



NORTHWESTERN UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2025-44

December, 2025

Process Management for Learning from Professional Source Code: Cultivating Diverse Experts in the Age of AI

Gobi Dasu

Abstract

Today's software industry increasingly values intricate conceptual knowledge (e.g., sufficient expertise to fix AI-generated code). Though program comprehension and scaffolding have mature literature, the intermediate-to-expert transition for programmers is underresearched relative to its socioeconomic impact. Learner-focused inspection tools exist (e.g., Ply, Isopleth, Python Tutor), but these tools often address specific problems (e.g., implicit dependencies in CSS, async JS, call stacks), without systematically guiding learners to develop deeper conceptual models across large codebases or interactions, where they must connect architectural decisions with a real-world domain. This lack of systematic guidance leaves learners overwhelmed by complexity, fixated on surface-level details, and unable to transfer concepts effectively across contexts. To address these challenges, we propose Process Management (PM), a framework for professional development tools that scaffold sensemaking on underused yet plentiful real-world code. PM tools systematically guide learners by embedding structured prompts, reflections on prior knowledge, interactive comparisons, and mapping activities that help them connect code components, understand architectural rationales, and transfer concepts across diverse examples. Prototyped and piloted over five years, our PM tools surface JavaScript, CSS, and Python techniques. Here, we report on a 1.5-year think-aloud study, including ~300 hours of remote observation, of diverse intermediate-level developers as they used these tools or controls

(Chrome DevTools, VS Code). Our mixed-methods evaluations show PM users explored effectively (e.g., comparing, mapping) and achieved $\sim 3X$ more learning outcomes (e.g., conceptual clarity, transferable patterns). Finally, we discuss PM's impact on cognitive load, the role of experts vs. AI, and plans for LLMs to scale process management.

Keywords

software engineering education, program comprehension, professional development, individual learning, self-directed learning, process management, design patterns, scaffolding, knowledge maps, code instrumentation, analogical transfer, authentic learning, developer tools, sensemaking

NORTHWESTERN UNIVERSITY

Process Management for Learning from Professional Source Code
Cultivating Diverse Experts in the Age of AI

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Computer Science

By

Gobi Dasu

EVANSTON, ILLINOIS

December 2025

© Copyright by Gobi Dasu 2025

All Rights Reserved

ABSTRACT

Today’s software industry increasingly values intricate conceptual knowledge (e.g., sufficient expertise to fix AI-generated code). Though program comprehension and scaffolding have mature literature, the intermediate-to-expert transition for programmers is underresearched relative to its socioeconomic impact. Learner-focused inspection tools exist (e.g., Ply, Isopleth, Python Tutor), but these tools often address specific problems (e.g., implicit dependencies in CSS, async JS, call stacks), without systematically guiding learners to develop deeper conceptual models across large codebases or interactions, where they must connect architectural decisions with a real-world domain. This lack of systematic guidance leaves learners overwhelmed by complexity, fixated on surface-level details, and unable to transfer concepts effectively across contexts. To address these challenges, we propose Process Management (PM), a framework for professional development tools that scaffold sensemaking on underused yet plentiful real-world code. PM tools systematically guide learners by embedding structured prompts, reflections on prior knowledge, interactive comparisons, and mapping activities that help them connect code components, understand architectural rationales, and transfer concepts across diverse examples. Prototyped and piloted over five years, our PM tools surface JavaScript, CSS, and Python techniques. Here, we report on a 1.5-year think-aloud study, including ~ 300 hours of remote observation, of diverse intermediate-level developers as they used these tools or controls (Chrome DevTools, VS Code). Our mixed-methods evaluations show PM users explored effectively (e.g., comparing, mapping) and achieved $\sim 3X$ more learning outcomes (e.g., conceptual clarity, transferable patterns). Finally, we discuss PM’s impact on cognitive load, the role of experts vs. AI, and plans for LLMs to scale process management.

ACKNOWLEDGEMENTS

I thank my advisors, professors Haoqi Zhang and Nell O’Rourke, for their patience, kindness, and consistent support. I am deeply grateful to them. Their expertise in needfinding, design research, the learning sciences, and human-computer interaction has profoundly shaped my research approach, sharpening my understanding of the intricate balance between practical problem-solving and conceptual contributions. I have realized that I must significantly advance my knowledge of literature and ideation abilities to be a true conceptual thinker. This realization has been profoundly humbling. I thank my advisors for helping me with this and for challenging me to slice, question my assumptions, refine my design arguments, perfect my methods, and rigorously evaluate my findings with an interdisciplinary perspective. They have taught me to appreciate research from a lens of both emotion and wisdom. Their guidance has inspired me to embrace resilience, adaptability, and a commitment to impactful, user-centered research that I hope to carry forward in my career.

I also would like to thank my family. My parents have taught me about the “wonder of the universe” around me, which many may dub God. They have unconditionally supported me throughout my life, providing me with incredible food, excellent education, solid values, fun times, careful guidance, and a life of opportunity. They emphasize the balance of work and joy and have given me a better understanding of the concepts of duty and gentle resilience. They are always there. Additionally, I want to thank my partner both in life and at work. She has moved across the world for me and has come to know me inside out over the last 4-5 years. She has been an indispensable pillar of support. I also want to thank my sister, grandparents, uncles, aunts, cousins, and well-wishers.

Finally, I would like to thank my committee (Darren Gergle, Christopher Riesbeck), my friends

in Delta Lab, the DTR community, the Segal Design Institute, and the Garage for their invaluable mentorship and insights throughout my PhD journey. I have learned so much from Garrett, Kapil, Ryan, Leesha, Harrison, Jamie, Evey, Gus, Eureka, Kristine, Spencer, Caryn, Melissa, Liz, Matt, Josh, Katie, Morgan, Nick, Yongsung, Maxine, Jonathan, Natalie, Yabi, Roxy, Fardeem, Daniel, Salome, Mieraf, Lev, Taylor, Nish, Anirudh, Hayes, N. Pinkard, U. Wilensky, B. Sherin, and many more at Northwestern, as well my close friends and teachers outside Northwestern like Abhi, Guillermo, Prag, Lucas, Sathish, Michael, Avi, Gerry, Ethan, Grace, B. Stoll, D. Hudkins, R. Meyer, Z. Andrews, J. Berry, M. Vickers, N. Manjoine, Jerry, C. Piech, M. Bernstein, and many more. I am grateful to the 100s of talent, customers, and investors in the LD community we've built across Africa, South Asia, and the U.S. When I look back on my PhD, I will remember all the great times we had together. Moreover, I hope that the self-serve professional development modules presented in this dissertation can bolster our goals of economic development through software engineering and AI-managed global teams.

TABLE OF CONTENTS

Acknowledgments	2
List of Figures	10
List of Tables	17
1 Introduction	19
1.1 A Design Argument for Process Management Tools	23
1.2 Process Management Framework for Professional Development through Real-World Code	26
1.3 PM tools for JS Value Transformations, Granular CSS Techniques, and Python Design Patterns	29
1.4 Research Questions around Exploration and Learning	32
1.5 PM Hypothesis	32
2 Related Work	34
2.1 Sensemaking Needs and Existing Tools	34
2.2 Best Practices Informing Tool Design	36
2.3 Recent Developments, Precautions, and Nuances	39

2.4	A Call for PM, Reflection, Comparing/Contrasting, and Analogical Transfer:	43
3	Needfinding through Pilot Cases of Novices Sensemaking Professional Code	44
3.1	Sensemaking Tools	45
3.2	Learning Outcomes and Websites	48
3.3	Qualitative Methods	51
3.4	Analysis and Findings for RQ1 on JS Level	52
3.5	Case Study of Exploration (RQ2) and Learning (RQ3)	54
3.6	XKCD Case Study Findings for RQ2, RQ3	55
3.7	Cyclemon Case Study Findings for RQ2, RQ3	57
3.8	Discussion and a Call for Process Management	59
4	Scaffolded Exercises (SE), a PM tool for JS Value Transformations	63
4.1	The Need for Process Management in Learning JS Value Transformations	63
4.2	How SE enacts PM: Ordered Exercises, Live Variable Values, Reflection, and Resurfaced Prior Notes	65
4.3	SE System Implementation: Reverse Call Stack Sequencing	71
4.4	Controlled Think-Aloud Evaluation of SE to Gauge Learning	75
4.5	SE Evaluation Results: Deconstructing the Effect of Scaffolded Process Manage- ment in SE	82
4.6	SE Discussion - Practical Learning and Technical Contributions	90
5	Knowledge Maps (KM), a PM tool for Granular CSS Techniques	93

5.1	How KM (Iteration 3) Enacts PM: Cycles of CSS Similarities and Differences . . .	97
5.1.0.1	How KM's Interface Implements KM's Process Management Steps	102
5.2	KM System Implementation	109
5.3	Controlled Think-Aloud Evaluation of KM to Gauge Learning	110
5.4	KM Evaluation Results: Process Management's Role in Attention to Detail	116
5.5	KM Discussion - Practical Learning and Technical Contributions	123
6	RALE Modules (RM), a PM tool for Python Design Patterns	126
6.1	How RM enacts PM: Intuitive Diagrams, Analogical Mapping, Comparing Trade-offs, Reflection	129
6.1.0.1	Illustration Example—Applying RALE Modules for Learning Mixins:	135
6.1.0.2	Analysis of RM (PM heuristics lens):	141
6.2	RM System Implementation	145
6.3	Controlled Think-Aloud Evaluation of RM to Gauge Learning	148
6.4	RM Evaluation Results: Process Management's Role in Facilitating Transfer from Theory to Practice	152
6.5	RM Discussion - Practical Learning and Technical Contributions	161
7	Discussion: Core Contributions and Principles for Designing PM Tools	164
7.1	Addressing Code Complexity through an Ordered Sensemaking Approach	164
7.2	Mitigating Surface-Level Exploration Biases through an Interactive Comparing and Linking Approach	167

7.3	Facilitating Transfer and Conceptual Awareness through a Mapping Approach . . .	168
7.4	Principles and Practical Guidance for Designing PM Tools to Facilitate Human-Driven Goals	169
7.5	Ongoing Challenges, Limitations, Methodologies, Ethics, and Future Opportunities	171
7.6	Technical Achievements and Gaps	176
7.6.1	Design Pattern Detection as an Open Research Problem	176
7.6.2	Automation Boundaries: What Can and Cannot Be Automated	177
7.6.3	Example Scope and Limitations	178
7.6.4	Technical Contributions and Limitations	179
7.6.5	Empowering Educators with New Capabilities	179
7.6.6	System Intelligence and Learner Modeling	180
7.6.7	Motivation and Commitment Considerations	180
7.6.8	Control Group Justification	181
7.6.9	Technical Learnings and Future Directions	181
7.6.10	Conceptual Contributions	182
7.7	Future Work and Generalization Opportunities	182
8	Conclusion: Democratizing Expert Practices through Process Management	189
8.1	Practical Problems and Technical Solutions	189
8.2	Conceptual Contributions	190
8.3	Evaluation Summary, PM Values, and Future Directions	192

