testing, design feedback), and more

Table 8.4: Participants’ background in Study 3.

PID	Course Name	Course Size
P1	Intro to Computer Security	300
P1,P2	Intro to UI Development	150
P2	Intelligent Interactive System	50
P3	Applied Data Science (Online)	145
P4	Natural Language Processing	100
P4	Data Mining	80

generic topics (e.g., algorithm design).

In particular, P1 listed a series of topics in computer security that could use the live peer testing feature, including fuzz testing, side channel attacks, and cross-site scripting attacks. Analogous to the test-driven learning approach [83], P1 envisioned that the live peer testing feature could help students learn programming concepts by writing test cases to “*break the instructors’ program*” (with respect to data security), as prior work has proposed [167]. Instructors (P1, P2) also suggested peer design feedback in which students can benefit from diverse responses from their peers.

More than half of the use cases (3/5) for the live peer code review feature related to team collaboration (e.g., guided peer support, working in groups, group competition), and the other two related to team matching. P1 suggested that rather than showing each other’s code immediately, which may cause students to “*lose the motivation to work on your own implementation*”, a future design of PuzzleMe could allow one student to guide others to complete the problem first, and then unlock each other’s code for further review. P3 mentioned that using a leaderboard among individuals or groups would motivate students to be more engaged with the exercise.

Participants also envisioned a set of use cases for other subjects. P2 mentioned that students in graphical design or creative writing courses could get peer support and feedback on their artifacts without solely relying on and waiting for their instructors’ feedback. P3 suggested that PuzzleMe could be used for coding interviews (e.g., “*it’s kind of similar to hackerrank (a coding interview site)*”), where one student plays the role of the interviewee to write the program and the other students are interviewers writing test cases to challenge their peer.

8.4.4.2 PuzzleMe lowers the effort for setting up in-class exercises.

Besides use cases, participants also pointed out the potential benefits of using PuzzleMe for lowering the effort involved in setting up in-class exercises. For example, in a UI development class, students can take a UI front-end-related exercise on PuzzleMe without the effort of configuring the environment (P3). Similarly, for exercises that require external resources (e.g., libraries, data sets), PuzzleMe could provide a resource hub to which the students can easily connect (P4). More generally, PuzzleMe could support instructors to create new exercises by modifying the previous ones, making it easier for students to practice on the same topic iteratively (P1, P4).

Table 8.5: Use cases of two PuzzleMe features—live peer testing and live peer code review—in various programming topics.

Live Peer Testing

Computer security-related testing: Test the security level of others' programs (e.g., login systems).

UI testing: Navigate each other's UI (e.g., the responsiveness effect).

Data visualization: Interact with each other's viz. systems (e.g., missing value, different input data).

Algorithm: Test edge cases for others' algorithms (e.g., regular expression to extract domain from a URL).

In-circuit testing: Test each other's circuits (e.g., virtual probe for breadboard testing).

Physical and digital artifact design feedback: Write feedback on each other's designs.

Peer challenge: One student writes a program, the other writes tests to challenge (break) it.

Live Peer Code Review

Working in a group: Students can work in groups to solve problems.

Roleplay: Students can choose to be a helper or a help seeker based on their interests.

Guided support: Students can provide hints to other students in their group.

Different matching mechanism: Match students by their process (similar/different), or engagement level.

Group competition: Students are divided into groups and compete with other groups.

8.5 Discussion

Our evaluation studies demonstrate that the design of PuzzleMe allows instructors to improvise in-class programming exercises, while effectively leveraging peer feedback through live peer testing and live peer code review. In particular, PuzzleMe motivates students to write more test cases, identify gaps in their code, and explore alternative solutions—all important pedagogical goals of an introductory programming course. We reflect on the design of PuzzleMe and discuss the implications for future HCI and Computer-Supported Cooperative Work (CSCW) research.

8.5.1 Design Lessons

8.5.1.1 The benefits of learnersourced test creation

Learnersourced test case creation benefits multiple stakeholders. Although individual students' test case coverage might not be as thorough as those written by instructors, they

save time and effort and enable improvised programming exercises. More importantly, the process of creating test cases helps learners verify their understanding of the problem and of test-driven development while simultaneously contributing to a larger pool of verification instruments to be used by the whole class. Writing tests also helps learners practice the ability to predict the outcomes of a given input by manually walking through their code (either in their mind or by writing intermediate results on paper), a critical strategy for scaffolding novice programmers [196]. Finally, the test cases themselves provide additional diagnostic material that might be used by instructors post-hoc, allowing them to reflect on the misconceptions students formed and aiding in the design of future learning activities.

8.5.1.2 The social and cultural value of building collaborative learning platforms

The live peer testing and live peer code review features in PuzzleMe scaffold peer interactions in programming classes. We observed that students used topics related to their interests when creating test cases (e.g., in the name-sorting exercise, students created test cases using names that are popular nationally, regionally, and culturally), which shed light on the unique social and cultural aspects of students' backgrounds [38]. We believe that the opportunity to create test cases related to their interests adds additional motivation for students to engage in the test creation process. In addition, our evaluation indicates that through sharing tests and discussing code with peers, students feel more engaged and connected with the class. Particularly in online classrooms, students would largely benefit from a stronger degree of social presence [111] by interacting with their peers in various learning activities. Designers of future collaborative learning platforms may learn from our design and leverage synchronous technologies to build social and structured activities [33, 29, 121] in platforms that connect students.

8.5.1.3 The challenges of connecting students

We designed PuzzleMe as a platform for easily creating and distributing programming exercises while engaging students through live peer testing and live peer code review. As we found when designing PuzzleMe, students might feel shy or hesitant to talk to each other, especially in situations that might expose their lack of understanding of the material to their peers, such as asking for help. We proposed various design decisions to create an encouraging collaborative space, such as providing real-time feedback on test cases so that students are comfortable sharing them with others, keeping students any-

mous when sharing code and reviews, and allowing instructors to monitor and intervene in group conversations. However, our evaluation suggests that prompting conversations between students is still challenging. Compared to live peer code review, students might feel more comfortable with non-conversational interaction with peers (e.g., sharing and using test cases created by others). Future research could look into the reasons that students fail to connect with their peers and address the challenge from both social and technical perspectives. For example, it is worth exploring the effect of the pairing mechanism on students' self-efficacy and power dynamics in group discussions.

8.5.2 Future Work

8.5.2.1 Towards test-driven learning

As participants in Study 3 mentioned, PuzzleMe can be useful in supporting *test-driven learning*, a pedagogical approach that can improve comprehension of concepts but requires extra learning effort in creating test cases, particularly in early programming courses [83]. PuzzleMe can help address these challenges by reducing the barriers and learning costs for students to write test cases. PuzzleMe introduces a straightforward design that maps the given variables, solution code, and assertion statement in a linear order so that even students who have no experience in testing frameworks can easily pick up how to create tests. PuzzleMe ensures that students can get immediate feedback on the test by running it against the instructor's standard solution. This mechanism increases students' confidence in their test cases, and thus the problem formulation, before even starting to write a solution. Finally, peer-driven test creation provides the opportunity for HCI and CSCW researchers to further explore different implementations of test-driven learning. For example, the instructor can provide a solution with intentional bugs and ask students to identify edge cases to catch these bugs or misconceptions.

8.5.2.2 Peer assessment beyond introductory programming

Adopting PuzzleMe's approach beyond introductory programming might require further design exploration regarding aspects such as assessment format or content presentation. Prior work has explored ways to improve peer assessment quality in open-ended tasks, such as providing comparisons [29], framing task goals carefully [77], and using expert rubrics [202]. Future work could explore the use case of live peer testing and live peer code review in open-ended assessment on programming-related topics (e.g., providing

feedback for system architectural design, code review). Content-wise, P3 from Study 3 suggested a “top down camera view” for in-circuit live peer testing. Prior work also introduced a representation of Graphical User Interface (GUI) test cases that is more readable than a standard textual log [34]. Future work could explore the appropriate design for exercises that require more than a text exchange between peers [26].

8.5.2.3 Towards matching peers intelligently

PuzzleMe leverages the correctness of students’ code for peer matching, but future work could use different criteria. For example, one could extract code fixes from students who have achieved the correct solution after multiple attempts and apply them to students who have incorrect solutions [74], capturing conceptual pathways through the problem space. One could also connect students who make similar mistakes, where one has resolved the problem and others have not, ensuring the help giver has experience in addressing their peers’ issues. The matching criteria may also depend on the deployment context. For instance, in Massive Open Online Courses (MOOCs), creating culturally diverse groups of learners may provide additional learning opportunities—students may be able to not only learn a given programming concept but gain intercultural competencies while doing so.

8.5.3 Limitations

The design of PuzzleMe was tailored for introductory programming courses, and the design of the live peer testing component was focused on simple programs made up of a pair of given conditions and expected outputs. Advanced testing techniques like exceptions, callbacks, and dynamic tests are not implemented in the current system but may represent additional opportunities for learner collaborations. PuzzleMe’s current design cannot be used directly for non-text-based programming and testing, like UI testing or Printed Circuit Board (PCB) testing. In addition, our evaluation was done on a small scale with fewer than 50 subjects. Future work should explore the effectiveness of peer assessment mechanisms in larger classrooms or MOOCs.

8.6 Conclusion

This paper presents PuzzleMe, an in-class programming exercise tool for providing high-quality peer feedback at scale. PuzzleMe achieves this by two peer assessment mechanisms: live peer testing, which allows students to identify and share test cases for assessing the robustness of programming work, and live peer code review, which groups students intelligently to improve code understanding. Our evaluation study demonstrates the usefulness of PuzzleMe in helping students identify cavities in their code and explore alternative solutions, and in reducing the teaching load for instructors. PuzzleMe opens up possibilities for HCI and CSCW researchers to further study learnersourced test creation and test-driven learning in introductory programming courses.

8.7 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos IIS 1755908 and DUE 1915515.

CHAPTER 9

Colaroid: Authoring Explorable Multi-Stage Tutorials

Multi-stage programming tutorials are key learning resources for programmers, using progressive incremental steps to teach them how to build larger software systems. A good multi-stage tutorial describes the code clearly, explains the rationale and code changes for each step, and allows readers to experiment as they work through the tutorial. In practice, it is time-consuming for authors to create tutorials with these attributes. In this paper, we introduce Colaroid, an interactive authoring tool for creating high quality multi-stage tutorials. Colaroid tutorials are augmented computational notebooks, where snippets and outputs represent a snapshot of a project, with source code differences highlighted, complete source code context for each snippet, and the ability to load and tinker with any stage of the project in a linked IDE. In two laboratory studies, we found Colaroid makes it easy to create multi-stage tutorials, while offering advantages to readers compared to video and web-based tutorials.

9.1 Introduction

Programmers often need to communicate how a program is built in increments from scratch. For instance, instructors teach programming students by showing them not just final solutions, but also how those solutions are written step-by-step, demonstrating the process of designing programming solutions as they do so. Streamers [56] broadcast their programming activities live to show how they build, deploy, and debug software projects of broad interest. During everyday work and study, programmers write reports and blog posts describing how they wrote code to reflect on their process and seek feedback. Members of software teams explore how code evolves by reviewing versions of that code in

pull requests and commit records. And authors of software libraries author getting-started guides [40] that show how to create simple applications that incorporate their software.

Ideally, one could create beautiful records of how programs are constructed with ease. Such a record might allow readers to easily see what has changed in the code from one stage of its development to the next, and empower readers to execute and tinker with each version of the code in their own development environment. However, in reality, creating such records of program construction can be time-consuming and difficult [134, 76].

Recently, computational notebooks have seen widespread adoption as a medium for creating rich descriptive documents about code. Notebooks allow authors to blend programming instructions with annotations of that code. In this way, computational notebooks support the practice of literate programming [96], or writing programs as “essays” intended to be read. Because they support the possibility of integrating code snippets, documentation, and figures, such notebooks have been used by data scientists to create code tutorials [103, 44]. They support a kind of exploration and tinkering that is central to “learning by doing”[97], because in a computational notebook, a code cell can be modified and executed, allowing readers to explore how changes to the code influence the results. Despite their value in describing programs in domains like data science, the notebook paradigm has yet to influence the practice of describing how code is built in stages. This is because the predominant execution paradigm of contemporary notebooks is one where each cell is executed independently by submitting it to a REPL (read-eval-print loop). Instructions are submitted to an interpreter in sequential order. Thus, in practice, these tutorials typically consist of cells of code at the granularity of standalone functions or processes.

In many domains of programming, code is developed through a cyclical process of editing, compiling, and running the code. For example, a programmer may iteratively tweak the labels of a data visualization until they arrive at a set that succinctly and clearly describes the data, or a web programmer may build a web program through incremental elaborations to “spaghetti code” [120] split across multiple files. Such a process, while common to software development and imperative to convey in many programming media, is difficult to describe in a conventional notebook. How can we help programmers document incremental code construction for a broader set of programming tasks?

In this paper, we introduce Colaroid,¹ a *temporal-based notebook* that enables authors to flexibly track and document multi-stage code construction, and creates tutorials

¹A portmanteau of the words “code,” referring to coding tutorials, and “Polaroid” [194], a series of cameras that allowed photographers to easily and rapidly take and print a series of photographs.

that are interactive, explorable, and IDE-integrated. Colaroid stands out from traditional computational notebook tools in several ways. First, it is embedded within the context of the authentic practice environment — the IDE where programmers can work on any programming activities in their familiar programming environments. Second, each code cell is made up of code changes, allowing programmers to organize the steps based upon pedagogical considerations rather than syntactic constraints. Meanwhile, Colaroid organizes the explanations and code snippets into the computational narrative structure for storytelling, providing output previews of each cell, and allowing users to easily tinker and explore an intermediate step.

We conducted two studies to evaluate the usefulness of Colaroid in the context of web programming tutorials. The first study focuses on the authoring experience where we asked instructors to create web programming tutorials on given topics using Colaroid. The second study explores the reading experience where we compared how readers interact and perceive differently among Colaroid, video, and article tutorials. The results show that the instructors find the process of creating web programming tutorials in Colaroid integrates well into their programming workflow. They found it easy to scaffold the programming process, annotate their thoughts while working on the programming, and post-edit the tutorials after they are done. In particular, they found it useful to not only explain the final solution, but also teach how to think like a programmer, demonstrate authentic practices of decomposing features, and show the hurdles of where things could go wrong. On the readers' side, Colaroid ensures that the narratives are easy to follow and reproduce. They found it better explains the construction of the program and the impact of code changes between steps. In addition, readers are more willing to explore and tinker with intermediate steps in Colaroid, and thus more engaged with the tutorials. Moreover, readers perceive that the Colaroid tutorials allow both quick skimming and deep dive, and take less time to read in general.

In summary, our work makes the following contributions:

- We contribute an alternative design of temporal computational notebooks that allow programmers to author computational narratives on incremental code construction;
- We implement Colaroid, a system that integrates the idea of temporal computational notebooks and tailors for the context of web programming;
- We reveal the advantages and limitations of this new approach from both the authoring and learning experience.

9.2 An Exploratory Analysis of Multi-Stage Programming Tutorials

Multi-stage programming tutorials allow authors to demonstrate the incremental construction process of a programming project and encourage learners to learn by doing. Prior work [76] has been done to describe technical tutorials broadly. In this work we are interested in the makeup of tutorials in a more narrow context, where the tutorial is intentionally designed to support learners who are replicating the work of the author by following a set of clearly delineated stages. To gain a better understanding of the nature and composition of stages in this tutorial format we expand on the work of [76] by engaging in an exploratory content analysis of 44 such tutorials to understand how authors arranged, formatted, and linked these stages together.

9.2.1 Collecting Representative Multi-Stage Tutorials

9.2.1.1 Selection Criteria

To better collect representative multi-stage tutorials, we came up with the following criteria:

- The tutorial must demonstrate the implementation process of a meaningful programming project. We exclude tutorials that are API documentation and example snippets, or blog posts that draw references to several code fragments from different projects. For example, tutorials on different ways to implement asynchronous programming in JavaScript would be excluded, while a tutorial which uses asynchronous programming in JavaScript as a stage in a project would be included.
- The tutorial must focus on achieving a specific programming project outcome. Thus we would exclude tutorials that teach configuration processes, such as how to configure a cloud service through GUI or using a given command line tool.
- The tutorial must contain at least two stages that involve coding. As we are interested in the mixture of technical and pedagogical support, we consider a stage to be a piece of the writing which has both English text which scaffolds the learning and code demonstrating how to achieve an outcome. Stages could also contain other forms of media support (e.g., images, animated GIFs).