References 207

Vita 208

LIST OF FIGURES

- 3.1 CDT provides users with (1) a DOM view and DOM inspection, (2) a file system, (3) a file view, (4) a breakpoint affordance, (5) variable values, and (6) live code stepping. 45
- 3.2 Isopleth's nodes represent function calls/arrows represent relationships. It has (1) a function call graph on a timeline, (2) multiple views including connected code components and code panes, (3) color coding of nodes by role, and (4) yellow/purple arrows for sync / async relationships. 47
- 3.3 The XKCD example presents a draggable map (1, 2, 3, 4) of a comic world. The system listens for the mousedown event, whose callback, when triggered, registers a listener for the mousemove event, which is in turn bound to a callback that updates image tiles, loading new ones as needed. On mouseup, the system deactivates the mousemove listener, unbinding it from its callback. 49
- 3.4 In Cyclemon, if you click the down arrow circled at the bottom of the page, it leads you to either the next bicycle/background scene (3, 4) or the bottom of the current one (1, 2), based on your current scroll position. The scroll callback manipulates the href attribute of the anchor $\langle a \rangle$ tag around the down arrow image. As you scroll to certain thresholds, the arrow points to either the current or next scene. . . . 50
- 3.5 In the pre-test, P26 listed events, listeners, and handlers without describing their coordination. But after Isopleth use in the post-test, they described the asynchronous architecture behind the drag functionality, identifying the mousedown callback triggers the binding of drag (1st block of POST) and the mouseup callback triggers its unbinding (2nd block of POST). P26 mentioned the loading and removing of different images as the user drags, depicting a load event and a remove function (third block). They said these three blocks repeat based on subsequent mousedown and mouseup events. 56

- 4.1 The XKCD Click and Drag example presents a draggable map of a comic world. In its implementation, mousemove events trigger the system to calculate a new page position for the user, which in turn is used to update the CSS positioning of a larger map div. The mouse position changes are also used to calculate a new value for a *centre* variable, which in turn is used to determine a *name* variable, which specifies which image tiles to load into the DOM. 66
- 4.2 Here we present a conceptual diagram of how SE works and enacts Quintana's heuristics, as well as several PM heuristics. The system presents exercises to the learner, starting with intuitive (PM1) DOM interactions (Step 1) and moving to variable value transformations deep within the call stack (Steps 2 and 3). The chunked exercises (PM3, PM4) provide learners with live updating variable values as they interact (PM5) with the webpage as an end user. The system prompts learners to articulate (PM8) their understanding of the semantics of variables and relevant code snippets while referencing their prior knowledge (PM2) in the form of their notes from previous exercises. This process repeats until they achieve coverage of all important variables and code (Step 4). 67
- 4.3 SE prototype on XKCD: It features (a) annotatable elements on the page, (b) resurfaced prior notes, (c) live variable values and DOM element tracking, (d) variable-specific notes, (e) reflection questions on DOM elements and variable values, (f) code views where live variables can be highlighted, (g) Tutorons, or hover-over definitions of key syntaxes, (h) reflection questions on code snippets, and (i) reflection questions on relationships among DOM elements and variable values. The code shown is dependent on which exercise within the overall ordering that the user is on. This figure's data was taken from P3's tool use on Ex. 2. 68
- 4.4 To isolate the effects of process management, we designed the control tool as a version of SE without key PM affordances but with state-of-the-art alternatives. This version had live variable values but none of SE's core features for making sensemaking intuitive, such as ordered exercises based on chunked code snippets, each with exercise-specific reflection questions and resurfaced prior notes. They also had access to CDT, the current state-of-the-art JS inspector, and a bulleted list of activities to consider, such as inspecting page visuals, noting variable values, linking code with visuals, and investigating relationships between variables. 77
- 4.5 Comparison of pre-test and post-test scores for SE experimental users. 84
- 4.6 Comparison of pre-test and post-test scores for SE control users. 84

4.7	This diagram shows P7's understanding of XKCD before SE use.	87
4.8	This diagram shows P7's understanding of XKCD after SE use.	87
4.9	This diagram shows P8's understanding of XKCD before control/CDT use.	88
4.10	This diagram shows P8's understanding of XKCD after control/CDT use.	88
4.11	This diagram shows P5's understanding of XKCD before SE use.	89
4.12	This diagram shows P5's understanding of XKCD after SE use.	89
5.1	KM Examples: Learners should also be able to recognize examples with (A) layouts where equally sized content overflows into new columns dynamically (e.g., Flat Icons, Hero Icons).	94
5.2	KM Examples: Learners should also be able to recognize examples with (B) layouts following a fixed table-like structure with sidebars and assorted rectangular sections (e.g., Italic, Masterclass).	95
5.3	KM Examples: Learners should be able to recognize examples with (C) responsive layouts without clear table-like structures (e.g., CSS Tricks, Smashing Magazine); (D) layouts with specific overflow behavior (e.g., CSS Tricks).	95
5.4	Here we present how KM helps users progressively build (PM2) a knowledge map of CSS concepts. Learners begin with the intuitive exercise of writing down a common overarching visual feature (PM1) in a subset of professional examples. They then categorize which examples have that common feature (step 1). They compare and contrast (PM6) examples to identify granular differences within each overarching similarity (step 2). Finally, learners dive deeper by looking at the code through the comparison tool, mapping (PM7) specific syntaxes with specific visual outcomes they have listed (steps 3-4). They answer reflection questions (PM8) [36] on how and why the syntaxes lead to specific visual outcomes. They then return to Step 1 and repeat these feature mapping cycles until they cover as much CSS layout code as possible (Step 5). Finally, they attempt to find instances of similar syntax leading to different visual outcomes and different syntax leading to similar visual outcomes (step 6). They also check to understand any lingering CSS code they did not cover.	98

5.5	KM walks users through (a) a sequence of steps (step 3 shown here as an example), each with (b) reflection inputs shown at the top of the webpage. Users can also select (c) a pair of websites to compare. Below the inputs, KM provides users the ability to (d) interactively resize examples' widths (to assess responsiveness) and (e) compare example pairs side-by-side. (f) Highlights also draw user attention to sites that share properties and property/value pairs. They can also (g) toggle (turn off and on) and (h) tinker with (change) the values of CSS properties. (i) Side-by-side HTML snippets and (j) hover-over definitions provide further context. These features help learners map granular visual outcomes to specific CSS techniques. . .	103
5.6	Step 1 - Identify Common Visual Layout Features (Sample Data)	105
5.7	Step 2 - Find Differences within Group (Sample Data)	106
5.8	Step 2 - Collect all Differences within Group (Sample Data)	107
5.9	Step 3 - Find Code behind Common Visual Layout Features, and the Rationale behind that Choice of Code (Sample Data)	107
5.10	Step 4 - Find Code behind Granular Differences, and the Rationale behind that Choice of Code (Sample Data)	108
5.11	Step 5 - Review Notes from Cycles of Finding an Overarching Visual Feature → Finding its Granular Sub-Features → Mapping its Code → Repeating the Process (P4 Data)	108
5.12	Step 6 - Find cases where different CSS techniques produce similar visual features and cases where similar CSS techniques produce vastly different visual outcomes. (Sample Data)	108
5.13	Three websites rich in granular CSS techniques used in the pre and post-tests. Handsome Frank (handsomefrank.com) is rich in grid techniques, media queries, transitions, and scrolling nuances. Alphabet Icons (by fontawesome.com) adds stretchable flex implementations to the equation. Rocksteady (rocksteadyltd.com) has tight spacing and custom grid arrangements. Together, these sites are sufficiently different from the examples used in tool use, so we could use them to test for transfer while still covering all the techniques taught during tool use.	112

5.14	The control participants have access to the KM system (Figure 5.6-5.12) without its primary process management affordances. The control does not have (a) a grouping activity, step 1 of the KM PM, (b) a comparing activity to find differences, KM's step 2, (c) structured prompting for code coverage with rationale, KM's step 3-4, (d) a comparison tool, (e) toggling/tinkering properties, or (f) code highlighting. However, the control users can access CDT, the same sites, and the same code snippets and definitions.	113
5.15	A sample of the code snippets available to all KM users.	113
5.16	A sample of the glossary terms available to both KM conditions (delivered via Tutorons for the experimental condition).	114
5.17	Comparison of pre-test and post-test scores for experimental (KM) users.	118
5.18	Comparison of pre-test and post-test scores for control users.	118
6.1	Process Management for Transfer in RALE Modules. Learners progress through three stages: (1) conceptual understanding by defining the core schema with its mechanism, goals, rationale, and tradeoffs, (2) mapping (PM7) the concept to a structured tutorial example, and (3) applying it in professional contexts where learners are prompted to reflect (PM8) on how [H] and why [R] it works, comparing and evaluating tradeoffs [T] (PM6). Diagrams (PM1), code snippets, and data flow (e.g., logs of specific data and program state, or function/method inputs/outputs) support each stage. By aligning the conceptual, tutorial, and professional cases, the model intends to foster deep understanding and robust knowledge transfer across the three contexts.	130

6.2	Knowledge Taught by RALE Modules. This figure demonstrates the application of RALE modules across three stages, using the pattern of mixins in Python/Django: (1) <i>Core Schema</i> : Introduces the concept of mixins (e.g., <i>Mixin</i> , adopter classes that inherit it, and associated adopted methods) to show how functionality can be inherited and combined across multiple classes. (2) <i>Tutorial Example</i> : Uses a request/response system with classes like <i>SpecialRequest</i> and <i>ExtraSpecialRequest</i> to explain how mixins (e.g., <i>AcceptMixin</i> , <i>ETagMixin</i>) provide reusable functionality, such as headers and tags. We highlight the tradeoffs of mixins versus traditional inheritance, showing how mixins allow flexible class combinations while avoiding rigid class hierarchies. (3) <i>Professional Example</i> : Applies mixins to a user analytics system, where different dashboards (e.g., <i>ActionViewSet</i> , <i>ClickHouseGroupTypes</i>) require specialized querying functionality. Learners using RM are guided to identify patterns, evaluate tradeoffs, and understand when to adopt each mixin based on their specific goals (e.g., filtering data or conditional tagging). Each stage includes scaffolds or prompts to explore this knowledge, aiming to foster deep understanding and transfer.	131
6.3	The design pattern definition and goals.	136
6.4	Diagram of the professional example's instantiation of the design pattern.	137
6.5	A sample of the professional example's code snippets.	137
6.6	A sample of the professional example's data flow.	138
6.7	A sample of the auto-generated activities around the professional example.	139
6.8	The tradeoffs exercise of the professional example.	140
6.9	Posthog is an open-source product analytics dashboard that helps software development teams analyze their users' behavior. It uses mixins to determine which logged-in team members (<i>c</i>) can access which users' data (<i>a</i>). It also uses mixins to determine team members' Posthog license status, affecting their ability to tag data (<i>b</i>) with labels. Finally, Posthog adopts an MVC structure in Django, with views or templates for different data, controllers that determine the data surfaced for each view, and models to keep track of different types of analytics data.	144
6.10	Here, we can see P2 mapping rationales from the theoretical and canonical contexts (see Figure 6.7) into the professional context.	145

6.11	Comparison of pre-test and post-test scores for experimental (RM) users.	153
6.12	Comparison of pre-test and post-test scores for control users.	153
6.13	Experimental P5's pre-test diagram of their understanding of Posthog's mixins. . .	158
6.14	Experimental P5's post-test diagram of their understanding of Posthog's mixins. . .	158
6.15	Experimental P2's pre-test diagram of their understanding of Posthog's MVT architecture.	158
6.16	Experimental P2's post-test diagram of their understanding of Posthog's MVT architecture.	158
6.17	Control P1's pre-test diagram of their understanding of Posthog's mixins.	159
6.18	Control P1's post-test diagram of their understanding of Posthog's mixins.	159
6.19	Control P1's pre-test diagram of their understanding of Posthog's MVT architecture.	159
6.20	Control P1's post-test diagram of their understanding of Posthog's MVT architecture.	159

LIST OF TABLES

4.1	SE pre/post-test learning outcomes grading rubric. Each item corresponds to one point. Partial credit details omitted for brevity.	80
4.2	Table showing SE users' pre/post-test scores and tool use metrics such as the exercises (E0-E8, E3 skipped, 8 total) and contributing subsections (v=variables, c=code, r=reflection) if we could pinpoint them, where users reached specific learning outcomes. In most cases, participants eventually reached the full learning outcome. In some cases, we included an exercise that got them to partial understanding or helped them build understanding before the '/'. For cases where participants did not reach the learning outcome or where the contributing exercise could not be clearly identified, we left the table cell blank.	83
5.1	Point Allocation by Category of the Learning Outcomes Rubric for Scoring Visual Layout Features and Underlying CSS Across Sites (much improved from previous KM iterations): Note we use ~ because some rubric items (codes) awarded a point for mentioning one of the several possible techniques (e.g., <i>flex</i> or <i>grid</i>) behind visual features (e.g., a table has a minimum of x columns), or some codes could go in multiple categories. Each rubric point was assigned to one of the possible categories it belonged to in the table above. Examples of techniques include <i>grid</i> , and examples of sub-techniques include <i>grid-template-columns: 1fr 1fr 1fr 1fr</i> , which produces a four-column layout, and <i>grid-template-columns: 1fr 1fr</i> , which produces two columns and is often used inside media queries for small widths. Our rubric contained many granular specifics to check if participants were noticing, conceptually understanding, and applying as many details as possible.	111

5.2	Table showing tool use statistics for KM users, including the number of granular visual layout differences found over ~ 3 cycles, as well as the number of user-discovered cases of similar visual features arising from different CSS techniques and vice versa. The table also shows learning outcomes KM users reached during tool use and their pre/post-test scores. The tool-use-specific learning outcomes included diverse aspects such as (a) flex nuances and transformations in Flat Icons and Smashing Magazine, (b) sophisticated transitions and scrolling behavior in CSS Tricks, and (c) intricate grid and spacing properties in Italic, Hero Icons, and Masterclass.	117
5.3	Table showing tool use statistics for control users, such as learning outcomes reached during tool use, along with pre/post-test scores (out of 80).	118
6.1	Table describing the rubric items across two design patterns (Mixins, MV*) that RM and control users were tested on. Our rubric items had specific details for each point and also awarded fractional points (0.25, 0.5, 0.75) for situations where learners partly understood aspects of our learning outcomes.	151
6.2	Table showing knowledge gains with pre/post-test scores (out of 31) for RM users.	152
6.3	Table showing knowledge gains with pre/post-test scores (out of 31) for control users.	152
7.1	Catalog of PM tool use intervention types and assessment of their automatability. Here, we define each intervention, including details around signals/heuristics that trigger it and the timing of when and if to intervene in each exercise/session.	186
7.2	Future Work Sketch for exploring automation of some PM interventions via a state-of-the-art LLM fine-tuned on our existing study transcripts and capable of retrieval (RAG) over a vector database of domain-specific content. We present 2 sample prompts out of several developed.	187

CHAPTER 1

INTRODUCTION

While the number and diversity of people with basic web development skills have grown [1], [2], novice and intermediate-level programmers still struggle to learn the more advanced concepts and skills required for professional practice. The rise of AI copilots has only heightened the need for conceptual understanding, as engineers must now prompt LLMs with high-level architectural inputs and verify and refine the generated outputs [3], [4]. While boot camps, videos, and online courses such as Codecademy and W3Schools help them learn basic syntax, they do not teach learners how to systematically explore and apply complex techniques, web architectures, and design patterns, which are the best practices professionals use to solve commonly occurring problems [5], [6], [7], [8]. Unable to apply these, non-experts may create ineffective, unmaintainable, and unscalable applications. For example, learners need to understand mixins, which allow classes to mix and match shared functionality, and detailed CSS techniques applied across diverse contexts. Books [5], [9], blogs [6], tutorials [10], [11], courses [12], and other manually created resources cover this knowledge gap, but they often focus on canonical examples in a single context, without discussing how to apply these techniques across complex real-world scenarios. As a result, aspiring developers may not understand the mechanisms and nuances of different approaches and cannot make tradeoffs, weighing one over the other. Moreover, they struggle to extract and transfer conceptual knowledge across multiple contexts. In this way, many novices¹ lack a process for obtaining a generalizable conceptual understanding of software architectures and fail to apply them

¹When we say “novices” in the context of the tools and studies of this work, we mean advanced novices or intermediate-level developers who have taken in-person or online courses and are ready to transition into experts. For a more precise description, please refer to the system evaluation sections of chapters 4, 5, and 6.

when building applications [13], [14].

To address this gap, researchers are investigating how novices can perform *sensemaking* and learn about design patterns and implementation techniques directly from the source code of authentic professional websites [13], [14]. **Sensemaking** is a learning sciences term referring to building understanding by actively exploring, seeking, and organizing knowledge into mental representations—in this context, making sense of a professional codebase [15], [16], [17]. Professional websites are a potentially powerful sensemaking context because they are inspectable with browser tools, reflect up-to-date techniques, and do not need to be maintained like tutorials [13]. In fact, experts do leverage these tools to inspect, gain inspiration from, and make sense of such complex professional code, as well as to make abstractions and draw connections between code components [18], [19], [20], [21]. They read code strategically [22], and actively inspect visuals and follow data flow [23], [24]. They organize their knowledge around deep rather than surface-level concepts [25] and develop accurate mental models. These strategies enable experts to trace data transformations, compare granular techniques, and understand domain-specific design pattern tradeoffs, which are outcomes indicative of conceptual expertise.

Even with tools such as Chrome DevTools (CDT) that help experts with sensemaking (e.g., mapping visuals to code), novices become overwhelmed by complex files, functions, variables, data flow, and relationships, not knowing where to focus first. They lack a straightforward process for inspecting, breaking down, and connecting key code-level details into high-level visual insights [14], [26], [27]. After observing novice developers use these web-inspection tools on professional codebases, we² hypothesized that each tool’s emphasis (e.g., debugging DOM or tracing events)

²Some writing in this chapter was written in collaboration with Nell O’Rourke and Haoqi Zhang, who have advised me throughout my PhD. Additionally, portions of Chapters 1, 2, and 3 are from unpublished work conducted in collaboration with my advisors. Content on Scaffolded Exercises (SE) includes writing done in collaboration with my mentees Maxine Whitely, Suzy Lee, Jonathan Liu, Natalie Brewster, and Yabi Ayele, members of the SE project. Content on Knowledge Maps (KM) includes writing done in collaboration with my mentees Roxy Wilcox, Fardeem Munir, Daniel Zhu, Salome Kariuki, Ava Robinson, Alexandra Andreiu, Mieraf Mulat, and Lev Rosenberg, members of the

inadvertently causes learners to **fixate** on certain architectures and ignore others. Such fixations may stem from novices’ process deficiencies and the absence of process-focused tools.

While modern learner-centric tools (e.g., CDT for debugging, Isopleth [14] for call graph visualization) provide excellent affordances for inspecting and understanding code (e.g., highlighting DOM elements, visualizing logic, or linking asynchronous relationships), they often focus on making code more accessible rather than guiding how to learn from it. These tools help novices see the code better, but they don’t teach them how to systematically investigate, compare, and transfer knowledge from examples (e.g., determining how to coordinate data flow to the DOM or choose granular CSS techniques to produce specific visual features). What’s missing is a structured process for learning—a way to guide novices through the complex task of making sense of professional code, similar to how experts naturally approach code exploration.

Our studies show that, in practice, novices and intermediate learners face multiple process obstacles. First, they struggle to locate meaningful starting points in complex codebases—for instance, when examining a React application, they might get lost in the component hierarchy without knowing whether to start with the UI components, state management, or data fetching logic. To overcome this, they need a process for systematically breaking down complex systems into manageable pieces, starting with high-level architectural views before diving into implementation details. Second, they focus too narrowly on surface-level details—when learning CSS Grid, they might notice the grid layout but miss how ‘grid-template-columns’ and ‘grid-auto-flow’ work together to create responsive designs. Here, they need a process for methodically comparing and contrasting different implementations to understand the nuanced relationships between code and visual outcomes. Third, they fail to transfer important architectural relationships into practice—after learning about the mixin pattern in a tutorial, they might not recognize how it’s

KM project. Some sections draw from our previous writings, whether unpublished or published through student research competitions (dl.acm.org/doi/abs/10.1145/3334480.3381444, dl.acm.org/doi/abs/10.1145/3491101.3516814).

implemented in a real-world analytics system. For this, they need a process for mapping abstract patterns to concrete implementations, including explicit steps for identifying pattern instances and understanding their contextual adaptations. Overall, challenges stem from novices’ limited mental models for executing effective sensemaking processes on foreign codebases, and from the inherent complexity of professional code. These process challenges manifest in different ways across domains and require careful consideration of how to structure learning experiences, as we detail in the following section.

To address these shortcomings and their manifestations across multiple domains (JS, CSS, and Python in this work), we propose **Process Management (PM)**, a framework for building professional development tools that scaffold learners to perform expert-level sensemaking on real-world code. PM is designed to help with process by providing structured guidance that mirrors how experts naturally explore and understand code. It scaffolds learners through intentional sequences of activities that build upon their existing knowledge, gradually revealing deeper architectural insights. PM tools guide learners to start with intuitive entry points (like visual outcomes or high-level patterns), then systematically explore implementation details through focused comparisons and mappings. By breaking down complex sensemaking tasks into manageable steps and providing explicit prompts for reflection and connection-making, PM aims to help learners develop the mental models and processes they need to understand professional code.

In this work, we embody the PM framework in three tools, each designed to teach critical yet challenging web development concepts: (1) JavaScript value transformations, (2) granular CSS layout techniques, and (3) Python design patterns, each of which stump process-weak novices. PM tools provide scaffolds [26], [28] that include intentional, step-by-step prompts and small, ordered activities to highlight key data flows, comparative tradeoffs [29], and design rationales in diverse professional codebases. Such process management and scaffolding have been implemented in or

recommended for domains such as writing and planning collaboration [30], [31] and searching for code [32]. Inspired, we investigate whether process management can help bridge the gap between theory and practice in software program sensemaking.

1.1 A Design Argument for Process Management Tools

To make sense of professional techniques, learners must first grasp their underlying structures, e.g., trace sequential JS variable transformations through the call stack to DOM updates, map granular visual differences to precise CSS code implementations, and relate abstract architectural schemas to real-world Python implementations. However, non-experts rarely recognize these hidden structures, let alone learn them strategically. Instead, without explicit scaffolding, learners become overwhelmed by code complexity, fixate on superficial features, struggle with concept transfer, and overlook architectural rationales. These core process obstacles are detailed below:

Code Complexity Challenges: Learners often struggle to navigate towards intuitive entry points for dissecting and decomposing large codebases. Their sensemaking processes are often overwhelmed by multiple components at once. This makes it difficult for them to recognize architectural elements and connect deeper logic. For instance, in JS, they may not trace value transformations of user inputs into visible DOM updates—not because they lack tools to inspect values (Chrome DevTools and Isopleth [14] provide this), but because they fixate on what these tools make most visible: the callback setup code and DOM updates. This fixation on the most immediately inspectable parts of the code (like event listeners, callback definitions, and DOM) leads them to start their investigation at a point that makes it difficult to understand the complete interaction flow with their default strategies. They get lost in the details of event binding before they can grasp the core data transformations that connect user input to DOM updates. Similarly, in CSS, they may ignore subtle differences in responsive layout because they start by inspecting individual style

properties rather than understanding the relationships between layout techniques. In Python, they may not see manifestations of design patterns (e.g., mixin pattern enactment in real-world code components) because they focus on class definitions in isolation rather than how classes interact to achieve specific design pattern instantiations and functionality. Without a structured and holistic approach [26], they cannot execute processes for incrementally building on their knowledge, delving into complexity, and developing mental models across varied domains.