9.2.1.2 Multi-Stage Tutorials Linked from Stack Overflow

Following the sampling methodology of Head et al [76], we harvested links to multi-stage tutorials from Stack Overflow. We scraped links from Stack Overflow answers that contained the keyword “tutorial”. The results were then filtered based on the recency (no later than 2017) and quality (has more than 5 up-votes) to narrow our investigation, and considered only the first 500 URLs matched. For each match we manually inspected each tutorial to determine whether it met our selection criteria. Of these tutorials, 27 (5.4%) matched our criteria, as many of the outgoing links from Stack Overflow responses were to API documentation, or tool-based tutorials. From this list of 27, the majority 17 of the tutorials (63.0%) were authored by official library teams or organizations and the remaining 7 (25.9%) were personal blog posts.

9.2.1.3 Multi-Stage Tutorials on FreeCodeCamp

To collect a wider variety of tutorials, we additionally collected 17 multi-stage tutorials that are personal blog posts on FreeCodeCamp, a popular programming tutorial sharing site. We located the tutorials by searching titles that contain keywords “step-by-step” or “from scratch”. We then manually skimmed through the tutorials and filtered them based on the selection criteria. In addition, we only kept one unique tutorial if there are multiple tutorials from the same author.

9.2.1.4 Data Analysis

Three authors filtered and initially examined the sampled tutorials. The selection criteria was iteratively refined on a sample of 50 tutorials until substantial agreement was attained (Fleiss’ $\kappa \geq 0.8$). In total, we collected 44 step-by-step tutorials. A list of tutorials that were analyzed appears in Appendix ???. Then one author selected tutorials using the criteria, and then conducted an initial qualitative analysis via open coding [168] to identify common themes (as shown in Table 9.3) related to the research questions. The themes were discussed, refined, and categorized by three authors. More specifically, we categorized the themes in to five aspects: scaffolding strategies, composition of code snippets, presence of code snippets, presence of intermediate results, and strategies to support learning by doing.

9.2.2 Results

9.2.2.1 How do authors scaffold the stages?

As shown in Table 9.3, we identified three different strategies for scaffolding the stages — iterative build-up, module-based build-up, and aggregated build-up:

Some tutorials use an iterative build-up strategy, where the current stage iteratively builds upon previous stages. In iterative build-up tutorials, authors may make changes at any line of the codebase. For example, T1 contains a stage where the authors declare a function definition. It then wrapped the function into a class definition. This scaffolding approach is used more in web programming or mobile development tutorials because of spaghetti code [120].

In contrast, module-based build-up and aggregated build-up have a complete block of changes that are self-contained. For module-based build-up, the module code blocks are independent of each other. They may be positioned in separate files and do not need to be combined in a certain order. We observed module-based build-up for both web programming and data science programming tutorials. For example, T13 is a data science tutorial on fine-tuning models. T13 is implemented with the functional programming paradigm where each stage declares a pure function. This tutorial focuses on how each function is implemented, rather than how functions are combined and used together.

For aggregated build-up, the new code block can be linearly appended to the end of the previous code base. This scaffolding approach is used more often for data science programming or machine learning programming tutorials.

9.2.2.2 How do authors structure the code snippets for a stage?

Next, we summarized the structures of the intermediate code snippets and discussed how they fit with the scaffolding strategies. The first composition structure we observed is showing changed code only. For example, T5 demonstrates the usage of an API for Android development and it involves changing multiple files. The tutorial contains only the modified code lines for stages. Without enough context, it might be hard for readers to understand where the changes occur. On the other side, we observed that most module-based build-up tutorials and aggregated build-up tutorials choose to display only the changes. Displaying only the changes is enough since the code blocks are independent of each other, and the learners can just position them at the end of the codebase. In addition, it saves space and makes the tutorial concise. For iterative build-up tutorials,

authors usually provide the context of the changes. Some tutorials provide the complete context of a relevant code file. However, this may not work when the code file gets long and the tutorial may contain too many duplicated code. Thus, some tutorials choose to provide partial context by attaching the code lines nearby the changes.

9.2.2.3 How do authors present the intermediate code snippets?

We found that most code snippets were rendered in the article with visual styles to make them stand out from the plain text. Most tutorials will wrap the code snippets with a different background and font style. And some tutorials further add syntax highlights to make them more distinct from other elements in the tutorial. In addition, code changes are highlighted in tutorials that have provided full or partial context. To help readers locate how the intermediate code snippets fit into the context, several tutorials (T4, T9, and T10) use mechanisms such as adding line numbers, adding file names, or directly explaining where the changes should go to. Notably, these styled code snippets with change highlighting and context locators are more likely to be found in official library tutorials. Most personal blog post tutorials use Markdown inline code rendering and are limited at tracking code changes and locating the changes in the entire project source code.

9.2.2.4 How do authors present the intermediate results?

In addition to the presence of code snippets, we investigated the display of output previews in tutorials. We believe that providing intermediate results can help readers better understand what they need to achieve in each stage when reading through a tutorial. It also helps readers to align their progress if they choose to learn by replicating the stages in the tutorials locally. However, we found that most tutorials have low coverage of intermediate results in general. We observed that authors may directly display the output if the output is textual (e.g., an output from console), take screenshots (either static image or animated GIF) of a visual output, textually describe what is expected to happen, or attach a working demo. Some tutorials only present the output of the project for the last stage. We suspect that this is due to the additional cost of presenting the intermediate results. For example, authors may need to embed a working partial version of the application which they would need to change if the code changes, or save a screenshot and upload it to the static assets of their site, or rehearse and record a GIF.

9.2.2.5 How do tutorials support learning by doing?

Lastly, we investigated elements in the tutorial that might encourage readers to learn by doing. It is common to see tutorials attaching a link to the final source code for readers to dive into details. Some tutorials also provide a starter code and encourage readers to follow along the way. However, in some cases, readers may not want to follow the tutorial from the beginning. Instead, they may want to skim the beginning, jump to a certain stage, and start from there. We observed that some tutorials provide an embedded code live playground where readers can directly tinker the code and see the updated output. However, these embedded code live playground are rarely provided for every stage because of the high cost of creating.

9.2.3 Design Opportunities to Improve Authoring and Reading Experience

To motivate the design of a literate programming approach for authoring multi-stage tutorials, our formative analysis explores the composition of representative multi-stage tutorials. Inspired by the results, we discuss potential challenges and opportunities to improve the authoring and reading experience of these tutorials.

9.2.3.1 Design Tutorial Authoring Tools to Capture and Document the Entire Incremental Building Process

Our formative study shows that multi-stage tutorials can be useful for demonstrating and explaining how a programming project is incrementally built from scratch. Depending on the programming project, the authors may create small or big incremental stages and provide the code and explanations for the stages. These tutorials allow learners to not only understand how the final source code works, but also the authentic practice of how to decompose the stages and build it incrementally. We argue that tutorial authoring tools should allow authors to capture the authentic incremental building process as if doing a video recording of the code editor.

9.2.3.2 Design Tutorial Authoring Tools to Capture the Context of Code Changes

In addition, we discovered three different ways to break down the coding process. For module-based build-up and aggregated build-up, existing approaches like computational

notebooks [137] and Codestrates [147] allow authors to treat each code change as an individual code cell and linearly aggregate them together in the final document. However, if the process can not be simply broken down into individual pieces, authors need to provide context to indicate where the incremental changes are positioned. More specifically, we observed several strategies for describing the context, including providing incremental changes only, providing the complete context of the current code file, and providing the partial context of the current code file. Comparing the strategies, we argue that it is more work to capture the complete context of the current code, and would result in a long and tedious document and cause information overload if the project scales up. However, due to expert blind spots, providing the partial context of the current code file or providing incremental changes only may result in learners' confusion to follow the stages. Thus, we further argue that tutorial authoring tools should help authors capture and efficiently present the full context of the changes.

9.2.3.3 Design Tutorial Authoring Tools to Preview Multi-Stage Output

Multi-stage tutorials usually capture both code changes and the output. The output of the stages can be presented in the form of screenshots, video recordings, or textual descriptions. However, we found that most tutorials only provide output previews for important stages, with an exception of a tutorial written in the Jupyter Notebook that captures the output preview for every stage. We argue that previewing the output of each stage is important for learners to understand the impact of code changes. Given that the process of creating an output preview for each stage is time-consuming, we believe that there is an opportunity to improve tutorial authoring tools to automatically capture the output overview.

9.2.3.4 Design Tutorial Authoring Tools to Encourage Learning by Doing

From the exploratory analysis, we found that multi-stage tutorials contain elements that make it easier for learners to try out the code, which include starter code, full source code, and a live playground. Learners can better understand and follow the tutorial by replicating and tinkering with the stages. However, we observed that most live code playgrounds embedded in tutorials are only provided for the last stage. Learners have to follow the stages and can not flexibly skip stages to explore a stage of their interests. This indicates a design opportunity for tutorial authoring tools to enable learners to run and edit stages easily.

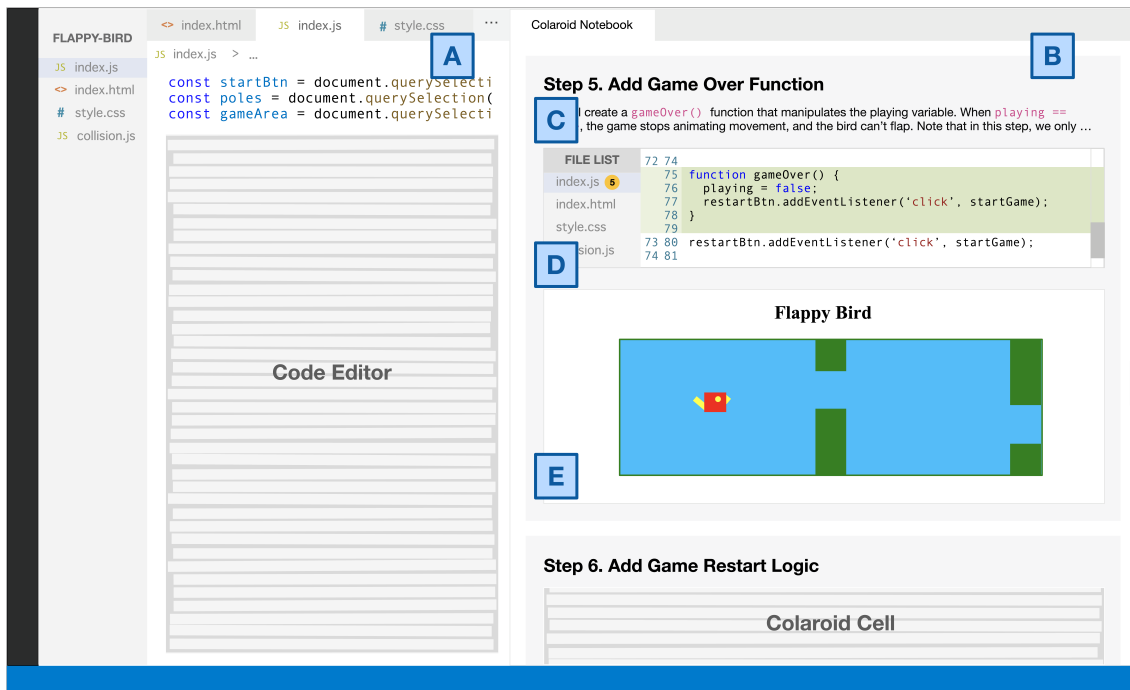


Figure 9.1: An overview of Colaroid. Colaroid is implemented as a VS Code extension. The user can open the Colaroid Notebook (B) side by side with their main code editor (A). A Colaroid notebook consists of cells. Each cell captures a history state of the codebase, which contains three components — a text annotation area explaining the rationale behind this state (C), a code editor area displaying the state of the code and highlighting the changes compared to the previous state (D), and an output area rendering the HTML display of the history state (E).

9.3 System Design

As our formative studies show, tutorial readers benefit from tutorials that are easy to distribute and enable them to skim the contents of the tutorial while providing enough detail to revisit individual steps in depth. We designed *Colaroid* to help authors easily create tutorials with these properties.

9.3.1 Illustrative Scenario

To illustrate the design of Colaroid for documenting incremental code construction and sharing tutorials, we will use a hypothetical scenario. *Alice* is a tutorial author who wants to create and distribute a tutorial that describes how to build an HTML-based “Flappy

Bird”² clone. We will also follow *Bob*, one reader of Alice’s tutorial. We will use Alice and Bob to illustrate the features of Colaroid in the following subsections.

9.3.2 Overview of Colaroid Notebooks

As shown in Figure 9.1, every Colaroid notebook exists as part of a larger codebase. Specifically, we implemented Colaroid as an extension for Visual Studio Code (VS Code), which is currently the most widely used IDE according to a recent survey³. This helps optimize Alice’s authoring experience by allowing her to write a tutorial while staying within her authentic development context. We implemented and tested Colaroid in the context of web programming (due to its ubiquity) but its design could be easily adapted and expanded to more languages and paradigms.

To start writing her tutorial, Alice first opens her VS Code editor and creates a new directory containing a HTML file with several starter lines, as she would do if she were writing this code outside the context of a tutorial. To create a tutorial, Alice opens the Colaroid tutorial authoring side-panel (shown in Figure 9.1.B) from the VS Code menu bar. The Colaroid panel is adjacent to Alice’s regular code editor (Figure 9.1.A). In this panel, Alice sets the tutorial title and adds a short description of the tutorial in natural language and Markdown.

9.3.3 Cells as Steps in Colaroid

In Jupyter (and most other computational notebooks), code is divided into “cells” where each cell usually represents a single conceptual block. For example, a cell might contain all the code responsible for compressing all of the data that another cell produced. However, this conceptualization of cells is a poor fit for interactive web applications, like the “Flappy Bird” game that Alice is building. This is because web applications rely on event listeners and callbacks, which often results highly inter-dependent “spaghetti code” [120]. As a result, the implementation of a single behavior might be split across many places in the code and difficult to isolate into a single cell. This can be particularly challenging in web programming, which relies on three separate languages (HTML, CSS, and JavaScript) to perform different functions on the same UI elements. For example, the code responsible for properly displaying an element might consist of HTML to define the

²https://en.wikipedia.org/wiki/Flappy_Bird

³<https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment>

content of that element (which needs to be placed in the appropriate part of the larger document), CSS to specify its appearance (which is typically in a different file), and multiple distributed segments of JavaScript (which is subject to the aforementioned spaghetti code phenomenon) that describe its dynamics.

Further, the order in which tutorial authors may want to explain their code often does not match the order of the code itself. It can be more intuitive to explain a code base through a description of components that are connected either conceptually or by their runtime behavior instead of through a line-by-line discussion from top to bottom.

To address these challenges, we re-conceptualized “cells” in Colaroid in a way that would allow them to represent code distributed across multiple locations, in any order. Rather than representing the code itself (which is often impossible to group into one cell) cells in Colaroid contain “pointers” to regions of code in the larger codebase context. More specifically, these pointers reference code *edits*—insertions and deletions in the larger codebase—that explain a part of the resulting code. In the context of tutorials, each cell typically represents a step in the tutorial. In other words, a Colaroid cell represents a historical state of the programming process.

Every cell contains three components to make the historical state that they represent more understandable for readers: a text area to describe the explanations and rationales of the code in this state (Figure 9.1.C), a code preview area showing the state of the code and highlighting the changes from the previous state (Figure 9.1.D), and an output area where the code in this state is rendered as a live HTML preview (Figure 9.1.E). For example, Alice might create a cell describing how to add a “Score” indicator that points to: 1) HTML code that defines its content, 2) the portion of CSS that specifies its font and size, 3) the JavaScript code that updates it to add to the score when the user avoids a boundary, 4) the JavaScript code that resets the score when the user starts a new game, etc. Alice might augment that cell with a brief description of what the code does and illustrate its effect by recording an example game session that Bob and other tutorial readers can replay, see, and interact with. The following sections will describe how Alice does this in more detail.

9.3.4 Authoring Tutorials by Documenting Incremental Changes

Colaroid optimizes the authoring experience by allowing the authors to write these tutorials while staying within their authentic development context, capturing the context and highlighting the changes with minimal effort, and enabling rich editing and styling on the

tutorials. Colaroid automatically tracks the changes across multiple project files and displays the code preview and the output preview under the explanations. This means users can progressively author the first draft of the tutorial as they construct the program.

After Alice initializes the tutorial, she begins to write code to create the central ‘bird’ character. In her code editor, she adds a `<div>` HTML element in the HTML file, writes CSS code to specify the bird’s size and color, and references the CSS code from within the HTML file. After testing the page locally, Alice decides to pause here to create a cell in Colaroid. The new cell automatically references Alice’s edits and Alice can add a more detailed explanation of her changes in Markdown.

9.3.5 Recording Interactions in Output Widgets

The impact of code changes on the output might be not straightforward to observe. Natural language explanations might be helpful but insufficient to understand the code changes in a cell. It can instead be helpful to have a chance to *see* and *try* the resulting program, particularly for UI code that reacts to user input. Colaroid cells contain an “output” widget that display the UI at the historical state represented by that cell.

However, the specific part of the output that readers should focus on might not be readily apparent. For example, a tutorial author might write a cell containing code that reacts when the user hovers over a given element. The effect of these changes will not be apparent until a reader interacts with the output in the correct way. Thus, Colaroid allows tutorial authors to optionally record example interactions (e.g., typing or clicking on UI elements). These example interactions are automatically replayed for tutorial readers.

For example, suppose that after Alice writes the aforementioned code to create the ‘bird’ character, she decides to implement the *interactive behavior* of the bird character. Alice creates a JavaScript file and links to it from her HTML code. In the JavaScript file, Alice writes code to make the bird jump when users click the screen. Alice makes this a new step in the tutorial. When she does, Colaroid displays an output preview where Alice can interact with all of the code up to that cell. As Alice clicks the output preview widget, she is able to see the bird moving. To make the effect of her code changes more apparent for readers, Alice records her interactions. Alice is able to replay her interactions by clicking on the ‘play’ button and can re-record as necessary. Readers like Bob can replay these interactions and experiment with the code and output at this (and every) step.

9.3.6 Revising and Editing Colaroid Notebook Cells

In our pilot studies with early prototypes of Colaroid, participants expressed the importance of correcting errors in post-editing. Colaroid supports a variety of post-editing with the draft tutorial, including editing text explanations, editing code, and annotating outputs. For text explanations, users can toggle the display into a Markdown editor and make changes to the content. We implemented Markdown styling with Colaroid, which could be easily extended into a rich text editor for editing and styling the explanations. In terms of modifying code, Colaroid allows users to zoom into a particular cell by restoring the local files into the state of the cell and directly making changes from the main code editor. In order to keep the edits synchronized [60], we chose to propagate these changes to the follow-up cells.

Thusfar, Alice has created three cells in Colaroid. However, she realizes that she forgot to change the HTML page title in the initial step, as shown in Figure 9.2. Alice can edit the initial step in Colaroid by clicking the ‘revise’ button, which then restores the state of every file in her codebase for that step in the tutorial. Alice then fixes her mistake by adding a page title and saving her changes. Colaroid automatically propagates her changes to later steps, which means the page title in steps 2 and 3 are also updated.

9.3.7 Sharing and Distributing Tutorials

After creating the tutorial, authors can share the entire project folder with learners so that they can open the tutorial in their own code editors. Authors can also export the tutorial into hosted webpages and static PDFs. Below, we explain how tutorials are stored, and describe several ways to share and distribute tutorials.

9.3.7.1 Leveraging Git for Code Versioning

Colaroid stores code changes by leveraging the git version control system. Additionally, Colaroid creates a JSON dictionary for storing the tutorial information, including the mapping between the code commit identification, the text annotations and the output interaction recordings. Thus, authors can directly pass the project folder to learners in order for them to open the Colaroid notebook in their own editor. Future front-ends for Git repositories (e.g., GitHub and GitLab) could easily add native support for Colaroid tutorials.

9.3.7.2 Sharing through Cloud Platforms

Alternatively, authors can also host their project folders through repository hosting platforms (e.g., GitHub, Bitbucket). When learners download the project code, they can access the Colaroid narrative by clicking the “Open Colaroid Notebook” option in the editor’s menu. With cloud computing environments that connect to these repository hosting platforms (e.g., GitHub Codespaces), learners can directly play around with the narrative in their browser. This approach overcomes the burden of cloning the project and opening it in their own editor, which requires Colaroid to be pre-installed.

Alice completes her tutorial and polishes it by fixing issues in intermediate steps and adding interaction recordings. Alice uploads her project directory to GitHub, a code repository sharing site. At some later point, Bob finds Alice’s tutorial and decides to open the GitHub Codespace to view it on a cloud-hosted VS Code editor.

9.3.8 Reading Tutorials

Colaroid enhances the experience of reading tutorials by encouraging learners to actively play around and explore the intermediate steps. In addition, it can render tutorials into several different formats according to learners’ needs.

9.3.8.1 Explorable Explanation

Inspired by computational notebooks, Colaroid allows learners to freely explore the intermediate steps to engage with the narrative. This way of learning concepts through live, interactive, and reactive environments has been characterized as “explorable explanation” [176]. We design two mechanisms to support the explorable explanation. First, when skimming through the narrative, learners can play with the live preview of the output or watch the recorded interactions to get a better sense of what the current progress is. Next, if they are interested in exploring an alternative solution, they can load the state of the cell into their main code editor and tweak around with it.

9.3.8.2 Rendering Notebooks into Multiple Formats

Colaroid notebooks can be rendered into multiple formats according to users’ needs. In addition to the default article view, learners can browse the steps in a “slide view”, which gives them the focus on a particular step. They can also browse the steps in a “timeline view”. As the learners move the progress bar, the state of the intermediate code will be

loaded to the main editor, which provides learners with a guided tour around the construction of the program.

After Bob finds Alice’s tutorial, he quickly skims through the initial steps in the ‘article’ view. Bob skims through the text annotations and highlighted code changes to skip steps when he already feels comfortable with the code changes. In addition, Bob can also visually observe the output of each step to gain an intuitive understanding.

At step 4, Bob notices that the tutorial describes the ‘repeat’ option in CSS. This is the first time Bob has heard of this property so he wants to do a deep dive and explore what things look like with alternative values. Thus, Bob clicks on the step to open the state of the codebase at that step in the VS Code editor. The state of the project is temporarily restored to that of step 4. Bob is able to browse and directly modify the code to experiment with its output. After exploring step 4, Bob can continue to read through the notebook and explore other steps.

9.3.9 Implementation

We implemented Colaroid as a VS Code extension so that we can enable the interaction between the tutorial panel and the code editor. In this section, we highlight the important implementation choices for Colaroid.

9.3.9.1 The Notebook View

The front-end component of Colaroid is built with the VS Code Extension Webview API, which renders HTML content and passes messages between the editor and the extension.

9.3.9.2 Mapping Git Commit with Tutorial Cells

Colaroid uses git to manage code versioning and editing and a separate JSON file (`.colaroid.json`) to determine how cells are rendered. Using git allows Colaroid to leverage a robust, widely-used versioning tool. For every cell in the notebook, `.colaroid.json` stores: `message` (the Markdown text of the step), `hash` (the corresponding git commit hash for the step), and `recording` (recorded interaction—mouse movement, clicks, etc.).

We chose to augment git with a separate data file (`.colaroid.json`) for several reasons. First, this structure allows us to easily remove a step without having to remove the commit. Second, we did not directly store the explanation as the commit messages for

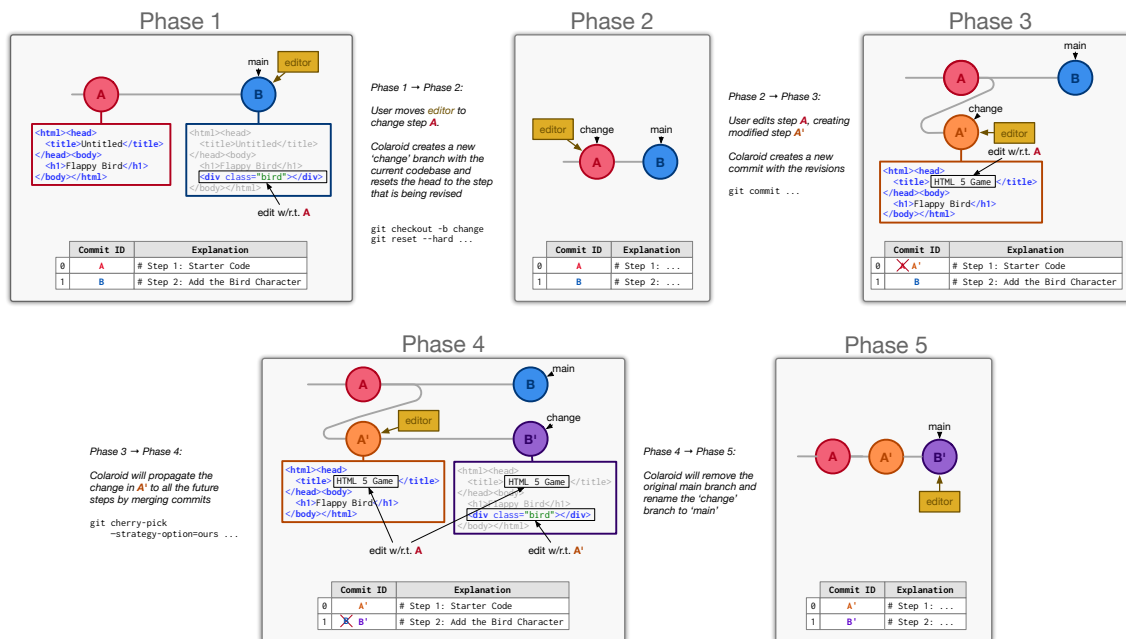


Figure 9.2: Colaroid implements code versioning and change propagating through git. Suppose the author wants to edit the first step (e.g., changing HTML page title) and propagate the change to subsequent steps. Colaroid maps steps with the hash ID of the code commits in Git. As shown in phase 1 and phase 2, Colaroid will first check out the main branch into a new branch named “change” and reset the head to the code version that needs to be edited. By doing this, authors would see commit A loaded in their code editor. Next, tutorial authors can make changes directly in the code editor. Once they confirm finishing the edits, Colaroid will create a new commit for the changes and merge commits in the later steps into the change branch.

easy modification and unlimited word length. Lastly, this approach also allows us to store additional annotations like interactive recording data with each step.

9.3.9.3 Propagating Changes

As Figure 9.2 shows, Colaroid leverages git to implement code versioning and change propagation. When a user clicks on the ‘edit’ button of a code snapshot, Colaroid checks out the current branch into a temporal branch, and reset the head to that code snapshot (say commit A). This loads the code snapshot into the user’s editor and allows them to make changes. After the changes are done, the user would click on the save button, which triggers Colaroid to create a new commit in the temporal branch (say commit A'). Next, Colaroid starts the change propagation process. Colaroid loops through all

the commit hash IDs for commits in the notebook JSON file. For each commit (say commit B), Colaroid would execute the `git cherry-pick` command with the merging strategy to be ours. This command would attempt to automatically merge the two commits (commit A' and commit B) and pick the original commit (commit B) if the changes can not be propagated automatically. After merging, we now get a new commit (say commit B'). Colaroid then updates the mapping table from [commit A, commit B] to [commit A', commit B']. The underlying mechanism of `git cherry-pick` is implemented using a three-way merge algorithm, similar to [60]. We choose to leverage `git cherry-pick` because it is a standard and widely-used implementation.

9.3.9.4 Recording Interactions in Output Snapshot

Colaroid allows users to record interactions to demonstrate behaviors in an output snapshot. The output snapshot is implemented as an `<iframe />` element that renders the code snapshot. We implemented the interaction recording by injecting a tracking script inside the `iframe` that captures mouse and keyboards input. This input data is timestamped and stored in the notebook JSON file.

9.4 System Evaluation Overview

To evaluate how Colaroid supports documenting incremental code construction, we designed two studies to investigate the authoring experience and the reading experience of Colaroid. To evaluate the authoring experience, we recruited instructors and senior students to create web programming tutorials from given topics using Colaroid. We summarized how they perceive the usability of Colaroid and how they compare Colaroid with other tutorial authoring tools. To further investigate the benefit of the Colaroid narrative in communicating the code construction process, we deployed Colaroid tutorials in a web programming workshop where students learned new programming concepts through project-based examples and applied them to a different context. We reported how students use and perceive Colaroid tutorials differently from a traditional static article tutorial and a video walkthrough.

Table 9.1: For study 1, we recruited 10 teaching assistants and senior students who are experienced in web programming.

PID	Background	Web Prog. Exp.	Teaching Exp.	Tutorial Topic	Tutorial Length	Word Count
I1	Ph.D. in CS	6 Years	Teaching Assistant	Lottery Number Generator	16	463
I2	Ph.D. in CS	15 Years	Teaching Assistant	Counter	4	98
I3	Master in IS	4 Years	Tutoring	Number Guessing Game	4	485
I4	Senior in IS	2 Years	Tutoring	Bootstrap	5	102
I5	Master in IS	4 Years	Teaching Assistant	Counter	7	178
I6	Ph.D. in CS	2 Years	None	Counter	5	30
I7	Master in IS	3 Years	Tutoring	Lottery Number Generator	7	152
I8	Master in CS	2 Years	None	Counter	10	237
I9	Master in CS	3 Years	None	Calculator	13	1063
I10	Master in IS	8 years	Teaching Assistant	Todo List	7	219

9.5 Study 1: Evaluating the Authoring Experience

To evaluate the authoring experience of Colaroid, we conducted a user study with 10 experienced web programmers where participants are asked to create a project-based tutorial using Colaroid. The scope of this study is to focus on the authoring experience from the tutorial creators’ perspective, instead of the learning experience from the learners’ perspective. More specifically, we aim to explore whether tutorial authors find Colaroid easy to use, and understand its usefulness compared to their prior experience in creating programming tutorials.

9.5.1 Method

9.5.1.1 Recruitment

We reached out to both instructors and senior students from the computer science program and the information science program on campus. We asked participants to fill a screening survey to indicate their prior experience in programming and teaching programming. Qualified participants identified themselves as experienced web programmers — including instructors, teaching assistants, or senior students who have previously taken an advanced web programming class or believe that they have equivalent skills. In total, we recruited 10 participants (9 graduate students and 1 senior undergraduate student). As shown in Table 9.1, their experience in web programming varies from 2 years to 15 years.

9.5.1.2 Study Task

Each study session consists of four components — a training component, a warm-up task, a freeform exploration task, and a post-task discussion. When participants joined the study, we first provided them with a 15 minutes training on how to install and use Colaroid. To ensure that they get enough practice of using Colaroid, we asked them to perform some exercises in a pre-made Colaroid tutorial. Participants were given a tutorial on the topic of creating a stopwatch. Participants need to make a few edits using the core features of , including making changes to a markdown text to explain a concept, making changes to a previous step, recording an interaction, and building an additional feature in the application while making it a new step. We encouraged participants to ask any questions about the usage of the Colaroid notebook. The warm-up exercises last for 15 minutes.

Then, participants completed a 30-minute open-ended authoring task. In this task, a participant created a first draft of a tutorial describing the construction of a simple web application. Participants could write about any web application they wished. To help participants pick a focus, we provided examples of web applications they could focus on, including counters, TODO lists, and lottery number generators. These recommended applications were chosen to be simple enough to implement in the time given, yet just complex enough that the tutorials would be interesting.

Participants were asked to write for an envisioned audience of students who have just started learning HTML, JavaScript, and CSS. They were asked to add commentary to their tutorial that described both the functionality of the code, and engineering considerations, like the rationale behind requirements, and how to debug the code. The amount of time allotted for the task was sufficient for creating a tutorial with several steps, and draft text commentary. Full drafts of text commentary and polish were considered outside of the scope of the task.

After the study session, we asked participants to complete a questionnaire about the usability of the tool. We also asked participants several semi-structured interview questions to probe into their feedback. In addition, we requested participants to upload their tutorials for additional analysis.

Each study session lasts around 80 minutes. All the sessions were conducted virtually with participants using Colaroid from their own VS Code editors and sharing screens through video conferencing tools.

9.5.2 Results

9.5.2.1 Overall Quality of the Tutorials

As shown in Table 9.1, all the participants were able to create a complete tutorial in Colaroid in the 30 minutes freeform exploration session. The topic of the tutorial covered a diverse range, including lottery number generator, number guessing game, calculator, todo list, and so on. We examined the tutorial artifacts and found that participants used different strategies to scaffold the steps (as referred as a base unit of the Colaroid notebook) of the tutorial. For example, I2, I6, and I8 all created tutorials on building a counter. I2 included 4 steps in the tutorial: setting up boilerplate code for the counter project, adding all the UI elements, programming interactive behaviors, and adding the style of the UI elements. I6 divided the steps by features with each step implementing a different button. I8 provided more detailed steps on how to link to external stylesheets and JavaScript files, how to get DOM elements in JavaScript, and how to respond to users' interactions. Regardless of the scaffolding strategies, Colaroid ensures that the code context of each step is captured and organized together into a single narrative. In addition, we observed that participants use Colaroid to create different types of explanations. Some participants (I2, I5, I6, I9) simply explained the purpose of the code changes to each step — what they did. Others also included various pedagogical instructions. For example, I4 added many reference links to the Bootstrap API as he went through example UI components; I9 created a fully explained tutorial (1063 markdown words in total) which not only covers what he did, but also how he did it and why he did it in styled markdowns. From the post-task questionnaire, most participants (8 out of 10) are satisfied with the tutorials they created. Several participants (I4, I8, I10) mentioned that if given more time, they would like to polish the explanations and add more external references, though the first draft is “good enough for capturing the process” (I4).