Surface-level Focus: When sensemaking, learners may gravitate toward surface-level features (e.g., two websites use CSS grid, mixins and inheritance afford code reuse, intermediary variables that may seem unrelated to end-user visuals) and miss subtler differences in both design and underlying code (e.g. *grid: auto-fill* adds new table columns upon need while *grid-template-columns: 1fr 1fr 1fr* maintains three columns, mixins allow for selective inclusion of functionality, intermediary variable transformations cascade into DOM updates) [29]. This isn't because they can't see these differences—modern tools like Chrome DevTools make it easy to inspect and compare styles. Rather, they lack a process for systematically comparing implementations to understand why these differences matter. They might notice that two sites use CSS Grid, but without guidance on what to compare and how to interpret the differences, they miss the architectural implications of different grid configurations. Similarly, when examining code reuse patterns, they might see that both mixins and inheritance are used, but without a process for analyzing their tradeoffs in context, they don't understand when to choose one over the other. Uncovering these conceptually driven fine-grained decisions is challenging when a learner's process consists of cursory observation.

No Transfer Process: Transfer refers to applying knowledge in new contexts. Learners struggle to connect theory to practice because they lack explicit mental models of techniques, design patterns, and architectures. They fail to draw analogies between tutorial concepts/schemas [33] and real-world applications, treating them as separate rather than interconnected. They may use

tutorial examples to understand patterns in theory (e.g., media queries for webpage responsiveness, inheritance for code reuse). However, they do not map these abstract ideas to professional use cases (e.g., media queries can switch between *grid* and *flex* at low page widths, inheritance promotes code sharing among four FinTech developers with different tasks). This isn't because they can't read the code—tools like GitHub and code editors make it accessible. Rather, they lack a process for systematically identifying pattern instances and understanding their contextual adaptations. For example, when learning about the mixin pattern, they might understand the basic concept from tutorials but struggle to recognize it in a real analytics system where it's adapted to handle complex permission hierarchies. Without guidance on how to map abstract patterns to concrete implementations, they miss the nuanced ways patterns are adapted to solve real-world problems. Effective transfer requires processes for de-siloing and recognizing shared representations across examples—linking general problems, solutions, and rationales to their specific real-world counterparts [34].

Lack of Auxiliary Awareness: Beyond the core obstacles above, learners may neglect factors such as data flow, detailed visual outcomes, end users, and the application domain, which can influence architectural decisions (e.g., failing to trace how intermediary variables carry data to the DOM, missing nuances such as *overflow:hidden* vs *overflow:scroll*, failing to recognize the complex use cases that demand selective code reuse). Additionally, they might find it challenging to identify when and which alternative architectures should be considered before applying a given technique (e.g., justifying mixins over inheritance).

These “messy” obstacles reoccur across varied domains (e.g., JS, CSS, Python) and hinder non-experts constructing an understanding of essential but easily missed knowledge structures (e.g., value transformations, granular techniques, design patterns). To overcome these obstacles, PM tools fit into the structures to be learned in any given domain. They scaffold learners in under-

standing how and why techniques are applied in professional examples. Unlike inspection tools focused on revealing code structure, PM tools help learners systematically investigate and build conceptual knowledge.

1.2 Process Management Framework for Professional Development through Real-World Code

PM tools aim to help learners overcome these obstacles to perform sensemaking and achieve conceptual understanding of key techniques and their structures. While existing learner-centered tools (e.g., Isopleth, Ply) offer localized code visualization or debugging support [13], [14], our key contribution is *scaffolding the entire sensemaking process* across real-world JS, CSS, and Python. To shape the learners' processes, our framework recommends that professional development tools interleave the following heuristics, each denoted (PM#).

To overcome **code complexity challenges**, PM recommends a strategic sensemaking process where tools:

- **(PM1) Start with visual intuition** to provide learners with an effortless entry point into complex systems. Leverage UI updates (e.g., DOM, comparable websites) or diagrams (e.g., of design patterns) so they can begin sensemaking quickly about the function and supporting structure of code [19], [21], [26].
- **(PM2) Build upon prior knowledge in an organized manner** by anchoring exploration in familiar details or concepts learners already understand. By working within their zone of proximal development, we can help them progress through unfamiliar content and professional techniques [35]. In this way, learners can gradually grow and organize their knowledge [25].

- **(PM3) Surface salient parts of code or visuals** so learners focus on the most conceptually meaningful segments rather than getting lost in extensive files [14], [24]. This heuristic also helps build **awareness** of key components and considerations.
- **(PM4) Scaffold away extraneous cognitive load** by breaking down large amounts of information into manageable steps, ensuring learners can incrementally process examples and build understanding [25], [26], i.e., by tracing variable-by-variable, through comparing/contrasting/mapping steps, by moving from goals to mechanisms to rationales.

To overcome **surface-level focus**, PM recommends embedding interaction into the learning process. Tools should:

- **(PM5) Encourage interactivity** to allow learners to experiment, inspect, and manipulate different parameters, fostering hands-on exploration and deeper engagement [17], [26].
- **(PM6) Enable comparison** so learners spot crucial similarities and differences across examples, recognizing and analyzing meaningful variations in how techniques are applied [29], [34]. Relevant activities can include assessing alternative techniques and tradeoffs. This prevents learners from missing details or stopping too soon before reaching key insights across various domains (e.g., comparing JS variables to see their interdependencies, CSS layouts to see their nuances, Python design patterns to see their differentiated use cases).

To overcome **transfer challenges**, PM recommends that tools move beyond the inspection capabilities of current program comprehension tools by explicitly prompting users to connect high-level concepts (e.g., architectural patterns, intricate CSS techniques) back to concrete code implementations, generalizing across multiple files, projects, or contexts. Tools should:

- **(PM7) Map relationships important for conceptual understanding**, ensuring learners recognize connections between code components, between code and visuals, or between code and concepts [25], [26], [34]. In the most formal case, mapping entails making sophisticated analogies.
- **(PM8) Support reflection on how and why techniques apply in domain-specific contexts for particular end-users** to reinforce understanding and foster **awareness**. This enables learners to articulate the mechanisms and rationales behind professional architectural decisions [25], [26], [36].

From these principles, we distill three high-level, interconnected themes that directly shape the learner’s sensemaking process and map onto the core obstacles of code complexity, surface-level focus, and transfer:

1. **Following an Intuitive but Structured Sensemaking Order:** Learners are guided to begin with visually intuitive or high-level aspects of a system (as emphasized in PM1 and PM3), which helps them overcome the initial overwhelm of code complexity by providing accessible entry points. This structured order scaffolds them to progressively delve into deeper, more complex architectural and implementation details, rather than getting lost or fixated on arbitrary starting points.
2. **Building and Linking Prior Knowledge:** As learners move through exercises or examples, PM2 encourages them to anchor new information to what they already know, and to organize their exploration in a way that incrementally builds their mental models. This process directly addresses the risk of surface-level focus by ensuring that learners do not just notice superficial similarities, but also understand the underlying rationale and relationships

that distinguish different techniques or patterns, grounding code-level nuances with prior observations.

3. **Actively Mapping and Transferring Concepts to Specifics:** Through explicit mapping (and subsequent reflection) activities (PM7 and PM8), learners are prompted to connect abstract concepts, patterns, or schemas to concrete code implementations and real-world contexts. This theme is crucial for overcoming transfer challenges, as it helps learners recognize how theoretical knowledge is adapted and instantiated in professional code, and supports them in generalizing insights across domains and examples.

Together, these themes operationalize the process management principles, with the goal of ensuring that learners are not only exposed to professional code but are also equipped with the processes needed to navigate complexity, focus on what matters, and transfer their learning to new situations.

1.3 PM tools for JS Value Transformations, Granular CSS Techniques, and Python Design Patterns

In this work, we demonstrate process management by building and evaluating three PM tools:

1. **Scaffolded Exercises (SE)** provides aspiring Javascript programmers intuitive exercise orderings [26], e.g., starting with visual DOM updates but delving deep into the call stack. SE affords stepwise, interactive investigation of data flow in JS applications, embeds reflection, and aims to minimize extraneous cognitive load by working with the learners' resurfaced prior knowledge and within their zone of proximal development (ZPD) [35].
2. **Knowledge Maps (KM)** guides learners in iteratively grouping CSS-heavy examples by intuitive visual features, initially scaffolding away code details. The system helps learn-

ers compare examples to discover granular visual layout features. Once primed with these features, KM lets its users map them with underlying CSS. These activities and embedded reflection prompts intend to prepare KM users for concept transfer [29].

3. **RALE Modules (RM)**³ explicitly teaches architectural schemas and scaffolds structural analogies between textbook design patterns, tutorial examples, and professional Python implementations. RM allows users to compare the tradeoffs of alternative patterns and integrate them with prior knowledge of design pattern theory. By embedding intentional process scaffolds such as intuitive diagrams and scoped reflection, RM aims to help intermediate-level engineers understand how and why patterns apply in professional practice.

These tools embody PM approaches that aim not only to ensure learners inspect and understand professional code but also to equip them with deep conceptual knowledge that they can apply in real-world scenarios, e.g., JS, CSS, and Python use cases. For example, when learning JavaScript value transformations, learners without PM tools might get stuck inspecting event listeners and DOM updates in isolation, missing how data flows through the system. PM tools like SE guide them to start with the visual outcome (what changes on screen) and then systematically trace how those changes connect to the underlying data transformations. Similarly, when learning CSS layout techniques, learners might notice that two sites use CSS Grid but miss the nuanced differences in how they handle responsive layouts. PM tools like KM help them systematically compare these implementations, understanding not just that they use Grid, but why different Grid configurations are chosen for different use cases. And when learning design patterns, learners might understand the basic concept of mixins from tutorials but struggle to recognize how they're adapted in a real analytics system. PM tools like RM guide them to map the abstract pattern to its concrete imple-

³RALE stands for Readily Available Learning Experiences. This refers to the fact that all professional codebases can be made into readily available learning experiences.

mentation, helping them understand both how and why the pattern is used in that specific context.

Let's walk through a complete story for each challenge to understand why they occur and how our tools address them. When a learner tries to understand how, say, an online comic website updates its visualizations, they face the code complexity challenge: they can use Chrome DevTools to inspect components, but the tools only show them the structure, not how data flows from JS events through state updates to DOM changes. This happens because the tools make inspection easy but don't guide systematic exploration, leading learners to get lost in the component hierarchy. Our SE tool addresses this by providing a clear starting point—the visual outcome—and then guiding learners to systematically trace the data flow. Similarly, when studying two different e-commerce sites that use CSS Grid, learners face the surface-level focus challenge: they can inspect styles with Chrome DevTools, but without guidance on what to compare, they miss why one site uses *grid: auto-fill* while another uses fixed columns. This happens because the tools help them see the code but don't help them understand the architectural implications of nuanced visual and code differences. Our KM tool addresses this by providing a structured process for comparing implementations and understanding why different approaches are chosen. Finally, when a learner who knows about mixins looks at, say, a web platform's permissions system, they face the transfer challenge: they can read the code on GitHub, but they don't recognize how, say, the mixin pattern is adapted to handle complex permission hierarchies. This happens because while the tools make the code accessible, they don't guide learners in mapping abstract patterns to concrete implementations. Our RM tool addresses this by providing explicit guidance on how to map patterns to their real-world adaptations.

Incorporating these approaches, each PM tool instantiates the PM framework in a manner tied to the structure of knowledge being taught (e.g., JS value transformations, granular CSS mappings, Python design pattern use). By emphasizing novices' incremental construction of mental

models and their engagement in structured comparisons/mapping, we differentiate PM from existing platforms that primarily address local debugging or single-feature exploration. Our emphasis on guiding the *process* of sensemaking helps learners avoid stopping too soon and missing critical architectural or conceptual insights.

1.4 Research Questions around Exploration and Learning

Given our process management framework and its application across professional development tools, we investigate:

- **RQ1 on Exploratory Interaction and Sensemaking:** How do PM affordances scaffold learners' sensemaking and help them overcome specific challenges in professional code? Specifically, how do the tools shape their navigation style and focus points to help them (1) find intuitive entry points and avoid getting lost in complex code, (2) move beyond surface-level features to understand nuanced architectural patterns, and (3) transfer theoretical knowledge of pattern mechanisms and rationales to practical implementations?
- **RQ2 on Learning Outcomes:** How does PM tool use impact learners' architectural understanding and/or ability to transfer concepts into novel real-world contexts? Do PM tool users learn more than users of popular state-of-the-art tools that lack explicit process management? How do they express those newfound insights?

1.5 PM Hypothesis

We hypothesize that process management will improve learners' ability to explore professional code systematically and build on prior knowledge. This will lead them to deeper architectural understanding and potentially improved conceptual knowledge transfer. Specifically, learners using

PM tools will exhibit more structured exploration, heed hidden details, articulate nuanced insights more effectively, and demonstrate better real-world application of professional concepts compared to those without explicit process management support. PM tool users' superior concept retention and detailed expression will be evident from their written descriptions and diagrams produced after tool use.

CHAPTER 2

RELATED WORK

We now assess the novelty of PM against related work.

2.1 Sensemaking Needs and Existing Tools

Sensemaking Challenges: Sensemaking code, though essential, is challenging for novices. Program comprehension (PC) research explores how developers make sense of the structure and function of code in the context of their goals [21], [37], [38]. Studies show that novices often struggle with PC because it involves inferring complex relationships between components and connecting code details with higher levels of abstraction [18], [19], [20], [39]. Research suggests that learners have mental models that help them predict the behavior of computer programs, but their knowledge primitives and organization affect their models [40], [41], [42]. Moreover, these learners may have design inertia and must engage in “hidden labor” to reorganize knowledge and change their mental models [43].

Interactive Tools: Well-studied sensemaking and learning tools include code and data structure visualization and tracing tools such as Python Tutor [44], [45] and notional machines [46]. The Racket language likewise supports learning, enabling users to prototype new languages at various levels of abstraction [47]. Given complex code, WebCrystal and Whyline help novices explore and ask specific questions about program state or page visuals [48], [49], Tutorons let programmers quickly access inline definitions of complicated syntax within code [50], and code foraging tools, such as Blueprint and Meta, help professionals locate and reuse useful code snippets found on the web [51], [52], [53]. However, these tools are not designed to target the same learning

outcomes as PM; they address different problems, such as data structure understanding and state exploration, rather than the end-to-end conceptual learning from professional codebases that PM aims to achieve.

While these tools clarify isolated code segments, they do not necessarily prompt a global understanding of how changes propagate across a codebase. For instance, Python Tutor [44] users can visualize function calls, but they may not carry those insights forward to other architectural layers. They may understand a particular call stack or relationship, but not how broader architectures manifest visually or transfer into new scenarios. This shortfall can lead to knowledge fragmentation and missed conceptual links.

Beyond tool affordances, exploration styles impact sensemaking. For instance, studies have shown that Eclipse users debug code better when they engage in quick, prompted reflection and read code in vertical saccades following data flow, rather than the horizontal ones needed for reading English text [22].

Overall, these tools and studies support local lookups and bottom-up debugging but do not guide users through holistic conceptual exploration or help them build mental models across codebases. In practice, learners handle immediate bugs or questions but might not investigate broader designs or repeated code patterns unless scaffolding prompts them to do so. For example, a user might fix an off-by-one error in a single method (after referencing a foraged snippet) without recognizing parallel logic in another class. This gap indicates a need for structural guidance.

Professional Web Source Exploration Tools: Among professional source code exploration tools, CDT allows experienced developers to inspect and debug frontend code and files, mapping HTML, CSS, and JS to uncover why websites visually behave in particular ways [23], [24]. Fire-Crystal, Unravel, and Telescope help users specify UI interactions and locate the slices of code responsible for particular features or visual effects within large codebases [54], [55], [56]. Addi-

tionally, tools such as Scry [57] and Timelapse [58] let users explore code responsible for program state changes upon user interaction. Still, these tools’ inspection mechanisms omit relationships between code components. They represent closer approaches to PM in that they help with inspecting real, authentic code, but they are primarily targeted at experts rather than novices learning from professional codebases. Designed for learners rather than experts, Isopleth goes further by surfacing relationships through visual links representing asynchronous coordination and letting learners organize their insights [14]. Overall, research on feature location tools and even learner-centered developer tools successfully highlights developers’ cognitive processes, such as iterative exploration and distributed cognition. At the same time though, the research reinforces the need for additional cognitive scaffolding to structure the learning [59], [60], [61].

In a sense, none of these tools help learners know what they should be understanding, in what order, or how to make sense of larger architectures—they show specifics but don’t guide with higher-order strategies. For example, novices using these tools may inspect just the final rendered output or the topmost function call. Chapter 3 documents how such partial inspections fail to illuminate deeper insights that are not immediately obvious in the tool (e.g., missing event chains when the tool focuses on DOM manipulation). Repeatedly, we have found novices overlook the non-obvious when equipped with targeted tools that lack explicit scaffolding.

2.2 Best Practices Informing Tool Design

The Need for Scaffolds Towards Professional Practice: Beyond tool use, research advocates for cognitive apprenticeship and authentic exposure [62] to professional codebases, encouraging work on large pre-existing projects to improve confidence [63]. However, while cognitive apprenticeship is valuable, such coaching is not often available to learners, and authoring professional examples into learning resources for novices isn’t straightforward—professional code is optimized

for functionality rather than pedagogical clarity. Though today’s students use LLMs to explain and debug complex code, they still grapple with task decomposition and validating AI-generated work [4], pointing to the need for structure and scaffolding.

Recent approaches in scaffolding show promise. Work on UML-based peer feedback environments, iterative learning for architectural design pattern mastery, and professional code quality checkers point to a future of AI-driven scaffolding [64], [65], [66]. Others propose industry-aligned scaffolding frameworks to bridge the academia-to-industry skill gap among software engineering students [67].

Still, these works don’t sufficiently address the messy, open-ended exploration challenges learners face when exploring real-world code. While these works emphasize real-world engagement, they often assume learners already know how to structure their exploration process. Yet in reality, without explicit scaffolding, novices remain uncertain about what to examine or why, leading to incomplete or superficial knowledge (e.g., “this professional code uses the mixin pattern but I don’t know why and whether this is important”).

Pedagogical Best Practices for Patterns and Tradeoffs: Key skill gaps relate to understanding and transferring architectural knowledge, with research emphasizing the value of real-world scenarios, niche techniques, tradeoff documentation, and iterative hypothesis generation [65], [68], [69], [70]. In this context, open-source engagement and progressive project complexity have been identified as effective approaches for improving students’ ability to apply design patterns [63], [65], [68]. Researchers also stress organizing decision-making processes in structured tabular formats to reduce the cost of capturing and understanding tradeoffs [71]. Accordingly, our PM framework embodies many of these practices, including tradeoff comparison prompts and reflection steps throughout the exploration process.

From Education Theory to Semantic Frames to Scaffolds: Research suggests that experts

think beyond code syntax, as they also focus on behavioral and functional aspects [72] (insights we must keep in mind when building affordances for non-experts). To promote such expert practices, we should provide novices authentic experiences, tinkerable/modifiable examples, reflection opportunities, and engaging visuals that lower their cognitive load [73] and help them understand higher level functions in complex real world contexts. Cognitive load theory emphasizes reducing extraneous load, distinguishing between load components, and designing instructional strategies that help students manage working memory while engaging with structured learning experiences [74].

Building on such insights, for experts, semantic frame analysis has been refined to better parse and structure software engineering artifacts (e.g., API documentation, bug/security reports) to support program comprehension [75] and thereby professional practice. For novices, recent advances in AI-powered scaffolding systems and intelligent tutors (i.e., leveraging RAG/retrieval-augmented generation) show that automated yet personalized questioning and feedback can strengthen existing scaffolding benefits to better ease comprehension and computer science education [76], [77], [78], [79].

Nevertheless, these semantic frame and AI scaffolding approaches do not seem tailored towards intermediates learning broadly from real-world codebases. They may help a non-expert more easily approach code examples, or they may help an expert quickly navigate professional repositories. However, they may not assist an intermediate-level engineer with the tricky process of leveling up and understanding code one step beyond their current abilities yet still within their ZPD. Current AI approaches often provide isolated explanations, retrieved or sampled snippets/insights, and parsed code/docs with structured annotations, but without guiding learners through systematic architectural exploration or helping them walk through how design decisions cascade through multiple components. PM, in contrast, prompts learners to trace relationships between modules,

identify recurring patterns across different contexts, and understand the tradeoffs behind architectural choices. In contrast to existing approaches, PM is specifically designed for intermediate-level engineers and helps them regularly connect code syntax to its bigger functional role (e.g., how patterns such as mixins influence business logic), helping them see the otherwise opaque design motivations behind expert-level code decisions and surface-level visual effects.

Scaffolding Strategies in Computing Education: In computing education, research on fade-in/out scaffolding, Parsons problems, and structured debugging support suggests that stepwise guidance improves knowledge acquisition and problem-solving [80], [81], [82]. Studies on self-regulated learning recommend guided externalization and metacognitive scaffolding structures [83], [84], [85] and strategies. In AI/ML and data science for instance, sensemaking requires adaptability. Studies stress continuous, tactical, and improvisational comprehension strategies that are both structured and flexible [86].

Research also underscores the role of integrated development environments (IDEs) and interactive representations, such as code diagrams, annotations, documentation, and in-situ practice problems, in improving navigation and comprehension of unfamiliar codebases [87], [88], [89], [90]. Beyond PC and scaffolding, domain-specific debugging tools and autonomous code repair agents have been shown effective [91], [92]. Together, these trends point to the importance of strategic scaffolding within the work environment and an impending shift toward high-level conceptual thinking (PM’s focus), as autonomous agents increasingly handle low-level debugging and repair.

2.3 Recent Developments, Precautions, and Nuances

Recent Developments that Redouble the Call for PM and Scaffolding: Even thirty years ago, software-realized scaffolding tools provided structured processes for coaching, reflection, goal-

setting, and planning in programming and science education [93]. However, technological advancements and today's web of rich open-source professional code reframe the discussion for training both humans and machines. Recent studies indicate that structured tools enhance software testing proficiency despite low self-perceived improvement [94]. They also indicate that code complexity metrics often misalign with ease of understanding [95]—an area where scaffolding can potentially help. Embodied cognition approaches such as VR assist with cognitive load management in PC [96], and scaffolding in programming games boosts problem-solving [97]. In AI-assisted programming, concerns over workflow integration, code validation, and privacy necessitate structured approaches to ensure AI-generated code aligns with best practices [3].

From these works, we see that while advanced frameworks and AI-based support can streamline some tasks, novices still need a guiding backbone for developing the conceptual understanding required to check and refine AI-generated outputs. Otherwise, learners may solve specific programming problems, but not reflect on how they affect the business or best practices given the domain (e.g., fixing a class's method vs. reflecting on whether the method is appropriate for the business use case). To help with this, PM provides structure so a learner remains engaged with how aspects of the code relate across modules or domains.