9.5.2.2 Easy to Author

Next, we examined how participants perceived the authoring experience in Colaroid. In the post-task questionnaire, most participants found the system not difficult to use (9 out of 10) and easy to learn (10 out of 10). Participants commented that the interface is “intuitive” (I1, I6, I7). Participants highlighted two features that improve the authoring experience — propagating changes from editing previous steps (I1, I4, I6, I9), and recording interactions on the output (I3, I4, I5, I9). For example, I1 and I6 mentioned that they

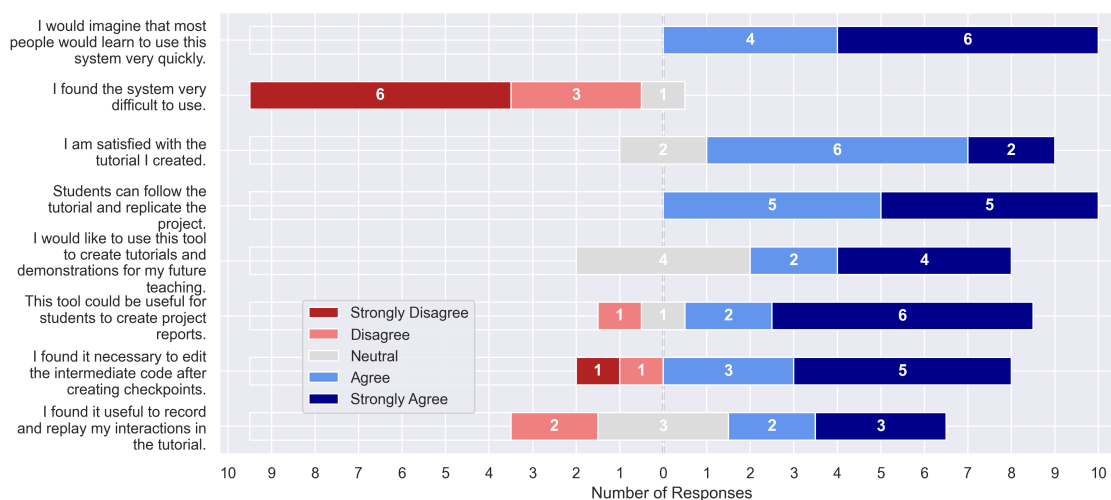


Figure 9.3: Results of the post-task questionnaire in study 1.

may want to later polish the code or fix mistakes in previous steps; I3 mentioned the benefits of recording interactions:

I really like how the recorded interaction would track your mouse. I have worked on similar tutorials before for react components. It is not always super apparent to learners on what happened to the output. (I3)

Several participants (I1-3, I6-8, I10) who have used Jupyter notebooks mentioned that the Colaroid notebook reminds them of Jupyter notebook and it is easy to understand the idea behind Colaroid. Participants also mentioned the differences between the two notebooks:

This reminds me of Jupyter notebook where you can use it to teaching things progressively and see how things are worked through. But with Jupyter Notebook, cells do not usually build on top of each other. Sometimes you can mess up with the notebook by executing the same cell multiple times or revise a previous cell and rerun it. I think Colaroid captures the process more honestly. (I3)

In the post-task questionnaire, we probed into participants' prior experience in demonstrating a coding project to others. Some participants mentioned live demo in classroom (I1, I2) or remote sharing (I3) and pointed out two issues with live demo: difficult for async setting — “may have different schedules” (I3), and hard to archive — “depends on the students in terms of how they take notes about the process” (I2).

Other participants mentioned their prior experience of authoring article tutorials and video tutorials and compared it with Colaroid tutorials. Participants reported several challenges of authoring article tutorials. For example, I2 reported the challenges of switching context and interleaving the development context when creating article tutorials — “I prefer creating tutorials directly inside the VS Code editor because I don’t have to go back and forth between different authoring tools.” (I2); I6 mentioned that it is a tedious process to supplement all the details in an article tutorial — “I really like to attach screenshots. But in a Medium post, I am not going to screenshot everything.” (I6); I7 said that making an engaging web article is technically hard — “With web articles, I think the biggest problem is that it is hard to create interactive elements in it. Some people are able to make very fancy web articles. But it takes efforts you know” (I7). For authoring video tutorials, participants reported the difficulties in post-editing:

In the past I have had to author documentation videos where I am recording myself going through things step by step for future programmers. But the problem with the video is the editing process. If I made a mistake, if it is just a word cut or something, I will start over and continue on. If it is something I realized later on, I will probably have to go through the entire process again.
(I9)

9.5.2.3 Perceived Benefits for Learners

Lastly, participants made several comments on how they think the Colaroid tutorials will benefit learners. We categorized the feedbacks into two aspects: potential usage scenarios and advantages over other tutorials.

For potential usage scenarios, most participants mentioned the Colaroid can be useful for instructors to deliver demonstrations to students. For example, I5 mentioned creating lecture notes in Colaroid to make students “easier to follow along in the class.”; I2 and I10 mentioned that students would “get a better sense of the flow by seeing the intermediate process”. Participants also mentioned that Colaroid can be useful for students to handle their assignments, which helps instructors understand “how they scaffold the project and why they do certain things” (I4). In addition, several participants mentioned using Colaroid for collaboration:

This tool can be potentially useful for collaboration. Consider working with massive number of people on an open-source project, the documentation is

very important. It can be helpful to explain decisions regarding each steps.

(I1)

Participants also solicited the advantages of Colaroid over other tutorials from learners' perspective, including capturing all the implementation details compared to article tutorials (I3, I5, I6, I8), encouraging learning by doing rather than passive reading (I2-3, I6), and less time consuming than video tutorials (I4-7, I9, I10). These results correspond to our findings in study 2 on the reading experience of Colaroid. Since this is not the focus of study 1, we will elaborate on these advantages later in the results of study 2.

9.6 Study 2: Evaluating the Reading Experience

Colaroid introduces not only a novel way of authoring tutorials, but also a new approach to interact with tutorials. Thus, we conducted a second study to evaluate how the affordances of Colaroid influence the experience of following a tutorial. In the second study, we provided learners with expert-created tutorials and asked them to apply what they learn into a new problem context. We compared Colaroid tutorials to two baseline formats of tutorials — text articles, and video tutorials.

9.6.1 Method

9.6.1.1 Recruitment

The study takes place as part of an advanced web programming workshop. The topic of the workshop is building HTML5 games, where the target audience are students who have basic knowledge of HTML5, but have never programmed HTML5 games before. We reached out to students who are currently taking or previously took the web programming class from our institution. In total, we recruited 16 participants for the study. All the participants had formally taken classes on web programming, and none of them had programmed HTML5 games before.

9.6.1.2 Study Setup

The study consisted of two sessions over a span of two weeks. The final project of the workshop is to build a dinosaur adventure game in HTML5. We provided participants with a set of tutorials on building a Flappy Bird game in HTML5. The dinosaur adventure

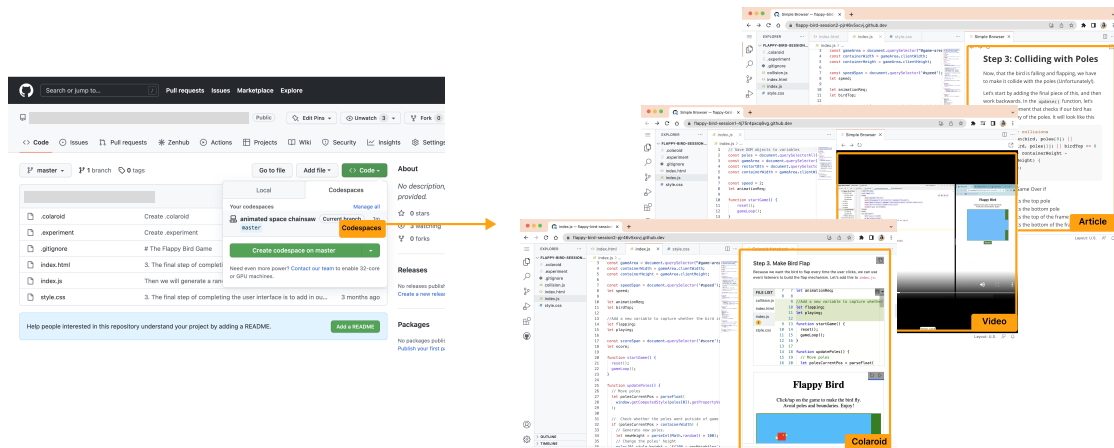


Figure 9.4: We used GitHub Codespaces for participants to access tutorials. All the three types of tutorials are displayed side by side with the main code editor.

game in the final project and the Flappy Bird game in the tutorial use similar APIs but are different in several mechanisms. We purposely made the final project challenging so that participants can maximally utilize the tutorials to help them accomplish the goal. We later validated the task difficulty with experts evaluating the project submissions and found that half of the participants were able to satisfy 60% of the final requirements.

The final project is scaffolded into two subgoals. In week one, students were asked to implement the layout and basic animation of the game. In week two, they finished the rest of the game by making the game interactive with users' input. We scheduled a 60-minute individual session each week with each participant to observe how they interact with the tutorials. In the first session, we provided 10 minutes of training on how to use the tutorial environment. Participants then spent 40 minutes exploring the session goal. After each session, participants were asked to complete a questionnaire asking them to assess their experience of following along with the tutorial. Lastly, after the second session where participants have experienced both conditions, we conducted a reflective interview for comparing the tutorials. All the sessions were conducted virtually using a video conferencing tool. Participants were explicitly told not to work on the game outside of the study session.

Our study used a within-subjects design where participants were given the Colaroid tutorial and one of the traditional tutorials. We counterbalanced the order of the tutorials. More specifically, there are 4 participants in each unique combination of conditions (Colaroid + article, article + Colaroid, Colaroid + video, video + Colaroid).

9.6.1.3 Study Apparatus

We used GitHub Codespaces for participants to access the tutorials. GitHub Codespaces allows participants to view the Colaroid tutorials in an online VS Code editor which has Colaroid installed. The online VS Code editor is connected to a virtual machine, thus, participants could edit, run, and test the code as if they are doing it locally. We hosted the tutorial projects on GitHub, and created the codespace instance ahead of time. We chose to use GitHub Codespaces because it simplifies the process of sharing the project, installing the Colaroid extension, and setting up the study environment on participants' local editors. It also avoids inconsistent versions or any incompatibility issues on users' local editors. To ensure participants have a similar experience in viewing tutorials and project code, we embedded the article tutorial and video tutorial inside the code editor. As shown in Figure 9.4, all three types of tutorials are shown side by side with the main code editor. Participants can open them in new tabs if needed.

9.6.1.4 Tutorial Preparation

We asked the two instructors of the web programming class to prepare the tutorials. The two instructors are familiar with participants' web programming experience and therefore can create instructional materials that best suit their learning. Each instructor was responsible for creating a set of tutorials for one session. Each set of tutorials contains the same instructional content in three forms — article, video, and Colaroid. For Colaroid tutorials, we provided instructors documentation on how to use Colaroid for creating tutorials. For article tutorials, instructors used a document editing tool (Dropbox paper) for creating styled texts with screenshots. Instructors can also include external links to provide more context for the tutorial. For video tutorials, instructors used a screen recording tool to demonstrate how they build the application while talking over the video to provide explanations. Instructors also did some post-production edits such as cutting and adjusting the speed. Instructors are explicitly told to create the best version of the tutorial they could and to make sure that the three types of tutorials convey similar instructional content. The research team further helped instructors edit the tutorials by fixing typos and styling issues, improving the quality of the screenshots, making sure the contents are reasonable and approximately the same quality across three formats.

9.6.1.5 Data Collection and Analysis

This study collected data from multiple sources. First, we collected students' background information on their familiarity with web programming, HTML5 game programming, and the VS Code editor. This data is used in screening the participants so that participants meet the same criteria for recruitment. For each session, two members from the research team were present and took observational notes individually. After discussing, synthesizing, and iterating the observation notes, we created a code book on interesting behaviors that emerged from observations. One member from the research team further applied closed coding on the screen recording to understand how students interact with the tutorials. For students' final artifacts for both sessions, we asked two experts to rate the quality of their submissions. The two experts first discussed the rubric for grading the functionality of the game (e.g., giving 10 points if the game character reacts to users' keyboard interaction, an additional 10 points if the game character demonstrates a "jumping" movement, and an additional 10 points if the game character stops movements when running into a tree.), This analysis is to help us understand how the tutorial formats led to noticeably different programming outcomes. In addition, we asked students to fill out a questionnaire after each session and compared the questionnaire results for each session. As shown in Table 9.2, we divided the population into two groups — groups that use Colaroid and article tutorials, and groups that use Colaroid and video tutorials. For each group, we conducted a paired t-test to understand the significance between the Colaroid condition and the regular tutorial's condition. We also conducted an exit interview with participants after the second session where we asked them additional questions comparing the tutorials they have experienced in the two learning sessions.

9.6.2 Results

9.6.2.1 How do participants engage with the tutorials?

Firstly, we are interested in how participants engage differently with three types of tutorials. We consider learners' engagement with the tutorials beneficial to the purpose of learning, though the frequent use of a tutorial may actually slow them down. We probe into participants' engagement by coding and visualizing their interactions with the tutorial.

Figure 9.5 illustrates how participants switch context between the tutorial and their own project where the x-axis represents the entire experiment duration. All participants

spent the full duration of 40 minutes on the task. Although there is no significant difference in participants' self-reported feeling of engagement, we found that there was a significant effect for the tutorial type, with participants' actual engagement time with Colaroid tutorials more than video tutorials ($M=8.79$ mins, $SD=5.40$, $p<0.01$) and article tutorials ($M=8.76$ mins, $SD=2.69$, $p<0.01$). Participants mentioned that they benefited from tinkering the intermediate steps in Colaroid:

I feel more engaged because I can run the tutorial code for each step and change the code to see how it works. (P6)

I feel more engaged because there's literally a workspace in the Colaroid tutorial which allows you to work alongside it. (P20)

In addition, we counted the occurrence of switching between the tutorial and the project, and found that participants switched more frequently in Colaroid ($M=16.5$, $SD=7.46$) and article tutorials ($M=14.38$, $SD=8.75$) than video tutorials ($M=7.13$, $SD=4.67$). We further observed that many participants switched more frequently in Colaroid and article tutorials to either copy the example code or compare their own code with the example code. In the reflection interview, some participants mentioned that video tutorials are less engaging because "you can't do it on your own pace" (P4), "you can not copy the code and revise it" (P12), and "I don't have patience to watch them" (P2). In particular, we noticed that some participants (e.g., P20, P12) gave up on the video tutorials after watching a segment at the beginning of the study, and decided to only use the final code of the Flappy Bird game to help them implement the dinosaur game.

9.6.2.2 Are Colaroid tutorials easier to follow along?

As shown in Table 9.2, we found significant differences in how participants perceive the time costs to follow along with three formats of tutorials. On a scale of 5 where 1 is completely disagree and 5 is completely agree, video tutorials ($M=3.88$, $SD=0.99$, $p<0.05$) are perceived to take more time to read than Colaroid tutorials ($M=2.38$, $SD=1.41$).

When comparing Colaroid tutorials with article tutorials, many participants mentioned that showing the code difference and output preview saved them time in reading:

It saves time to only read the code diff, and the preview works great because I don't need to guess myself. (P6)

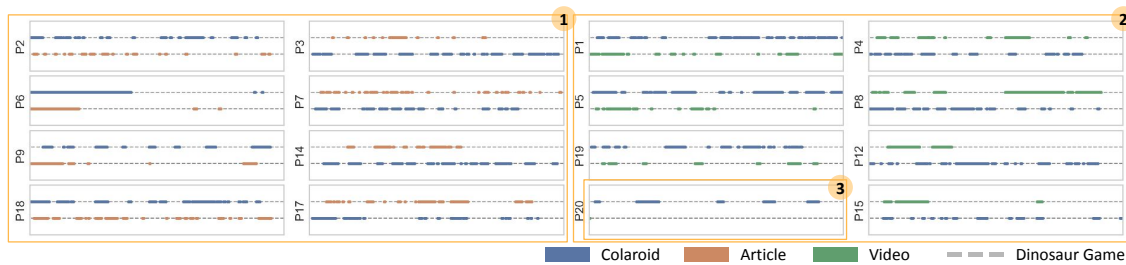


Figure 9.5: We manually coded the screen recording to understand how students interact with the tutorials. All participants used the full 40 mins on the task. Participants’ engagement time with Colaroid tutorials is significantly more than article tutorials (1) and video tutorials (2). In particular, we noticed that some participants (3) gave up on the video tutorials after watching a segment at the beginning of the study.