Scaffolding Structure for Program Comprehension—Precautions and Nuances: Research on programming misconceptions [98], metacognition [83], and mental models [99] underscores the need for consistency and structure in instruction, in order to reduce errors and improve understanding. Eye-tracking studies further underscore this need, showing that programmers employ structured scanning patterns aligned with their conceptual schemas [100]. Moreover, adaptive learning systems in software engineering face challenges in engagement [101] and potentially need scaffolding.

In order to support program comprehension and overcome challenges and misconceptions,

research recommends trajectories of activities, each custom to specific levels of abstraction and learning goals around program structure, behavior, and function [102]. Some recommend multimodal engagement, such as interactive visualizations and sketching [103], while others caution against excessive context switching that hinders deep focus [104]. Taken together, these findings highlight that systematic, goal-aligned activities are essential if we want to help developers understand complex codebases and improve their professional practice.

Amidst “ways one can go wrong”, understanding novices’ weaknesses is a critical step before building tools that elicit reflection and knowledge construction from them. Studies on source code reuse caution that novices often repurpose code without full understanding, leading to potential misunderstandings and reduced software quality [105]. Similarly, research on online resource usage strategies by novice programmers highlights the risks of code copying and the need for critical evaluation training [106]. Broader reviews on software engineering education and professional skill gaps stress communication, self-reflection, software architecture/design, and configuration management [107], [108], which remain relevant in the intermediate stage and may require structured scaffolding to hone.

While literature is rife with best practices, tools that operationalize them end-to-end for intermediate-level learners studying real-world code seem scarcer. Our pilot findings (discussed in Chapter 3) suggest that standard debuggers or knowledge-specific tools (e.g., asynchrony-focused ones) alone do not provide learners with sufficient structured exploration to guarantee they cover and reflect on all pedagogical aspects of real-world code. PM is designed to mitigate this by prompting systematic reviews of each major component’s role in the overall design.

Authentic Learning in Computing Education: Authenticity is increasingly recognized as a key factor in computing education, influencing students’ motivation, engagement, and perceptions of programming [109], but the authentic experiences must be tractable for learners. Research

demonstrates that effective authentic learning experiences—those that connect to real-world contexts and professional practice—enhance student engagement and learning outcomes [110], [111]. For instance, while block-based environments can help novices, they can potentially impact later confidence in text-based programming [112]. Studies have shown that student perceptions of authenticity are nuanced and shaped by multiple aspects of the learning environment. DiSalvo [113] found that high school students viewed text-based programming as more authentic than block-based alternatives, especially when aspiring to traditional computing careers. Similarly, Weintrop and Wilensky [114] found that while students perceived block-based languages as easier, they also viewed them as less powerful and less authentic compared to textual programming. However, Lin et al. [115] demonstrated that student perceptions are more complex, with overall authenticity being influenced by factors beyond programming modality, including task authenticity and environmental design.

Theoretical frameworks for understanding authenticity in education emphasize its subjective and emergent nature. Betz et al. [116] argue that authenticity is constructed through meaningful participation rather than being pre-defined by instructors, with individual characteristics influencing sensitivity to perceiving authenticity. Schriebl et al. [117] provide a comprehensive model identifying key dimensions of authentic learning environments, including real-world connections, disciplinary practices, and personal contextualization. Finally, Guzdial and Tew [118] introduced approaches that design authentic experiences by having computing tasks create a perception of real industry practices (or ones potentially possible in the future), finding greater student engagement.

In our case, though the study of complex real-world professional source examples (particularly if they resonate with the learner) offers an authentic environment and hands-on task, authenticity without appropriate scaffolding can leave learners overwhelmed rather than empowered, as they may become blocked by extraneous code before reaching core problems. This nexus of needs un-

derscores the importance of tools and frameworks that can provide authentic learning experiences while still offering the structured guidance necessary for effective learning from professional codebases—a gap that our PM framework addresses.

2.4 A Call for PM, Reflection, Comparing/Contrasting, and Analogical Transfer:

The preceding sections reveal three critical insights: (a) existing sensemaking tools help with local debugging or identifying specific techniques but lack holistic architectural guidance; (b) pedagogical best practices emphasize authentic learning experiences and structured scaffolding, but there is room for development of tools for intermediate-level engineers; and (c) recent developments in AI-assisted programming and adaptive learning systems, while promising, still require explicit cognitive scaffolding to help learners develop the conceptual understanding needed to validate and refine AI-generated outputs. To address these gaps and help learners make sense of the web’s vast supply of authentic open source code [119], [120], our process management framework builds on research showing that conceptual transfer requires active articulation and contrasting diverse examples [29], [121], [122], while drawing from cognitive scaffolding principles [26] and analogical transfer theory [34]. Unlike existing tools that focus on specific techniques (e.g., Ply for helping learners identify visually impactful CSS [13]) or static pattern definitions [7], [123], PM provides structured guidance for holistic sensemaking across entire codebases, enabling learners to construct [124] the cross-cutting mental models needed for professional practice. This approach addresses the gap where novices using conventional tools may perform localized lookups but fail to construct a broader conceptual understanding of system architecture and design rationale.

Next, we review needfinding cases that motivate precautions against fixation during sensemaking.

CHAPTER 3

NEEDFINDING THROUGH PILOT CASES OF NOVICES SENSEMAKING PROFESSIONAL CODE

Since existing tools and their evaluations address specific problems that developers face, we know little about the existing processes of novices of varying JS experience levels when they perform holistic sensemaking, which would entail studying an entire codebase and its underlying architectures. So, we conducted a case study to examine how novices explore, reach 'aha' moments, and learn when using either CDT or Isopleth¹ to holistically make sense of authentic professional websites and their underlying architectures.

Our research questions follow. Answering them can provide insights into a new way of teaching web development—via holistic sensemaking of expert examples.

- (RQ1) Could novices of varying JS familiarity levels leverage tools such as Isopleth and CDT to learn from professional websites?
- (RQ2) How did tools such as Isopleth and CDT shape the sensemaking and exploration strategies of the novices?
- (RQ3) How did these tools shape their learnings about the architectures of the examples?

We found from the case study that JS familiarity level matters when leveraging tools to explore professional examples. We also found that different tools fit different types of web architectures (e.g., CDT helps with understanding DOM manipulations, Isopleth with asynchrony), pushing

¹<https://github.com/gobid/rale-rnd> fork

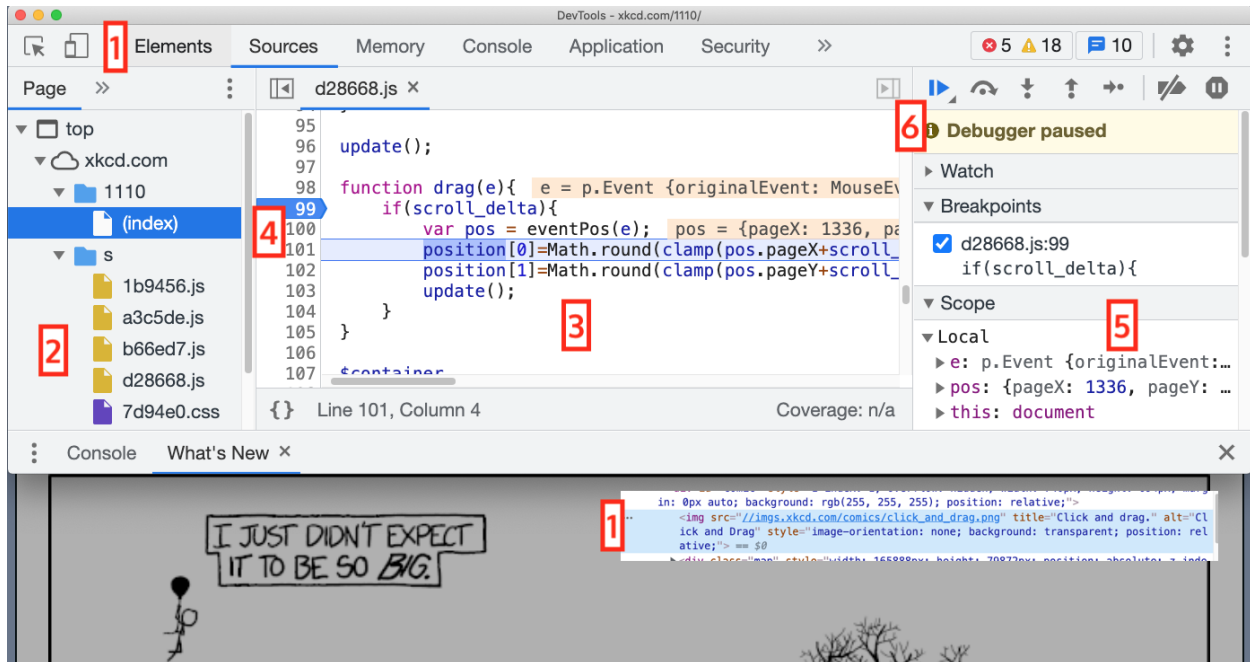


Figure 3.1: CDT provides users with (1) a DOM view and DOM inspection, (2) a file system, (3) a file view, (4) a breakpoint affordance, (5) variable values, and (6) live code stepping.

novices to fixate on specific architectures and ignore others within the same underlying professional websites. This finding makes sense, given that novices often do not know how to navigate a new professional codebase. The finding calls for a process management framework, which can incrementally scaffold learners with robust sensemaking strategies, using prompts and visual affordances to ensure they focus on all critical aspects of examples' architectures without missing some due to tool biases.

3.1 Sensemaking Tools

For our case study, we chose two substantially different tools, CDT and Isopleth, to gain broad-reaching insights into how tools available to novices shape their sensemaking of professional code. These insights should inform future systems that perform process management.

CDT, shown in Figure 3.1, is the industry standard browser-based inspection tool designed for web professionals to explore a website's file system, HTML, CSS, and JS. CDT provides an *Elements* view that lets users hover over and inspect parts of the webpage, highlighting relevant DOM elements in an expanded HTML tree, jumping to related HTML / JS code if the user wishes (Figure 3.1 (1)). Also, CDT's file system lets users browse tabs for HTML, CSS, JS, and media files on the frontend (Figure 3.1 (2)) and a file view which lets users read the source code of files and "pretty print" them (Figure 3.1 (3)). CDT is a "debugging" tool because it allows users to set breakpoints in JS files to pause code execution (Figure 3.1 (4)), step through code line-by-line to analyze control flow and data (Figure 3.1 (6)), and view local and global variable values in a live context (Figure 3.1 (5)).