When comparing Colaroid tutorials with video tutorials, most participants mentioned that video tutorials are lengthy to watch and hard to navigate around:

The explanations in two tutorials (Colaroid and video) are both clear to me. It (Colaroid tutorial) is easier for me to find what I want, but in the video tutorial, I have to go over it and find what I need. (P1)

P3 recalled her prior experience with video tutorials:

When I first learn programming, I need to have at least two screens. I need to watch how professors do the coding, and I need to do it by myself. When I watch a video, I sometimes need to spend time understanding what the professor is talking about. So I have to press pause and read the code again. (P3)

In addition, some participants also mentioned the challenges in navigating between steps in video tutorials:

I like the Colaroid tutorial because I can skip around, look at the code, and play around with it for myself. I think I like to look at things twice over on and maybe read twice. With video, it’s pain to rewind and rewatch and rewind. (P15)

9.6.2.3 Does Colaroid support more incremental procedure following?

Next, we investigated how different tutorial modalities communicate the process. Participants complained that article tutorials were not good at tracking the process for several

reasons. First, article tutorials only showed a sliced range of the code snippets. We observed that a common pattern for participants to learn from a tutorial is by trial and error. Many participants copied and pasted the tutorial's code into their own projects to see how it applied to their scenarios. However, it is not straightforward for them to see where the new code should be pasted. In a step where the instructor inserts a statement into a declared function, some participants were confused about where to locate the newly added code and pasted it outside of the declared function. In the post-task questionnaire, participants perceived that it is clear to them how the steps evolved in Colaroid ($M=4.5$, $SD=0.73$) than in the article tutorial ($M=3.63$, $SD=0.52$, $p < 0.01$). For example, one participant reported:

I had a harder time with this one (article). I can't easily compare the steps. Although each step has a link to a github repo, having them open separately and not able to see the differences between the repo does make it a bit more difficult. And a lot of the article tutorials don't guarantee to have that. It was a lot easier to have the centralized space and to see changes between each step (in Colaroid). (P7)

Colaroid tutorials also provide a better translation of the process by showing how the step changes affect the output in the tutorial. We observed that when skimming the tutorial, many participants would try the intermediate output in the Colaroid tutorial to understand the outcome of the step and then decide whether they are interested to look into more details. One participant compared the fidelity across three modalities and rated Colaroid as between video and article:

I think it really helps to see someone do it and be able to understand how each step they are doing and make sure I understand how to replicate it. The only issue with video tutorials is sometimes that I don't have the patience to watch them because I read much faster than I can. So sometimes I prefer to skim an article. I think Colaroid is like between the video and article. I could skim through it, but I can understand the materials much more thoroughly and comprehensively like actually being able to watch someone go through every step of the process and explain it. (P2)

9.6.2.4 Does Colaroid lead to better learning outcomes?

In the post-task questionnaire, we asked participants to rate a few statements regarding to the task performance. As shown in Table 9.2, there is no significant differences in terms of participants' satisfaction to the final artifacts they built. In addition, we did not see a significant differences in terms of how experts' evaluation of the artifacts regarding to different formats of the tutorials.

Despite that there no evidence showing that Colaroid tutorials can significantly improve participants' learning outcome, several participants mentioned Colaroid encourage active learning: "I think I learned by doing. And having an interactive notebook is more suitable for that." (P20). One participant provided an interesting analogy of the learning experience provided by three tutorial modalities:

An analogy to that is like you are in a chemistry class or biology class, Colaroid is like the lab where you can quickly do something to a chemical experiment, where the video is like watching a lecture recording and the article is like reading a textbook. You will learn it but I felt like it's not as useful because you don't see what actually happens in the environment of your world.
(P19)

9.6.2.5 Summary of the Results

In summary, our evaluation shows that Colaroid provides a more engaging reading experience by following along the scaffolded steps with active exploration. Colaroid harnesses the advantages of article tutorials in terms of providing a self-paced and easy-to-skim reading experience. Colaroid also harnesses the advantages of video tutorials in terms of capturing the details and context for reproducing, supporting more incremental procedure following. Although there are no significant differences in the learning outcomes of Colaroid, participants perceive Colaroid to encourage active learning.

9.7 Discussion

This paper describes the design of the Colaroid system and demonstrates how literate programming principles can be applied to support the instruction of software packages and libraries. We contribute to the field (1) a novel design for creating step-by-step technical education content, extending the concept of literate programming into a temporal

dimension, (2) a system, Colaroid, which supports this design and is embedded within the software development tools commonly used by programmers, and (3) an evaluation of this system, looking at how both authors and learners would use it to create and understand learning resources.

By weaving together instructional narrative, code context, and output interactions we are able to support both the authors who create instructional content and the learners who aim to use it to improve their skills. Embedding this system within the authentic work environment for software developers – the integrated development environment – increases the engagement of learners with the instructional content, supporting an active learning experience.

Our design presents a new perspective on how literate programming [96] can be extended to support guided complex instruction. Many of the literate programming systems in wide use, such as the Jupyter programming environment [137] for computational notebooks, are “spatially-based”, where the narrative components are generally used to describe pieces of code in a top-down order through the document. Through the design of Colaroid we have introduced a new paradigm of “temporally-based” literate artifacts, where the code being constructed is described in an order in which someone would take to build a running system. The key difference between our design and existing systems [137, 18] is that each step in a temporal literate artifact is its own state, and the narrative, code, and runtime environment (e.g., web browser, or Python interpreter) for that state is unique. Users can navigate between steps, and doing so shows them the appropriate execution state and instructions for that step. There is no “run all cells” command or the like – a user simply chooses to inspect the last step of the narrative to see the final state of the system.

This design ties together narrative (instruction), code, and system state, and for learners it promotes both guided instruction as well as active learning. Colaroid expanded prior studies [141, 25, 200, 68] on linking tutorials with application state into the domain of learning web programming, revealing the benefits in both authoring in authentic environments and learning in authentic environments. Compared to other application domains (e.g., drawing, 3D modeling), we can leverage existing code versioning tools and sharing platforms for tracking system states of the target application — code editors. In addition, Colaroid is different from existing approaches to annotating and replaying application states (e.g., CodeTour [15]) as it generates the narrative for learners to skim through and learn at their own pace. Beyond supporting just the learner, this design also supports the

iterative authoring process of tutorial content, encouraging encapsulation of instructional explanations with the appropriate code state.

9.7.1 Outlook

The growth of zero install web-based integrated development environments supported by software code repositories (e.g., gitpod, GitHub Codespaces, Binder) offers a new opportunity to streamline education as it relates to the features and use of software packages. Many open-source software repositories already contain a directory of examples that are intended to go with web tutorial content. By leveraging the Colaroid system, these repositories could provide one-click learning opportunities, allowing users to go from the familiar source code version control interface into an authentic IDE with detailed instructional content. From within the browser, learners could immediately begin to explore both the code base and the runtime state of tutorials, engaging in active learning immediately. This also opens the potential for educational content to be woven into the software as a first-class artifact, by extending the continuous integration systems in place to produce regression tests against the individual steps of the educational tutorials. Such an approach would allow project communities to require that a software release include up-to-date educational examples of a given library, reducing inconsistencies between online tutorials and the libraries they aim to teach.

9.7.2 Limitations

9.7.2.1 Limitations of Colaroid

As presented, Colaroid is engineered for creating web programming tutorials specifically and the current output preview is limited to this task. There is opportunity to consider how the system might need to be changed in order to support other kinds of tutorial content. For instance, rendering Python output in the preview, or visualizing data changes [179], maybe a straightforward way to support data science programming instruction. It is possible that this approach might go beyond traditional programming as well. For instance, in the field of graphic design step-by-step tutorials are often used for instruction, and share many similarities to the web programming context we explored. It would be interesting to apply temporally-based literate techniques inside of a tool such as Adobe Photoshop, where the software code is replaced with layered images, and the narrative describes how tool functions are used to manipulate the images to achieve a desired ef-

fect. Being able to load a given image (state) and set of instructions may be particularly interesting to study as graphic design often includes a kinesthetic expression component (e.g., drawing) which may be strengthened through repeated practice.

Through use of the system it became clear that there were many additional supports which might be integrated to improve experiences for both authors and learners. For instance, voice dictation through speech to text for narrative portions of the tutorial would naturally fit instructional approaches such as lecturing or massive online courses. We limited our narrative component markdown text, the rough format used in online tutorials, but there are other media types which may be appropriate and further the authoring experience. In addition, Colaroid does not save learners' exploration of the steps, or compare them with the reference solution.

9.7.2.2 Limitations of our Evaluation

We chose to explore the Colaroid with an eye to both authoring and learning tasks. For the authoring study, only a small number of participants ($n = 10$) were observed, and they created small-size tutorials for simple web development tasks, and were not given explicit instructions or time for polishing of the educational content. This evaluation method allowed us to study their interactions in-depth, and follow up with interview questions to understand their thinking. However, a larger field trial of the tool would help to uncover whether tutorial style has an interaction on adoption and acceptance of the authoring experience. It would be especially beneficial if Colaroid was enabled for more programming domains as the interaction between domain, tutorial style, and the temporal literate programming design could be explored.

For the learner study, the tutorial topic (Flappy Bird) and the task topic (Dinosaur Game) only represent one usage case of tutorials — following a tutorial step-by-step to create a similar application. It is worth exploring different usage cases of tutorials, including replicating a project by following a tutorial, or learning key concepts as one might do in lecture content. Importantly, we did not measure learning gains which is one of the reasons users engage in consuming online tutorial content. The interactions between productivity, engagement, and long-term learning (versus immediate performance learning) are significant, and thus we make no claims here that Colaroid results in longer-term knowledge retention. However, we are excited by the increased engagement that learners experienced in the Colaroid condition, as there is significant evidence that active learning and hands-on practice does result in long-term learning gains [97].

9.8 Conclusion

This paper presents a literate programming approach to author explorable and multi-stage tutorials. We implemented a prototype Colaroid, an IDE-integrated tutorial editor that captures the scaffolded implementation process in the authentic work environment. On the other hand, Colaroid provides the IDE-integrated reading experience, allowing learners to explore and tinker with the steps in the tutorial directly in their authentic work environment. Our evaluation shows that Colaroid can benefit both the authoring experience by effective and rich editing, and the reading experience by encouraging learning by doing.

9.9 Acknowledgements

This material is based upon work supported by the National Science Foundation under DUE 1915515. We would like to thank Michael Nebeling for his feedback on the paper. We thank Yuyu Yang and Steven Nguyen for their help in collecting the programming tutorials. We thank the SI 579 teaching team for their help in the evaluation study. We also thank the study participants and reviewers for their time and effort. Finally, we thank Alexander Brooks for his art assets.

Table 9.2: Perceptions of the three tutorials. Participants rated their agreement with nine questions on a scale from 1 (strongly disagree) to 5 (strongly agree). (M: mean, SD: standard deviation).

Statement	Condition	N	M	SD	p	Agreement: 1 to 5
This tutorial is easy to follow.	Colaroid	8	4.13	0.64	0.17	
	Article	8	3.63	0.52		
	Colaroid	8	3.88	1.36	0.32	
Video	8	3.38	1.19			
It is clear to me how the steps evolved.	Colaroid	8	4.63	0.52	0.001**	
	Article	8	3.63	0.52		
	Colaroid	8	4.38	0.92	0.19	
Video	8	4.00	1.07			
It is easy to understand how the step changes affect the output in the tutorial.	Colaroid	8	4.38	0.74	0.07	
	Article	8	3.63	0.92		
	Colaroid	8	4.25	1.04	0.11	
Video	8	3.50	1.51			
This tutorial helps me with making progress on my dinosaur project.	Colaroid	8	4.50	0.53	0.10	
	Article	8	4.00	0.76		
	Colaroid	8	4.13	1.13	0.49	
Video	8	3.63	1.50			
After reading the tutorial, I am confident that I can replicate the Flappy Bird project from scratch by myself.	Colaroid	8	4.25	0.70	0.001**	
	Article	8	2.88	1.25		
	Colaroid	8	3.50	1.69	1.0	
Video	8	3.50	1.51			
After reading the tutorial, I am confident that I can build similar HTML5 games from scratch by myself.	Colaroid	8	3.50	1.20	0.04*	
	Article	8	2.75	0.89		
	Colaroid	8	3.00	1.69	1.0	
Video	8	3.00	1.60			
The tutorial takes too much time to read.	Colaroid	8	2.25	0.89	0.06	
	Article	8	3.25	0.89		
	Colaroid	8	2.38	1.41	0.04*	
Video	8	3.88	0.99			
I feel engaged when reading the tutorial.	Colaroid	8	4.13	0.64	0.04*	
	Article	8	3.50	0.75		
	Colaroid	8	3.88	1.25	0.49	
Video	8	3.38	1.50			
I am satisfied with the progress of the dinosaur game so far.	Colaroid	8	4.00	0.76	0.04*	
	Article	8	3.38	0.74		
	Colaroid	8	3.88	1.13	0.84	
Video	8	4.00	1.07			
Expert Evaluation on the Artifact	Colaroid	8	60.00	27.26	0.12	
	Article	8	54.38	32.12		
	Colaroid	8	58.13	35.75	0.81	
Video	8	63.13	35.25			
Actual Engagement Time (mins)	Colaroid	8	14.68	3.00	0.007**	
	Article	8	8.76	2.69		
	Colaroid	8	14.35	4.87	0.002**	
Video	8	8.79	5.40			

Category	Theme	Example	Tutorials
Scaffolding Strategies	Iterative Build-up: The current stage iteratively build upon prior stages where the changes can take place in a nested way or at multiple locations.	T1 first declared a function definition. It then wrapped the function into a class definition.	T1-6; T9-10; T15; T17-24; T26; T29; T31-36; T38-40; T42-44
	Module-based Build-up: The stages are divided into modules that are independent from each other. The order of the stages does not matter.	T25 has each stage implemented as an independent method in its own file.	T13; T16; T25; T27; T37; T41
	Aggregated Build-up: The newly added code can be linearly aggregated to the end of the previous code base. It is different from module-based build-up since the order matters.	T7 uses a Jupyter notebook styled format where each stage contains lines of code to be executed after previous stages.	T7-8; T11-12; T14; T28; T30
Composition of Code Snippets	Changes Only: The code snippets only present the changes.	T19 only present the changes in each stage and never duplicate the content of adjacent stages.	T5; T7-8; T11-15; T19-20; T23; T28; T30-31; T34-35; T37; T41-42; T44
	Full Context: The code snippet contains the complete context of the current code file.	Each stage in T4 shows the entire implementation of the related code file.	T1-4; T9-10; T15; T17-18; T21; T24; T27; T33; T36; T38
	Partial Context: The code snippet contains partial context of the current code file.	One stage in T6 involves changing a long xml file. The irrelevant part in the xml file is hidden.	T6; T21-22; T25-26; T29; T39-40
Presence of Code Snippets	Changes Highlighted: Changes are highlighted from the context.	T9 uses a darker background to highlight the changes in a stage.	T9; T17-18
	With Syntax Highlights: The code snippets have syntax highlights.	The code snippets in T6 have syntax highlights.	T2-4; T6-15; T17; T20-27; T29-35; T38-40; T42-43
	Context Locator: The code snippets use context locators (e.g., line number; file name) to indicate where the changes are.	T4 marks both file names and line numbers in each stage.	T4; T9-10
Presence of Intermediate Results	Textual Outputs: The textual output from the console	Several stages in T15 contain textual output.	T1-2; T7-8; T11; T14-15; T17-18; T24; T31; T41
	Screenshot or Video of Intermediate Output: The output preview for intermediate stages	T16 takes screenshot of the intermediate output.	T20-22; T26; T28-30; T32-36; T40; T42
	Screenshot of the Last Stage Only: The output preview for the last stage	T25 only has one screenshot of the final output.	T5; T15; T25; T38
	Textual Description: The textual description of the output	T13 describes what will happen after executing the stages.	T4; T6; T12-13; T20; T25; T27; T37; T43
	Attach a Working Demo: An interactive working demo	T3 embeds a working demo into the tutorial content.	T3-4; T9; T12; T23; T28; T30; T34; T36; T41
Learn by Doing	Starter Code: Providing the starter code for learners to follow	T4 links to a starter code at the beginning of the tutorial.	T4; T22; T33
	Full Source Code: Providing the full source code for reference	T6 attaches a link to the full source code.	T5-6; T9; T12-13; T16; T20; T24-31; T33-34; T36; T38-41; T43-44
	Live Playground: An online IDE hosting the full source code	T12 contains a link to a cloud-hosted Jupyter notebook	T4; T9; T12; T28; T30; T36; T38-39; T44

Table 9.3: Exploratory Analysis Results.

Part 4

Conclusions

CHAPTER 10

Conclusions and Future Directions

Data scientists must embrace collaboration to improve work efficiency. In the future, collaboration in a data-centric world will not just be a matter of fair for data science practitioners, but for everyone on any tasks. In this dissertation, I conduct mixed-methods inquiries to identify real-world problems that data science practitioners face with collaboration, as well as design and build novel interfaces for improving collaborative data science tools for productivity and learning. In this chapter, I review the summary of the contributions of each project and discuss future work.