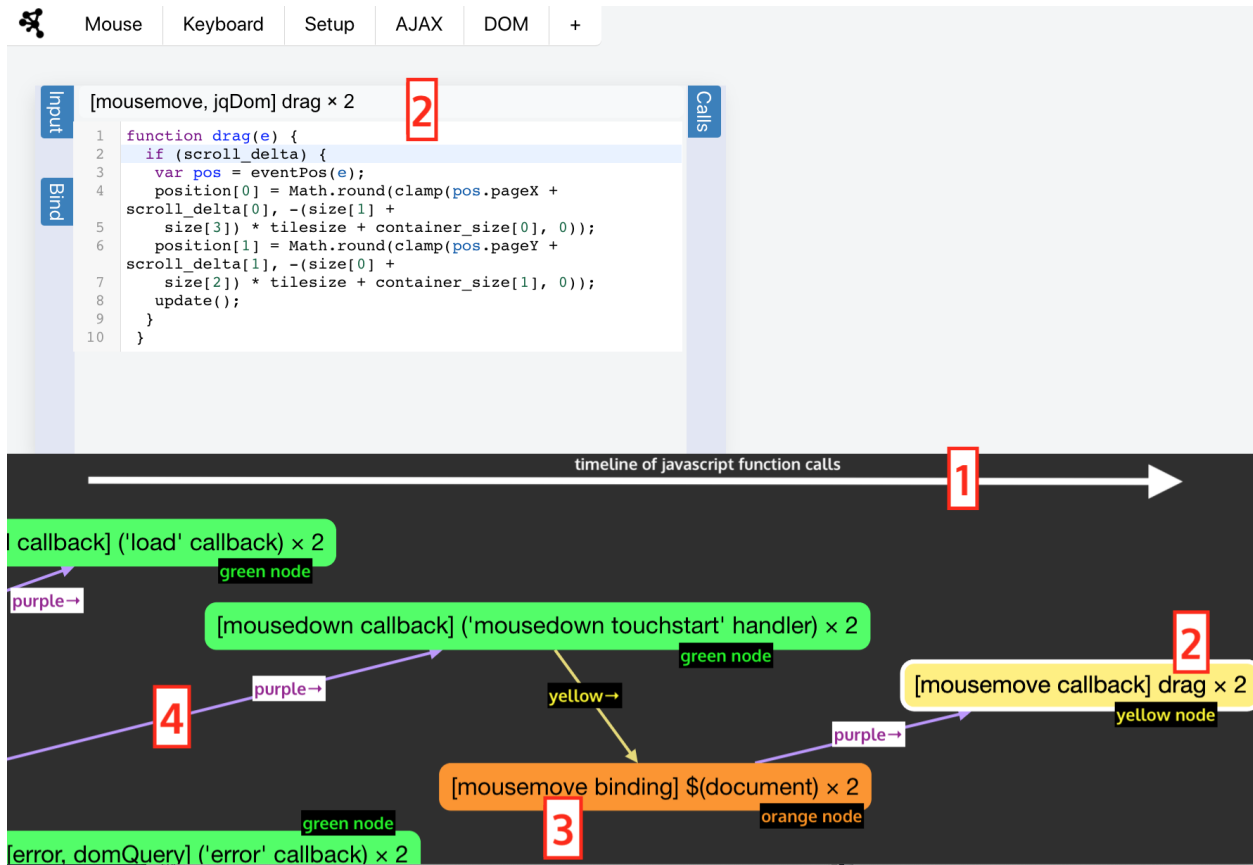


Figure 3.2: Isopleth’s nodes represent function calls/arrows represent relationships. It has (1) a function call graph on a timeline, (2) multiple views including connected code components and code panes, (3) color coding of nodes by role, and (4) yellow/purple arrows for sync / async relationships.

Isopleth was designed to scaffold novices to uncover callback bindings and asynchronous relationships when learning from professional websites [14]. In Figure 3.2 displaying Isopleth, the “mousemove binding” is an asynchronous function that listens for mousemove events. Once a user performs a “mousemove,” the callback function *drag* fires. Isopleth is a “graphical tool” as it offers a call graph view for JS’s non-obvious functional dependencies and execution timeline. It represents function calls as nodes and the relationships between them as arrows, with nodes positioned by their call time as in Figure 3.2 (1). Isopleth lets users identify synchronous and asynchronous

relationships between code components, depicted by yellow and purple arrows, respectively (Figure 3.2 (4)). Users have multiple views, such as node labels, which, when clicked, surface an alternate view, “code panes,” which present the node’s source code, as shown in Figure 3.2 (2). Finally, Isopleth presents the role of nodes through their colors, such as orange for Mouse-related, blue for Setup, white for AJAX (technology to fetch server data asynchronously), and purple for DOM as shown in Figure 3.2 (3).

3.2 Learning Outcomes and Websites

Among JS architectures, including overall architectures (that span the whole codebase, e.g., *MVC*), design patterns (more localized architectures, e.g., *constructors*, *client-server*), and recipes (step by step solutions), our study focuses on architectures teachable through and relevant to professional web applications [5], [6], [9]. These include the complex asynchronous coordination found in the draggable comic map XKCD ², and the JS manipulation of DOM found in the Cyclemon website ³. Our learning outcomes follow:

- **XKCD’s asynchronous architecture learning outcome** entails understanding its dragging, specifically stating how one asynchronous event callback binds another, as in Figure 3.3.
- **Cyclemon’s DOM manipulation learning outcome** entails understanding the intricate JS modification of DOM elements, specifically the scroll callback’s thresholded manipulation of where the down arrow leads, as in Figure 3.4.

²<https://xkcd.com/1110/>

³<http://cyclemon.com/>

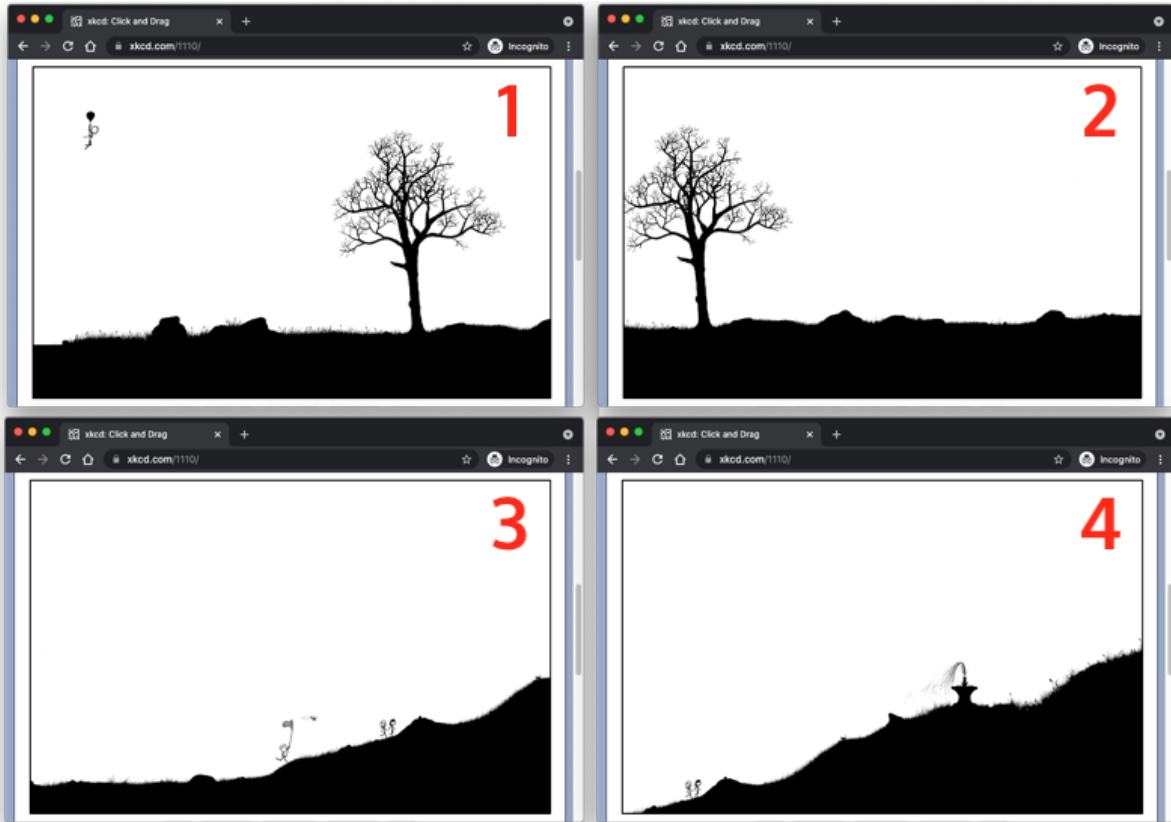


Figure 3.3: The XKCD example presents a draggable map (1, 2, 3, 4) of a comic world. The system listens for the mousedown event, whose callback, when triggered, registers a listener for the mousemove event, which is in turn bound to a callback that updates image tiles, loading new ones as needed. On mouseup, the system deactivates the mousemove listener, unbinding it from its callback.

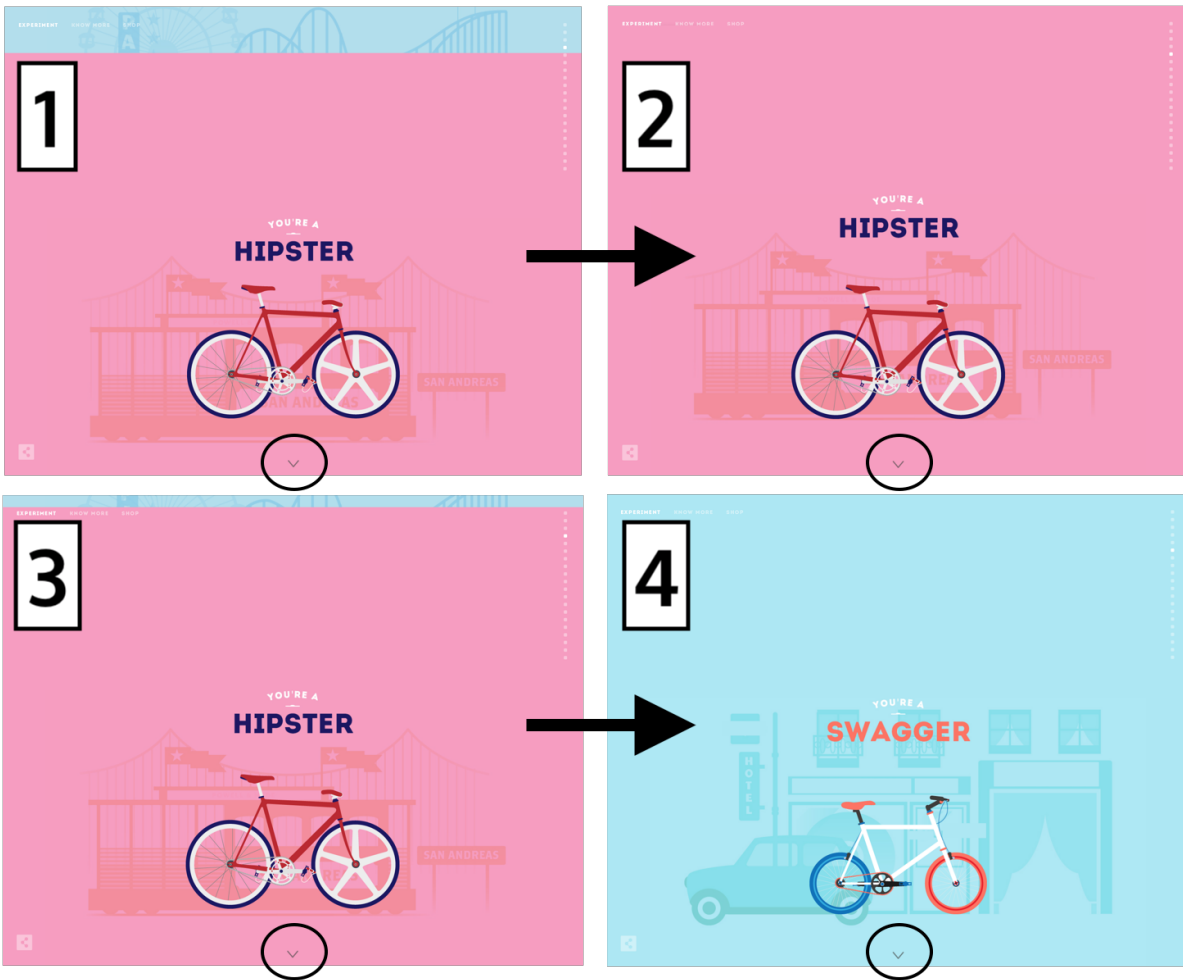


Figure 3.4: In Cyclemon, if you click the down arrow circled at the bottom of the page, it leads you to either the next bicycle/background scene (3, 4) or the bottom of the current one (1, 2), based on your current scroll position. The scroll callback manipulates the href attribute of the anchor $<a>$ tag around the down arrow image. As you scroll to certain thresholds, the arrow points to either the current or next scene.