10.1 Summary of Contributions

Through a series of mix-methods studies, tool buildings, and evaluations, this dissertation makes the following contributions towards interactive programming interface for data science collaboration and learning:

- In Chapter 2, I synthesize related work across HCI, Software Engineering, and Computer Science Education to motivate the topic of this dissertation. I summarize the design and revolution of computational notebook environments and identify the research gaps in studying the unique challenges and opportunities for data scientists and learners to collaborate with current programming platforms.
- In Chapter 3, I conduct mixed-methods inquiries to comprehend the specific obstacles that data scientists face while collaboratively using computational notebooks. This work has created a taxonomy of common collaboration styles in data science and identified several advantages and challenges with synchronous notebook editing. These insights lead to design implications to enhance collaborative program-

ming environments for data science learners and professionals, and serve as the basis for the tools proposed in the subsequent chapters.

- Drawing from the design implications discussed in Chapter 3 – specifically, that tools should aid collaborators in better comprehending the rationale behind a shared notebook – I developed Callisto in Chapter 4. This comes with a suite of features intended to enhance the utility of chat messages for understanding previous exploration processes within the notebook. Through an evaluation study, I provide empirical insights into user engagement with these features and their perception of them. Moreover, I present evidence demonstrating that establishing connections between messages, elements of the notebook, and versions assists data scientists in understanding and following up on the exploration pipeline.
- Conversely, to enhance the documentation of disorganized analysis notebooks composed asynchronously, I delve deeper into AI-assisted documentation in Chapter 5. This project provides empirical knowledge of best practices regarding how individuals document a notebook, garnered through an analysis of highly-rated Kaggle notebooks; showcases the design of a human-centered AI system — Themisto — capable of working in conjunction with human data scientists to produce high-quality computational narratives; and provides empirical evidence that Themisto can collaborate effectively with data scientists to generate high-quality computational notebooks that satisfy users’ needs, and does so in considerably less time.
- Chapter 4 and Chapter 5 investigate design approaches to help data scientists better understand code changes. In Chapter 6, I also contend that the comprehension of iterative data changes, produced by the code, should hold equal importance to code changes throughout an analysis. In Chapter 6, I highlight the advantages of using visualizations to emphasize data differences as a central feature within a data science programming environment. Additionally, the chapter uncovers insights into users’ needs and their utilization of both code and data differences during exploratory data analytic processes. This is based on a user study involving 16 data scientists.
- In response to the issue of conflict editing in a real-time shared notebook as discussed in Chapter 3, I introduce three mechanisms in Chapter 7, exemplified by the PADLOCK system. The PADLOCK system allows data scientists greater control over the visibility and editability of sensitive cells, as well as the runtime state of

shared notebooks. It also designates spaces where data scientists can formulate and share their unique ideas. Moreover, I present a series of evaluations to gain a deeper understanding of how these features can be used in collaborative data science.

- The expansion of the data science field has resulted in an increased demand for foundational education in programming and data science. However, the requirements and objectives for collaboration can vary greatly between professional data scientists and those within educational settings. In Chapter 8, I introduce a unique methodology for real-time code sharing in educational environments. I design PuzzleMe, a web-based platform that allows instructors to conduct in-class programming exercises by sharing code and descriptions in real-time. PuzzleMe incorporates two strategies—live peer testing and live peer code review—that facilitate peer assessment during in-class exercises, thereby fostering collaborative learning among students.
- Computational notebooks are frequently used to develop tutorials for data science students, but are limited in developing tutorials for other programming activities where the learners seek authentic programming experience in traditional IDEs. In Chapter 9, I propose an alternate design for temporal computational notebooks, enabling programmers to construct computational narratives on incremental code development. I implement this concept through building the Colaroid system and assess its strengths and drawbacks from both the authorship and learning perspectives.

10.2 Future Work

This dissertation contributes to the research vision to lower the barriers for users to collaboratively explore, understand, and communicate in a data-centric world. Moving forward, I would like to discuss three future research themes.

10.2.1 Understanding Heterogeneous Collaboration in Data Science

I envision that the workspaces of the future will persistently be a blend of physical and digital realms, perpetually encouraging a collaborative atmosphere. The role of technology in facilitating collaboration will be increasingly important, particularly with the advancement of computer-supported collaborative technologies. These technologies will

empowering data science practitioners, diverse stakeholders, and even emerging AI collaborations to work together harmoniously in a data-centric world. In my previous work, I have studied how collaboration takes place in various contexts, including real-time editing [181], multidisciplinary teams [138], knowledge sharing between domain experts and data scientists [132], reusing analysis code [55], peer collaboration in classrooms [177], and writing documentation together with AI agents [183]. Moving forward, I am looking forward to examining the broad spectrum of collaborative data science — for example, how to support collaboration between mixed synchronicity teams? How does mixed reality technology open new avenues of collaboration and communication?

10.2.2 Human-AI Systems for Data Science Programming

My prior work demonstrates how emerging AI technologies can help solve particular challenges in data science workflows such as incomplete documentation [183, 207]. In particular, I found that maintaining control of the initiative and the final decision is an important aspect of people’s enjoyment and acceptance of the AI system. I believe in the potential of building human AI data science systems that allow data scientists to maintain control. In the future, I will continue striving to explore this direction by collaborating with researchers from machine learning, NLP, and software engineering. For example, I am particularly interested in the application of AI pair programming and automated machine learning. How can data scientists leverage large language models for exploratory data analysis? How can we design interactions to improve the trust and explainability of the generated code from automated machine learning? To what extent can existing systems like GitHub Copilot understand domain-specific task descriptions? How can we design tools to support data scientists rephrasing their high-level domain-specific queries when things go wrong?

10.2.3 Making Data Science Accessible for Everyone

As the world becomes data-centric, the ability to work with data, understand data, and tell stories from data becomes a literacy for everyone. Just like computational literacy [182, 178], I believe that everyone should learn data literacy, and data science tools should be made accessible to everyone. My prior work explores several data science practitioner communities, such as the online forum Kaggle [183], or data scientists in software teams [55]. Looking forward, I would like to study learning resources for the general public

(e.g., online courses, and informal workshops) to gain data literacy and their effectiveness. Another area of improvement is to lower the barriers for non-programmers to use data science tools. Most data science tools are designed for people who are proficient in programming to work with data, but are not accessible to the general public. Looking forward, I am excited to explore how techniques like programming by demonstration, block-based programming, and domain-specific programming languages can build easy-to-learn data science tools for people who do not have programming backgrounds while enabling them to author expressive data stories.

APPENDIX A

Appendix for Chapter 7

A.1 Example of Documentation Generation in Themisto

Table A.1: Example notebook (House Price) (T - Markdown cells created by Themisto only, C - Markdown cells co-created by Humans and Themisto, H - Markdown cells created by Humans only).

Source Code	DL-Based	Query-Based	Prompt-Based	P2
<pre>import pandas as pd import numpy as np from sklearn.linear_model import LassoCV from sklearn.model_selection import cross_val_score</pre>	Importing libraries	Pandas is for data manipulation and analysis; NumPy is a library for ...	This code cell is for _ -----	Importing libraries (T)
<pre>train = pd.read_csv('train.csv') test = pd.read_csv('test.csv')</pre>	Read the data	Read a comma-separated values (csv) file into DataFrame; Return the first 5 rows.	This code cell is for _ -----	Read the data (T)
<pre>train.head()</pre>	Let's see the values	Return the first 5 rows	The table shows ___ --	Return the first 5 rows. (defValue=5) (C)
<pre>all_data = pd.concat((train.loc[:, 'SubClass': 'SaleCond'], test.loc[:, 'SubClass': 'SaleCond']))</pre>	A generator for feature	Concatenate pandas objects along a particular axis with optional set logic along the other axes.	This code cell is for _ -----	Concat train and test col "SaleCondition" (C)
<pre>all_data = pd.get_dummies(all_data)</pre>	Convert all the data	Convert categorical variable into dummy/indicator variables	This code cell is for _ -----	Convert categorical variable into dummy/indicator variables. (T)
<pre>all_data = all_data .allna(all_data.mean())</pre>	Check the missing values	Fill NA/NaN values using the specified method	This code cell is for _ -----	
<pre>X_train = all_data[:train.shape[0]] X_test = all_data[train.shape[0]:] y = train.SalePrice</pre>	Create the target and the test data	Slice string	This code cell is for _ -----	
<pre>model_lasso = LassoCV(alphas = [1, 0.1, 0.001, 0.0005]) .fit(X_train, y)</pre>	Model	Lasso linear model with iterative fitting along a regularization path.	This code cell is for _ -----	Fit regression model (H)
<pre>def rmse_cv(model): rmse= np.sqrt(-cross_val_score(model, X_train, y, scoring= "neg_mean_squared_error", cv = 5)) return(rmse) rmse_cv(model_lasso).mean()</pre>	A simple example model with the lasso	Evaluate a score by cross-validation	The result indicates that _ _ _ _ _	Define score function and evaluate (H)

Table A.2: Example notebook (Covid Prediction) (T - Markdown Cells Created by Themisto Only, C - Markdown Cells Co-created by Humans and Themisto, H - Markdown Cells Created by Humans Only).

Source Code	DL-Based	Query-Based	Prompt-Based	P5
<pre>import numpy as np import pandas as pd from sklearn.ensemble import RandomForestClassifier</pre>	Importing libraries	li- Pandas is for data manipulation and analysis; NumPy is a library for ...	This code cell is for _ _ _ _ _	Importing libraries (T)
<pre>train = pd.read_csv("train.csv") test = pd.read_csv("test.csv") train.head()</pre>	Read the data	Read a comma-separated values (csv) file into DataFrame; Return the first 5 rows.	The table shows _ _ _ _ _	Read and sanity check the data (C)
<pre>train.describe()</pre>	Let's see the values	Generate descriptive statistics. Descriptive statistics include ...	The table shows _ _ _ _ _	
<pre>train["Date"] = train["Date"] .apply(lambda x: x.replace("-", "")) train["Date"] = train["Date"] .astype(int) train.head()</pre>	Convert all the data	Replace a specified phrase with another specified phrase	The table shows _ _ _ _ _	Preprocess the data (C)
<pre>train.isnull().sum()</pre>	Check the missing values	Detect missing values for an array-like object	The result indicates that _ _ _ _ _	Check the missing values (T)
<pre>test["Date"] = test["Date"] .apply(lambda x: x.replace("-", "")) test["Date"] = test["Date"]</pre>	Convert all the data	Replace a specified phrase with another specified phrase	This code cell is for _ _ _ _ _	Preprocess the date column (C)
<pre>x = train[['Lat', 'Long', 'Date']] y = train[['ConfirmedCases']] x_test = test[['Lat', 'Long', 'Date']]</pre>	Create the target and the test data	Select subsets of data	This code cell is for _ _ _ _ _	Create the train/test data and the target (C)
<pre>Tree_model = RandomForestClassifier(max_depth=200, random_state=0) Tree_model.fit(x,y)</pre>	Model	A random forest is a meta estimator that fits a number of decision tree classifiers on ...	This code cell is for _ _ _ _ _	Define and configure the model A random forest is a meta ... We also train the model with '.fit()' (C)
<pre>pred = Tree_model.predict(x_test) pred = pd.DataFrame(pred) pred.columns = ["ConfirmedCases_prediction"]</pre>	Predicate to use a predicate function for tests	A random forest is a meta estimator that fits a number of decision tree classifiers on ...	This code cell is for _ _ _ _ _	Run the model to generate predictions on the test data and store them as a 'DataFrame'(H)

A.2 Coding Book for the Interview Transcripts

Table A.3: Coding Book for the Interview Transcripts

Theme	Code
Pros of Themisto	Easy to Use Provide Inspirations Improve Content Efficiency Hybrid Approach Useful for Long Term Prefer the Plugin
Cons of Themisto	Inaccurate Not Useful
Perceptions of the Deep-Learning-Based Approach	Concise Useful Accurate Inaccurate For Own Use For Collaboration Use
Perceptions of the Query-Based Approach	Descriptive Too Long Useful Confusing Instructive
Perceptions of the Prompt-Based Approach	Tedious Easy to Use Inspiring
Future Adoption	Positive Adoption Propensity Scenarios for Future Adoption Negative Adoption Propensity
Design Improvements	More Options Generated by AI Handle Presentation and Formatting Summarize Other Information (e.g., Reasons, Summary, Errors) Customization Optimize UI Adaptive Prompts

APPENDIX B

Appendix for Chapter 9

B.1 Tutorial Lists in Formative Study

ID	Tutorial Title	Link	Author	Source
T1	Polymorphism	https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html	Official	Stack Overflow
T2	Guide to Java 8 Comparator.comparing()	https://www.baeldung.com/java-8-comparator-comparing	Third-Party	Stack Overflow
T3	React Authenticator	https://ui.docs.amplify.aws/react/connected-components/authenticator	Official	Stack Overflow
T4	React Router Tutorial	https://reactrouter.com/en/v6.3.0/getting-started/tutorial#tutorial	Official	Stack Overflow
T5	Explore SplashScreen API, Android 12, Kotlin	https://medium.com/realm/explore-splashscreen-api-android-12-kotlin-7a8bf83b061a	Personal	Stack Overflow
T6	Adding a splash screen to your mobile app	https://flutter.dev/go/android-splash-migration	Official	Stack Overflow
T7	Federated Learning for Image Classification	https://www.tensorflow.org/federated/tutorials/federated_learning_for_image_classification	Official	Stack Overflow
T8	Building Your Own Federated Learning Algorithm	https://www.tensorflow.org/federated/tutorials/building_your_own_federated_learning_algorithm	Official	Stack Overflow
T9	RTK Query Quick Start	https://redux-toolkit.js.org/tutorials/rtk-query	Official	Stack Overflow
T10	Redux Toolkit TypeScript Quick Start	https://redux-toolkit.js.org/tutorials/typescript	Official	Stack Overflow
T11	Text generation with an RNN	https://www.tensorflow.org/text/tutorials/text_generation	Official	Stack Overflow
T12	A Visual Guide to Using BERT for the First Time	https://jalamar.github.io/a-visual-guide-to-using-bert-for-the-first-time/	Personal	Stack Overflow

Table B.1: Tutorial Lists in Formative Study (1)

ID	Tutorial Title	Link	Author	Source
T13	Hugging Face Transformers: Fine-tuning DistilBERT for Binary Classification Tasks	https://towardsdatascience.com/hugging-face-transformers-fine-tuning-distilbert-for-binary-classification-tasks-490f1d192379	Personal	Stack Overflow
T14	Huggingface Fine Tuning	https://nbviewer.org/github/omontasama/nlp-huggingface/blob/main/fine_tuning/huggingface_fine_tuning.ipynb	Personal	Stack Overflow
T15	Swift 5.5: Asynchronous Looping With Async/Await	https://www.biteinteractive.com/swift-5-5-asynchronous-looping-with-async-await/	Third-Party	Stack Overflow
T16	How do Spring Boot 2.X add interceptors?	https://programmer.group/how-do-spring-boot-2.x-add-interceptors.html	Third-Party	Stack Overflow
T17	Testing Smart Contracts	https://hardhat.org/tutorial/testing-contracts.html#using-a-different-account	Official	Stack Overflow
T18	Tutorial: Create a Go module	https://go.dev/doc/tutorial/create-module	Official	Stack Overflow
T19	5 minute guide to deploying smart contracts with Truffle and Ropsten	https://medium.com/coinmonks/5-minute-guide-to-deploying-smart-contracts-with-truffle-and-ropsten-b3e30d5ee1e	Personal	Stack Overflow
T20	Using HDR rendering	https://github.com/microsoft/DirectXTK12/wiki/Using-HDR-rendering	Official	Stack Overflow
T21	JSF 2.3 tutorial with Eclipse, Maven, WildFly and H2	https://balusc.omnifaces.org/2020/04/jsf-23-tutorial-with-eclipse-maven.html#InstallingWildFly	Personal	Stack Overflow
T22	Developing an Accessibility Service for Android	https://codelabs.developers.google.com/codelabs/developing-android-a11y-service	Official	Stack Overflow
T23	Practical use of scoped slots with GoogleMaps	https://vuejs.org/v2/cookbook/practical-use-of-scoped-slots.html	Official	Stack Overflow
T24	Using Django Check Constraints to Ensure Only One Field Is Set	https://adamj.eu/tech/2020/03/25/django-check-constraints-one-field-set/	Personal	Stack Overflow
T25	JWT Auth in ASP.NET Core	https://codeburst.io/jwt-auth-in-asp-net-core-148fb72bed03?gi=cef51cc81e61	Personal	Stack Overflow
T26	How to add SectionIndexTitles in SwiftUI	https://www.fivestars.blog/code/section-title-index-swiftui.html	Third-Party	Stack Overflow
T27	Creating a React and Spring REST application that queries Amazon DynamoDB data	https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/javav2/usecases/creating_dynamodb_web_app	Official	Stack Overflow
T28	How to Train BPE, WordPiece, and Unigram Tokenizers from Scratch using Hugging Face	https://www.freecodecamp.org/news/train-algorithms-from-scratch-with-hugging-face/	Personal	FreeCodeCamp
T29	React CRUD App Tutorial – How to Build a Book Management App in React from Scratch	https://www.freecodecamp.org/news/react-crud-app-how-to-create-a-book-management-app-from-scratch/	Personal	FreeCodeCamp
T30	How to Build a Neural Network from Scratch with PyTorch	https://www.freecodecamp.org/news/how-to-build-a-neural-network-with-pytorch/	Personal	FreeCodeCamp

Table B.2: Tutorial Lists in Formative Study (2)

ID	Tutorial Title	Link	Author Source
T31	How to Build a Blockchain from Scratch with Go	https://www.freecodecamp.org/news/build-a-blockchain-in-golang-from-scratch/	Personal FreeCodeCamp
T32	PHP Laravel Tutorial – How to Build a Keyword Density Tool from Scratch	https://www.freecodecamp.org/news/how-to-build-a-keyword-density-tool-with-laravel/	Personal FreeCodeCamp
T33	How to Create a Production-Ready Webpack 4 Config From Scratch	https://www.freecodecamp.org/news/creating-a-production-ready-webpack-4-config-from-scratch/	Personal FreeCodeCamp
T34	How to build a PWA from scratch with HTML, CSS, and JavaScript	https://www.freecodecamp.org/news/build-a-pwa-from-scratch-with-html-css-and-javascript/	Personal FreeCodeCamp
T35	How to build an Angular 8 app from scratch in 11 easy steps	https://www.freecodecamp.org/news/angular-8-tutorial-in-easy-steps/	Personal FreeCodeCamp
T36	How to Build Your Coding Blog From Scratch Using Gatsby and MDX	https://www.freecodecamp.org/news/build-a-developer-blog-from-scratch-with-gatsby-and-mdx/	Personal FreeCodeCamp
T37	How to build a Neural Network from scratch	https://www.freecodecamp.org/news/building-a-neural-network-from-scratch/	Personal FreeCodeCamp
T38	Progressive Web Apps 102: Building a Progressive Web App from scratch	https://www.freecodecamp.org/news/progressive-web-apps-102-building-a-progressive-web-app-from-scratch-397b72168040/	Personal FreeCodeCamp
T39	How to build a range slider component in React from scratch using only div and span	https://www.freecodecamp.org/news/how-to-build-a-range-slider-component-in-react-from-scratch-using-only-div-and-span-d53e1a62c4a3/	Personal FreeCodeCamp
T40	How to build an HTML calculator app from scratch using JavaScript	https://www.freecodecamp.org/news/how-to-build-an-html-calculator-app-from-scratch-using-javascript-4454b8714b98/	Personal FreeCodeCamp
T41	You don't need chatbot creation tools — Let's build a Messenger bot from scratch	https://www.freecodecamp.org/news/you-dont-need-chatbot-creation-tools-let-s-build-a-messenger-bot-from-scratch-8fcbb40f073b/	Personal FreeCodeCamp
T42	HTML and CSS Project – How to Build A YouTube Clone Step by Step	https://www.freecodecamp.org/news/how-to-build-a-website-with-html-and-css-step-by-step/	Personal FreeCodeCamp
T43	The SaaS Handbook – How to Build Your First Software-as-a-Service Product Step-By-Step	https://www.freecodecamp.org/news/how-to-build-your-first-saas/	Personal FreeCodeCamp
T44	A step-by-step guide to making pure-CSS tooltips	https://www.freecodecamp.org/news/a-step-by-step-guide-to-making-pure-css-tooltips-3d5a3e237346/	Personal FreeCodeCamp

Table B.3: Tutorial Lists in Formative Study (3)

BIBLIOGRAPHY

- [1] Datasette: a shared data repository. <https://github.com/simonw/datasette>.
- [2] Sharedb, 2013. <https://github.com/share/sharedb>.
- [3] Project jupyter: Computational narratives as the engine of collaborative data science, 2015.
- [4] LinkedIn workforce report, 2018. <https://economicgraph.linkedin.com/resources/linkedin-workforce-report-august-2018>.
- [5] Google colab, 2019. <https://colab.research.google.com>.
- [6] Genepattern notebook, 2020. <https://notebook.genepattern.org/>.
- [7] H2o.ai, 2020. <https://www.h2o.ai/>.
- [8] Skulpt, 2020. <https://skulpt.org/>.
- [9] Cars, 2021. <http://lib.stat.cmu.edu/datasets/>.
- [10] D3.js, 2021. <https://d3js.org/>.
- [11] Jupyter, 2021. <https://jupyter.org>.
- [12] Kaggle starbucks satisfactory survey, 2021. <https://www.kaggle.com/mahirahmzh/starbucks-customer-retention-malaysia-survey?select=Starbucks+satisfactory+survey.csv>.
- [13] Tidyuesday - weekly challenge, 2021. <https://github.com/rfordatascience/tidyuesday/blob/master/data/2021/2021-05-18/readme.md>.
- [14] Vs code, 2021. <https://code.visualstudio.com/>.
- [15] Codetour, 2022. <https://marketplace.visualstudio.com/items?itemName=vscode-contrib.codetour>.
- [16] Deepnote, 2022. <https://deepnote.com/>.
- [17] JupyterLab, 2022. <https://jupyter.org/>.

- [18] Observable Notebook, 2022. <https://observablehq.com/>.
- [19] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216*, 2019.
- [20] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.
- [21] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.
- [22] Adrian Bachmann and Abraham Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 119–128. ACM, 2009.
- [23] Liang Bai and Yanli Hu. Problem-driven teaching activities for the capstone project course of data science. In *Proceedings of ACM Turing Celebration Conference-China*, pages 130–131, 2018.
- [24] Lorena A. Barba, Lecia J. Barker, Douglas S. Blank, Jed Brown, Downey. Allen B., Timothy George, Lindsey J. Heagy, Kyle T. Mandli, Jason K. Moore, David Lippert, Kyle E. Niemeyer, Ryan R. Watkins, Richard H. West, Elizabeth Wickes, Carol Willing, and Michael Zingale. Teaching and learning with jupyter, 2019.
- [25] Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. Docwizards: a system for authoring follow-me documentation wizards. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 191–200, 2005.
- [26] Brian Burg, Amy J. Ko, and Michael D. Ernst. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, page 259–268, New York, NY, USA, 2015. Association for Computing Machinery.
- [27] Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42, 2010.

- [28] Carrie J Cai, Emily Reif, Narayan Hegde, Jason Hipp, Been Kim, Daniel Smilkov, Martin Wattenberg, Fernanda Viegas, Greg S Corrado, Martin C Stumpe, et al. Human-centered tools for coping with imperfect algorithms during medical decision-making. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2019.
- [29] Julia Cambre, Scott Klemmer, and Chinmay Kulkarni. Juxtapeer: Comparative peer review yields higher quality feedback and promotes deeper reflection. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, New York, NY, USA, 2018. Association for Computing Machinery.
- [30] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. What’s wrong with computational notebooks? pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.
- [31] Charles H. Chen and Philip J. Guo. Improv: Teaching programming at scale via live coding. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale*, L@S '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [32] Yan Chen, Jaylin Herskovitz, Gabriel Matute, April Yi Wang, Sang Won Lee, Walter S Lasecki, and Steve Oney. Edcode: Towards personalized support at scale for remote assistance in cs education. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5. IEEE, 2020.
- [33] Yan Chen, Walter S Lasecki, and Tao Dong. Towards supporting programming education at scale via live streaming. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW3):1–19, 2021.
- [34] Yan Chen, Maulishree Pandey, Jean Y Song, Walter S Lasecki, and Steve Oney. Improving crowd-supported gui testing with structural guidance. In *Proceedings of the SIGCHI conference on human factors in computing systems*, CHI '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [35] Ruijia Cheng and Mark Zachry. Building community knowledge in online competitions: Motivation, practices and challenges. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW2):1–22, 2020.
- [36] Matthew Conlen and Jeffrey Heer. Idyll: A markup language for authoring and publishing interactive articles on the web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 977–989, New York, NY, USA, 2018. ACM.
- [37] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*, 3rd ed. Basics of qualitative research:

- Techniques and procedures for developing grounded theory, 3rd ed. Sage Publications, Inc, 2008.
- [38] Diana Cordova and Mark Lepper. Intrinsic motivation and the process of learning: Beneficial effects of contextualization, personalization, and choice. *Journal of Educational Psychology*, 88:715–730, 12 1996.
- [39] Catherine H Crouch and Eric Mazur. Peer instruction: Ten years of experience and results. *American journal of physics*, 69(9):970–977, 2001.
- [40] Barthélemy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 127–136, 2010.
- [41] Gabriele DAngelo, Angelo Di Iorio, and Stefano Zacchiroli. Spacetime characterization of real-time collaborative editing. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW):41:1–41:19, November 2018.
- [42] Datarobot, 2020. <https://www.datarobot.com/>.
- [43] Thomas H. Davenport and D. J. Patil. Data scientist: The sexiest job of the 21st century. 2012.
- [44] Alan Davies, Frances Hooley, Peter Causey-Freeman, Iliada Eleftheriou, and Georgina Moulton. Using interactive digital notebooks for bioscience and informatics education. *PLoS computational biology*, 16(11):e1008326, 2020.
- [45] Robert DeLine and Danyel Fisher. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 111–119. IEEE, 2015.
- [46] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James F Terwilliger, and John Wernsing. Tempe: Live scripting for live data. In *VL/HCC*, volume 15, pages 137–141, 2015.
- [47] Robert A DeLine. Glinda: Supporting data science with live programming, guis and a domain-specific language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2021.
- [48] Joan Morris DiMicco, Werner Geyer, David R Millen, Casey Dugan, and Beth Brownholtz. People sensemaking and relationship building on an enterprise social network site. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2009.
- [49] Distill, 2020. <https://distill.pub/>.

- [50] David Donoho. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, 2017.
- [51] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.
- [52] Jaimie Drozdal, Justin Weisz, Dakuo Wang, Gaurav Dass, Bingsheng Yao, Changruo Zhao, Michael Muller, Lin Ju, and Hui Su. Trust in automl: Exploring information needs for establishing trust in automated machine learning systems. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*, pages 297–307, 2020.
- [53] Carolyn D. Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspán, and James Lin. Predicting developers’ negative feelings about code review. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE ’20)*. IEEE, 2020.
- [54] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *29th International Conference on Software Engineering (ICSE’07)*, pages 688–697. IEEE, 2007.
- [55] Will Epperson, April Yi Wang, Robert DeLine, and Steven Drucker. Strategies for reuse and sharing among data scientists in software teams. In *44th International Conference on Software Engineering (ICSE)*, May 2022.
- [56] Travis Faas, Lynn Dombrowski, Alyson Young, and Andrew D Miller. Watch me code: Programming mentorship communities on twitch. tv. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–18, 2018.
- [57] Jesus Fernandez-Bes, Jerónimo Arenas-García, and Jesús Cid-Sueiro. Energy generation prediction: Lessons learned from the use of kaggle in machine learning course. *Group*, 7(8):9, 2016.
- [58] John C Flanagan. The critical incident technique. *Psychological bulletin*, 51(4):327, 1954.
- [59] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. Evaluating usage and quality of technical software documentation: an empirical study. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 24–35, 2013.
- [60] Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Bjorn Hartmann. Authoring multi-stage code examples with editable code histories. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 485–494, 2013.

- [61] Elena L. Glassman, Lyla Fischer, Jeremy Scott, and Robert C. Miller. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, page 609–617, New York, NY, USA, 2015. Association for Computing Machinery.
- [62] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):1–35, 2015.
- [63] Michael Gleicher, Danielle Albers, Rick Walker, I. Jusufi, C. Hansen, and Jonathan C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10:289–309, 2011.
- [64] Max Goldman et al. *Software development with real-time collaborative editing*. PhD thesis, Massachusetts Institute of Technology, 2012.
- [65] Max Goldman, Greg Little, and Robert C. Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 155–164, New York, NY, USA, 2011. Association for Computing Machinery.
- [66] Google colab, 2020. <https://colab.research.google.com>.
- [67] Julien Gori, Han L Han, and Michel Beaudouin-Lafon. Fileweaver: Flexible file management with automatic dependency tracking. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 22–34, 2020.
- [68] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH 2009 papers*, pages 1–9. 2009.
- [69] Scott Grissom and Mark J Van Gorp. A practical approach to integrating active and collaborative learning into the introductory computer science curriculum. In *Proceedings of the seventh annual CCSC Midwestern conference on Small colleges*, pages 95–100, 2000.
- [70] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814, 2014.
- [71] Philip J. Guo. *Software tools to facilitate research programming*. PhD thesis, Stanford University Stanford, CA, 2012.

- [72] Philip J. Guo. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, page 599–608, New York, NY, USA, 2015. Association for Computing Machinery.
- [73] Philip J. Guo and Margo Seltzer. BURRITO: Wrapping your lab notebook in computational infrastructure. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, TaPP'12, pages 7–7. USENIX Association, 2012.
- [74] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, page 89–98, New York, NY, USA, 2017. Association for Computing Machinery.
- [75] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.
- [76] Andrew Head, Jason Jiang, James Smith, Marti A Hearst, and Björn Hartmann. Composing flexibly-organized step-by-step tutorials from linked source code, snippets, and outputs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.
- [77] Catherine M. Hicks, Vineet Pandey, C. Ailie Fraser, and Scott Klemmer. Framing feedback: Choosing review environment features that support high quality peer assessment. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, page 458–469, New York, NY, USA, 2016. Association for Computing Machinery.
- [78] Fred Hohman, Kanit Wongsuphasawat, Mary Beth Kery, and Kayur Patel. Understanding and visualizing data iteration in machine learning. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–13, 2020.
- [79] Eric Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 159–166, 1999.
- [80] Youyang Hou and Dakuo Wang. Hacking with npos: collaborative analytics and broker roles in civic data hackathons. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):1–16, 2017.
- [81] ipyannotate: Jupyter widget for data annotation, 2020. <https://github.com/ipyannotate/ipyannotate>.

- [82] ipysheet: spreadsheet in the jupyter notebook, 2020. <https://github.com/QuantStack/ipysheet>.
- [83] David S. Janzen and Hossein Saiedian. Test-driven learning: Intrinsic integration of testing into the cs/se curriculum. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, page 254–258, New York, NY, USA, 2006. Association for Computing Machinery.
- [84] Jeremiah W Johnson. Benefits and pitfalls of jupyter notebooks in the classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education*, pages 32–37, 2020.
- [85] Malin Källén, Ulf Sigvardsson, and Tobias Wrigstad. Jupyter notebooks on github: Characteristics and code clones. *arXiv preprint arXiv:2007.10146*, 2020.
- [86] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. 18:2917–2926, 2012.
- [87] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3025453–3025626, 2017.
- [88] Mary Beth Kery, Bonnie E John, Patrick O’Flaherty, Amber Horvath, and Brad A Myers. Towards effective foraging by data scientists to find past analysis choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2019.
- [89] Mary Beth Kery and Brad A. Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29, 2017.
- [90] Mary Beth Kery and Brad A. Myers. Interactions for untangling messy history in a computational notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 147–155, Oct 2018.
- [91] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2018.
- [92] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid moves between code and graphical work in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 140–151, 2020.

- [93] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 96–107. ACM, 2016.
- [94] Younghoon Kim and Jeffrey Heer. Gemini: A grammar and recommender system for animated transitions in statistical graphics. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):485–494, 2020.
- [95] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks—a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.
- [96] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- [97] Kenneth R Koedinger, Elizabeth A McLaughlin, Julianna Zhuxin Jia, and Norman L Bier. Is the doer effect a causal relationship? how can we tell and why it’s important. In *Proceedings of the sixth international conference on learning analytics & knowledge*, pages 388–397, 2016.
- [98] Laura Koesten, Emilia Kacprzak, Jeni Tennison, and Elena Simperl. Collaborative practices with structured data: Do tools support what users need? In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2019.
- [99] Markus Konkol, Daniel Nüst, and Laura Goulier. Publishing computational research—a review of infrastructures for reproducible and transparent scholarly communication. *arXiv preprint arXiv:2001.00484*, 2020.
- [100] David Koop and Jay Patel. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.
- [101] Yasmine Kotturi, Chinmay E Kulkarni, Michael S Bernstein, and Scott Klemmer. Structure and messaging techniques for online peer learning systems that increase stickiness. In *Proceedings of the Second (2015) ACM Conference on Learning@Scale*, pages 31–38, 2015.
- [102] David R Krathwohl. A revision of bloom’s taxonomy: An overview. *Theory into practice*, 41(4):212–218, 2002.
- [103] Sean Kross and Philip J Guo. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–14, 2019.

- [104] Sean Kross and Philip J Guo. Orienting, framing, bridging, magic, and counseling: How data scientists navigate the outer loop of client collaborations in industry and academia. *arXiv preprint arXiv:2105.05849*, 2021.
- [105] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R. Klemmer. Peer and self assessment in massive online classes. *ACM Trans. Comput.-Hum. Interact.*, 20(6), December 2013.
- [106] Chinmay E. Kulkarni, Michael S. Bernstein, and Scott R. Klemmer. Peerstudio: Rapid peer feedback emphasizes revision and improves performance. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale, L@S '15*, page 75–84, New York, NY, USA, 2015. Association for Computing Machinery.
- [107] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2020.
- [108] Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. Improved code summarization via a graph neural network. *arXiv preprint arXiv:2004.02843*, 2020.
- [109] Xuye Liu, Dakuo Wang, April Yi Wang, Yufang Hou, and Lingfei Wu. HAConvGNN: Hierarchical attention based convolutional graph neural network for code documentation generation in Jupyter notebooks. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4473–4485, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.
- [110] Yang Liu, Tim Althoff, and Jeffrey Heer. Paths explored, paths omitted, paths obscured: Decision points & selective reporting in end-to-end data analysis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2020.
- [111] Patrick R Lowenthal. Social presence. In *Social computing: Concepts, methodologies, tools, and applications*, pages 129–136. IGI global, 2010.
- [112] Julia S Stewart Lowndes, Benjamin D Best, Courtney Scarborough, Jamie C Afflerbach, Melanie R Frazier, Casey C O’Hara, Ning Jiang, and Benjamin S Halpern. Our path to better science in less time using open data science tools. *Nature ecology & evolution*, 1(6):1–7, 2017.
- [113] Walid Maalej and Hans-Jorg Happel. From work to word: How do software developers describe their work? In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 121–130. IEEE, 2009.

- [114] Thomas W Malone. How human-computer'superminds' are redefining the future of work. *MIT Sloan Management Review*, 59(4):34–41, 2018.
- [115] John H Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 367–371, 2008.
- [116] Bernard Marr. The top 5 data science and analytics trends in 2023.
- [117] Joseph Bahman Moghadam, Rohan Roy Choudhury, HeZheng Yin, and Armando Fox. Autostyle: Toward coding style feedback at scale. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale, L@S '15*, page 261–266, New York, NY, USA, 2015. Association for Computing Machinery.
- [118] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q Vera Liao, Casey Dugan, and Thomas Erickson. 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*, pages 1–15, 2019.
- [119] Michael Muller, April Yi Wang, Steven I. Ross, Justin D. Weisz, Mayank Agarwal, Kartik Talamadupula, Stephanie Houde, Fernando Martinez, John Richards, Jaimie Drozdal, Xuye Liu, David Piorkowski, and Dakuo Wang. How data scientists improve generated code documentation in jupyter notebooks. 2021.
- [120] Brad A Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th annual ACM symposium on User interface software and technology*, pages 211–220, 1991.
- [121] Thi Thao Duyen T. Nguyen, Thomas Garncarz, Felicia Ng, Laura A. Dabbish, and Steven P. Dow. Fruitful feedback: Positive affective language and source anonymity improve critique reception and work outcomes. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW '17*, page 1024–1034, New York, NY, USA, 2017. Association for Computing Machinery.
- [122] Sylvie Noël and Jean-Marc Robert. Empirical study on collaborative writing: What do co-authors do, use, and like? *Computer Supported Cooperative Work (CSCW)*, 13(1):63–89, 2004.
- [123] Designing the nteract data explorer, 2018. <https://blog.nteract.io/designing-the-nteract-data-explorer-f4476d53f897>.
- [124] Cathy O'Neil and Rachel Schutt. *Doing Data Science: Straight Talk from the Frontline*. "O'Reilly Media, Inc.", 2013. Google-Books-ID: ycNKAQAAQBAJ.

- [125] Steve Oney, Christopher Brooks, and Paul Resnick. Creating guided code explanations with chat. codes. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–20, 2018.
- [126] Paircast: Code screencast software, 2020. <https://paircast.io/>.
- [127] Pandas - python data analysis library, 2021. <https://pandas.pydata.org>.
- [128] Pandas profiling, 2020. <https://pandas-profiling.github.io/pandas-profiling/docs/master/rtd/>.
- [129] Pandasgui, 2021. <https://github.com/adameroose/PandasGUI>.
- [130] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- [131] Raja Parasuraman, Thomas B Sheridan, and Christopher D Wickens. A model for types and levels of human interaction with automation. *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and Humans*, 30(3):286–297, 2000.
- [132] Soya Park, April Yi Wang, Ban Kawas, Q Vera Liao, David Piorkowski, and Marina Danilevsky. Facilitating knowledge sharing from domain experts to data scientists for building nlp models. In *26th International Conference on Intelligent User Interfaces*, pages 585–596, 2021.
- [133] Soya Park, Amy X. Zhang, and David R. Karger. Post-literate programming: Linking discussion and code in software development teams. In *The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings, UIST '18 Adjunct*, pages 51–53, New York, NY, USA, 2018. ACM.
- [134] Chris Parnin, Christoph Treude, and Margaret-Anne Storey. Blogging developer knowledge: Motivations, challenges, and future directions. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 211–214. IEEE, 2013.
- [135] Samir Passi and Steven J. Jackson. Trust in data science: Collaboration, translation, and accountability in corporate data science projects. 2:136:1–136:28, 2018.
- [136] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.
- [137] Jeffrey M Perkel. Why jupyter is data scientists’ computational notebook of choice. *Nature*, 563(7732):145–147, 2018.

- [138] David Piorkowski, Soya Park, April Yi, Wang, Dakuo Wang, Michael Muller, and Felix Portnoy. How ai developers overcome communication challenges in a multidisciplinary team: A case study. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW1):1–25, 2021.
- [139] Pluto.jl, 2020. <https://github.com/fonsp/Pluto.jl>.
- [140] Joe Gibbs Politz, Joseph M Collard, Arjun Guha, Kathi Fisler, and Shriram Krishnamurthi. The sweep: Essential examples for in-flow peer review. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 243–248, 2016.
- [141] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F Cohen. Pause-and-play: automatically linking screen-cast video tutorials with applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 135–144, 2011.
- [142] Ilona R Posner and Ronald M Baecker. How people write together (groupware). In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 4, pages 127–138. IEEE, 1992.
- [143] David Preston. Pair programming as a model of collaborative learning: a review of the research. *Journal of Computing Sciences in colleges*, 20(4):39–45, 2005.
- [144] Thomas W. Price, Joseph Jay Williams, Jaemarie Solyst, and Samiha Marwan. Engaging students with instructor solutions in online programming homework. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery.
- [145] Xiaoying Pu, Sean Kross, Jake M Hofman, and Daniel G Goldstein. Datamations: Animated explanations of data analysis pipelines. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2021.
- [146] Qri: community shared datasets, 2020. <https://qri.io/>.
- [147] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R Eagan, and Clemens N Klokmoose. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 715–725, 2017.
- [148] Bernadette M Randles, Irene V Pasquetto, Milena S Golshan, and Christine L Borgman. Using the jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–2. IEEE, 2017.

- [149] Reviewnb, 2020. <https://www.reviewnb.com/>.
- [150] El Kindi Rezig, Ashrita Brahmaraoutu, Nesime Tatbul, Mourad Ouzzani, Nan Tang, Timothy Mattson, Samuel Madden, and Michael Stonebraker. Debugging large-scale data science pipelines using dagger. *Proceedings of the VLDB Endowment*, 13(12):2993–2996, 2020.
- [151] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of nlp models with checklist. *arXiv preprint arXiv:2005.04118*, 2020.
- [152] José Miguel Rojas, Thomas D White, Benjamin S Clegg, and Gordon Fraser. Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 677–688. IEEE, 2017.
- [153] Marc J. Rubin. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13*, page 651–656, New York, NY, USA, 2013. Association for Computing Machinery.
- [154] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. Ten simple rules for writing and sharing computational analyses in jupyter notebooks, 2019.
- [155] Adam Rule, Ian Drosos, Aurélien Tabard, and James D Hollan. Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–12, 2018.
- [156] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proc. ACM Hum.-Comput. Interact.*, 2:150:1–150:12, 2018.
- [157] Adam Rule, Aurélien Tabard, and James D Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.
- [158] Adam Carl Rule. *Design and Use of Computational Notebooks*. PhD thesis, University of California San Diego, 2018.
- [159] Jeffrey Saltz, Kevin Crowston, et al. Comparing data science project management methodologies via a controlled experiment. 2017.
- [160] Jeffrey Saltz and Robert Heckman. Using structured pair activities in a distributed online breakout room. *Online Learning*, 24(1):227–244, 2020.

- [161] Sanddance, 2021. <https://microsoft.github.io/SandDance/>.
- [162] Isabella Seeber, Eva Bittner, Robert O Briggs, Triparna de Vreede, Gert-Jan De Vreede, Aaron Elkins, Ronald Maier, Alexander B Merz, Sarah Oeste-Reiß, Nils Randrup, et al. Machines as teammates: A research agenda on ai in team collaboration. *Information & management*, 57(2):103174, 2020.
- [163] Ben Shneiderman. Human-centered artificial intelligence: Reliable, safe & trustworthy. *International Journal of Human-Computer Interaction*, 36(6):495–504, 2020.
- [164] Nischal Shrestha, Titus Barik, and Chris Parnin. Remote, but connected: How#tidytuesday provides an online community of practice for data scientists. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW1):1–31, 2021.
- [165] Jeremy Singer. Notes on notebooks: is jupyter the bringer of jollity? In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 180–186, 2020.
- [166] Barbara Leigh Smith and Jean T MacGregor. What is collaborative learning, 1992.
- [167] Joanna Smith, Joe Tessler, Elliot Kramer, and Calvin Lin. Using peer review to teach software testing. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research, ICER '12*, page 93–98, New York, NY, USA, 2012. Association for Computing Machinery.
- [168] Anselm Strauss and Juliet Corbin. Grounded theory methodology: An overview. 1994.
- [169] Krishna Subramanian, Nur Hamdan, and Jan Borchers. Casual notebooks and rigid scripts: Understanding data science programming. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5. IEEE, 2020.
- [170] Sweetviz, 2021. <https://pypi.org/project/sweetviz/>.
- [171] Aurélien Tabard, Wendy E. Mackay, and Evelyn Eastmond. From individual to collaborative: The evolution of prism, a hybrid laboratory notebook. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work, CSCW '08*, pages 569–578. ACM, 2008.
- [172] James Tam and Saul Greenberg. A framework for asynchronous change awareness in collaborative documents and workspaces. *Int. J. Hum.-Comput. Stud.*, 64(7):583–598, July 2006.

- [173] Steven L Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, 2013.
- [174] Christoph Tauchert, Peter Buxmann, and Jannis Lambinus. Crowdsourcing data science: A qualitative analysis of organizations’ usage of kaggle competitions. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.
- [175] M. Rita Thissen, Jean M. Page, Madhavi C. Bharathi, and Toyia L. Austin. Communication tools for distributed software development teams. In *Proceedings of the 2007 ACM SIGMIS CPR Conference on Computer Personnel Research: The Global Information Technology Workforce, SIGMIS CPR ’07*, pages 28–35, New York, NY, USA, 2007. ACM.
- [176] Brad Victor. Explorable explanations. 2011.
- [177] April Yi Wang, Yan Chen, John Joon Young Chung, Christopher Brooks, and Steve Oney. Puzzleme: Leveraging peer assessment for in-class programming exercises. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW2):1–24, 2021.
- [178] April Yi Wang and Parmit K Chilana. Designing curated conversation-driven explanations for communicating complex technical concepts. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 211–215. IEEE, 2019.
- [179] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. Diff in the loop: Supporting data comparison in exploratory data analysis. In *CHI Conference on Human Factors in Computing Systems*, pages 1–10, 2022.
- [180] April Yi Wang, Andrew Head, Ashley Zhang, Steve Oney, and Christopher Brooks. Colaroid: A literate programming approach for authoring explorable multi-stage tutorials. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023.
- [181] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–30, 2019.
- [182] April Yi Wang, Ryan Mitts, Philip J Guo, and Parmit K Chilana. Mismatch of expectations: How modern learning resources fail conversational programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

- [183] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. Documentation matters: Human-centered ai system to assist data science code documentation in computational notebooks. *ACM Transactions on Computer-Human Interaction*, 29(2):1–33, 2022.
- [184] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. Callisto: Capturing the “why” by connecting conversations with computational narratives. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.
- [185] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. Don’t step on my toes: Resolving editing conflicts in real-time collaboration in computational notebooks. Under Submission, 2023.
- [186] Dakuo Wang. How people write together now: Exploring and supporting today’s computer-supported collaborative writing. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, pages 175–179. ACM, 2016.
- [187] Dakuo Wang, Q. Vera Liao, Yunfeng Zhang, Udayan Khurana, Horst Samulowitz, Soya Park, Michael Muller, and Lisa Amini. How much automation does a data scientist want? In *preprint*, 2021.
- [188] Dakuo Wang, Parikshit Ram, Daniel Karl I Weidele, Sijia Liu, Michael Muller, Justin D Weisz, Abel Valente, Arunima Chaudhary, Dustin Torres, Horst Samulowitz, et al. Autoai: Automating the end-to-end ai lifecycle with humans-in-the-loop. In *Proceedings of the 25th International Conference on Intelligent User Interfaces Companion*, pages 77–78, 2020.
- [189] Dakuo Wang, Haodan Tan, and Tun Lu. Why users do not want to write together when they are writing together: Users’ rationales for today’s collaborative writing practices. *Proc. ACM Hum.-Comput. Interact.*, 1(CSCW):107:1–107:18, December 2017.
- [190] Dakuo Wang, Justin D Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. Human-ai collaboration in data science: Exploring data scientists’ perceptions of automated ai. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–24, 2019.
- [191] Daniel Karl I Weidele, Justin D Weisz, Erick Oduor, Michael Muller, Josh Andres, Alexander Gray, and Dakuo Wang. Autoaiviz: opening the blackbox of automated artificial intelligence with conditional parallel coordinates. In *Proceedings of the 25th International Conference on Intelligent User Interfaces*, pages 308–312, 2020.

- [192] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. Fork it: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2021.
- [193] Justin D Weisz, Mohit Jain, Narendra Nath Joshi, James Johnson, and Ingrid Lange. Bigbluebot: teaching strategies for successful human-agent interactions. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 448–459, 2019.
- [194] Wikipedia contributors. Instant camera — Wikipedia, the free encyclopedia, 2022. [Online; accessed 15-September-2022] https://en.wikipedia.org/w/index.php?title=Instant_camera&oldid=1109305997.
- [195] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. B2: Bridging code and interactive visualization in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 152–165, 2020.
- [196] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3):205–253, 2019.
- [197] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*, 2018.
- [198] Ying Xu, Dakuo Wang, Penelope Collins, Hyelim Lee, and Mark Warschauer. Same benefits, different communication patterns: Comparing children’s reading with a conversational agent vs. a human partner. *Computers & Education*, 161:104059, 2021.
- [199] Matin Yarmand, Dongwook Yoon, Samuel Dodson, Ido Roll, and Sidney S. Fels. “can you believe [1:21]?!”: Content and time-based reference patterns in video comments. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI ’19, pages 489:1–489:12, New York, NY, USA, 2019. ACM.
- [200] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192, 2009.
- [201] Kimberly Michelle Ying and Kristy Elizabeth Boyer. Understanding students’ needs for better collaborative coding tools. In *Extended Abstracts of the 2020 CHI*

- Conference on Human Factors in Computing Systems Extended Abstracts*, CHI '20, page 1–8, New York, NY, USA, 2020. Association for Computing Machinery.
- [202] Alvin Yuan, Kurt Luther, Markus Krause, Sophie Isabel Vennix, Steven P Dow, and Bjorn Hartmann. Almost an expert: The effects of rubrics and expertise on perceived value of crowdsourced design critiques. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, page 1005–1017, New York, NY, USA, 2016. Association for Computing Machinery.
- [203] Amy X. Zhang and Justin Cranshaw. Making sense of group chat through collaborative tagging and summarization. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW):196:1–196:27, November 2018.
- [204] Amy X Zhang, Michael Muller, and Dakuo Wang. How do data science workers collaborate? roles, workflows, and tools. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW1):1–23, 2020.
- [205] Ge Zhang, Mike A Merrill, Yang Liu, Jeffrey Heer, and Tim Althoff. Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis. *arXiv preprint arXiv:2008.12828*, 2020.
- [206] Xiong Zhang and Philip J Guo. Ds.js: Turn any webpage into an example-centric live programming environment for learning data science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 691–702, 2017.
- [207] Chengbo Zheng, Dakuo Wang, April Yi Wang, and Xiaojuan Ma. Telling stories from computational notebooks: Ai-assisted presentation slides creation for presenting data science work. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2022.
- [208] Kevin Zielnicki. Explorations in reproducible analysis with nodebook, 2017.