3.3 Qualitative Methods

We conducted a video-recorded study where participants explored professional websites using Isopleth or CDT.

Participants: We recruited 21 undergraduate and graduate students from a mid-sized mid-western university (14 male, 7 female) to participate voluntarily. We needed students with enough JS knowledge to be able to learn but who were still novices without much experience applying JS concepts in practice. To gauge their JS familiarity level, we used a screening report. We asked the students to report on how well they understood a set of JS terms that are relevant to our learning outcomes on a scale of 0 to 3, with 0 meaning they hadn't heard of the concept, 1 meaning they had heard of it, 2 meaning they knew its definition, and 3 meaning they had applied it in practice. The terms were: *listener*, *event*, *AJAX request*, *asynchronous function*, *callback*, *event handler*, and *binding*. We chose a self-report given its precedence in literature [125]. Given our goal of identifying novices, we selected participants whose average score was strictly less than 3.0 but more than 1.25 on the screening report. This is so they would at least have heard of many of the concepts and know the meaning of some, but not have applied every one. All participants gave informed consent, they were compensated at a rate of 28 US dollars per hour, and each session lasted approximately 3 hours and 15 minutes. The dissertation author who conducted the study had no relationships or conflicts of interest with any participants.

Procedure: We conducted the study remotely with video/screen recording over Zoom from August to November 2020. We randomly assigned participants to use either CDT or Isopleth. To ensure similar access to knowledge across participants, we gave them a sheet describing the seven JS terms used during screening. First, novices learned how to use either Isopleth or CDT by watching a training video of tool use on a demo example, pausing the video to replicate the

actions on their own. Then, we asked them to explore both the XKCD and Cyclemon professional examples with the goal of understanding and describing their architectures, including details such as user events, data flow through and relationships between functions, and implementation of visual features. In this way, we encouraged them to holistically investigate properties related to both asynchronous event architectures and visual site DOM manipulation. We captured participant understanding at multiple points to detect tool impact and understand their sensemaking processes. First, as a pre-test, we asked participants to explore the site as an end user and then diagram how they think it is implemented while talking aloud. Next, they explored the example's source code using either Isopleth or CDT while talking aloud. Finally, for the post-test phase, we asked them to put away the tool and code, diagram how the example is implemented while talking aloud, and answer interview questions.

3.4 Analysis and Findings for RQ1 on JS Level

Analysis for RQ1 on JS Level: We watched and analyzed all the sessions (~ 70 h of video) to determine the JS level of novices who could leverage the tools, noting their sensemaking strategies and whether they could reach our learning outcomes. The dissertation author and their co-advisor independently analyzed 6 participants for XKCD (28% of the data) and 7 participants for Cyclemon (33%). They agreed on outcomes achieved by participants in all cases⁴. So, the first author analyzed the rest of the data.

While the tools did not advertise a target level, we found that the novices' JS level matters. The novices with screening reports in the range of 2.0 or less (12 users) formed a majority of our participants, and their post-test and interview data suggest the tools could not help them achieve

⁴More formal learning outcome IRRs are conducted in future chapters.

learning outcomes⁵. Novices with this lower level of conceptual JS knowledge and sensemaking ability could not easily leverage tool affordances to investigate functionality, make connections and abstractions, and draw conclusions. We will describe how and why for a few learners.

Isopleth fails P14, insufficient JS Level: For instance, P14 (screening report of 1.71) could follow individual asynchronous relationships in the XKCD example using Isopleth. Still, the tool could not help them piece together the architecture involving one event callback binding another. Perhaps P14 missed this learning outcome because they had serious misconceptions about asynchrony, thinking that a series of mouse events together could trigger a single callback for loading images when, in reality, events immediately trigger their respective callbacks. P14 could have also failed because they lacked the sensemaking strategies and processes that some of the more experienced novices developed, involving conceptually connecting within Isopleth the different coordinating asynchronous relationships, investigating how each relationship serves a key role in the overall architecture.

CDT fails P7 / P9, insufficient JS Level: Similarly, P7 and P9 (both with screening reports of 1.57) explored Cyclemon by inspecting its HTML and jumping to relevant JS code, which they read and in which they set breakpoints. However, CDT could not prevent them from missing Cyclemon's DOM Manipulation learning outcome, as they still viewed the code sections as siloed. CDT could not get them to investigate the down arrow button's role or conceptually connect the button's HTML with the JS code manipulating it. However, some novices reporting higher JS levels could do this in CDT.

Refocus on Novices with Sufficient JS Level: We only found tool-enabled learning outcome success cases among novices with JS level screening reports above 2.0, meaning they had heard of most terms and applied a few. There were nine such novices (4 Isopleth, 5 CDT). Two of the

⁵N.B. This is not a statistical, predictive claim about novices in general but rather a report to aid our case study.

four Isopleth users, but no CDT user, reached the asynchronous architecture outcome. Two of the five CDT users, but only one Isopleth user, achieved the DOM manipulation outcome. We selected participants for our case study from these nine, who seemed at a level where they could learn by leveraging the tools.

3.5 Case Study of Exploration (RQ2) and Learning (RQ3)

Case Studies for RQ2 on exploration, RQ3 on learning: Having identified a group of sufficient JS level for sensemaking of professional code, we developed cases to answer our questions on exploration and learning. Rather than set specific hypotheses, we chose an exploratory case study approach to show nuanced differences in ways novices explored, their “aha” moments or sparks of understanding, and processes and mechanisms behind their successes and failures in reaching architectural learning outcomes. Since this is a nascent research space, comparing a few detailed cases will be more beneficial than a less-nuanced summative analysis.

Case Selection: Inspired by Lewis’s case selection from her study on debugging behavior [126], we picked four cases to answer our research questions. To form our cases, we picked one success case from Isopleth and one from CDT, similar to the other success cases, except that the participant had more descriptive diagrams and talk-alouds. We also picked a contrasting case of a participant who used the opposite tool but had a similar screening report (within 0.5) and hence JS background. These four cases formed a two-by-two, half using Isopleth and half using CDT, half studying one example and half another. Using a method similar to Dziallas and Fincher’s in their programmer career cases [127], we compared our participants’ processes, sensemaking strategies, and learnings by understanding their prior knowledge in the pre-test, sticking close to their quotes, behaviors, and “aha” moments in tool use, and aligning their parallel stories up until the post-test and interview for detailed, nuanced analysis.

3.6 XKCD Case Study Findings for RQ2, RQ3

We compare how two novices explored and learned from XKCD.

P26 successfully explored asynchronous architectures with Isopleth: Our first case, P26, explored the XKCD example with Isopleth. In the pre-test, P26 mentioned events but not how they coordinate to implement the dragging, as shown in Figure 3.5 (PRE). In researching RQ2 around their exploration strategies, we found that P26 read node labels and followed arrows in the Isopleth call graph, helping them see the interplay between listeners, callbacks, and bindings, but missed other architectural aspects. Intentionally ignoring an image-updating function, P26 instead visited nodes labeled “mousemove binding” and “mousemove unbinding” and then followed purple arrows representing asynchronous relationships, in their words *“back a level to [their] parents”*. Moving back and forth between bindings and their parents, P26 said *“Oh ok mouseup callback, you’re unbinding whereas when it’s down you are binding it to the function that tells it to drag”*. In response to RQ3 on learning outcomes, we found that this exploration helped P26 understand the asynchronous event architecture and draw the post-test diagram shown in Figure 3.5 (POST). When asked about event responses, P26 said *“when you press down it binds the drag function, when you’re moving with your mouse down, that’s when the drag actually happens, when you have mouseup, it unbinds the drag function and you’re no longer moving”*. This case shows how Isopleth supported P26 in uncovering the architecture behind the dragging functionality. However, P26 did not explore other aspects, such as image updating.

P21 missed asynchronous architectures, discovered lazy loading with CDT: Our next case, P21, also explored XKCD but using CDT. In the pre-test, P21 incorrectly thought that the entire map was pre-loaded without mention of event binding coordination. In researching RQ2 on exploration strategies, we found that P21 mapped between page visuals, HTML, and JS but did not focus

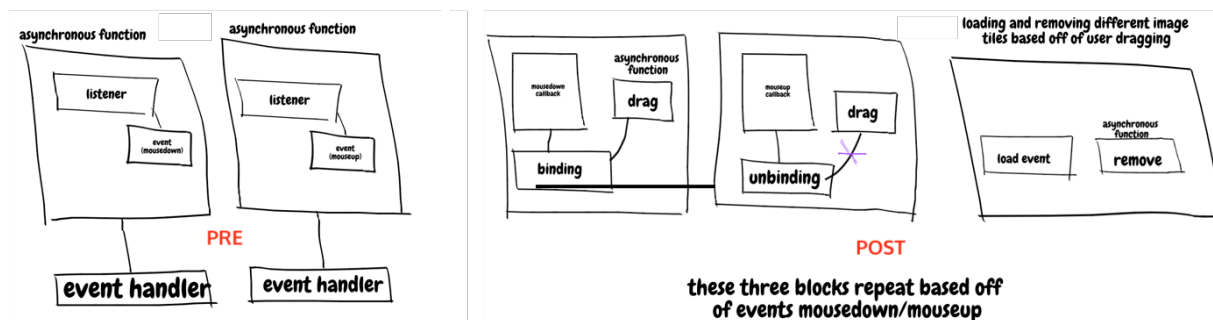


Figure 3.5: In the pre-test, P26 listed events, listeners, and handlers without describing their coordination. But after Isopleth use in the post-test, they described the asynchronous architecture behind the drag functionality, identifying the mousedown callback triggers the binding of drag (1st block of POST) and the mouseup callback triggers its unbinding (2nd block of POST). P26 mentioned the loading and removing of different images as the user drags, depicting a load event and a remove function (third block). They said these three blocks repeat based on subsequent mousedown and mouseup events.

on listeners, callbacks, and bindings like P26. Instead, after seeing the map’s images in CDT’s file system, they set breakpoints in image loading code, saying “*maybe that’s why it has all these ... ya it’s tiling together*”, expressing how multiple image tiles, rather than one large one, form the map. In the post-test, they said “*when you are dragging, drag was pinged ... from my breakpoint ... drag goes to update ... update did a whole lot of mapping and ... seemed to recalculate tile centers, chose new images*”. When interviewed about the interaction’s implementation, they described a threshold, saying the system would load new images “*based upon the distance that you’ve mousemoved*”, and if you have “*move[d] far enough*”. In this way, in response to RQ3 on learning outcomes, we found that P21 rejected their original pre-loading hypothesis, stating that dragging triggers the update function, which lazily loads and positions images upon need. Overall, CDT helped P21 clarify lazy loading, which is the on-demand fetching of images to reduce page load time. However, their sensemaking processes did not prevent them from missing the asynchronous event architecture, an outcome that P26 reached.

Tool-architecture fit and Tool-strategy fixation, XKCD: By comparing these cases, we noticed two phenomena. The first, *tool-architecture fit*, is how each tool’s features support some learning outcomes over others. Specifically, Isopleth’s binding labels and purple arrows helped P26 discover how the mousedown and mouseup events trigger callbacks that bind and unbind the mousemove event. In contrast, we saw how CDT’s DOM and file structure views helped P21 discover image loading but not event architecture. These findings show how different tools can lead novices to different learnings. Our analysis also revealed *tool-strategy fixation*, which is defined as how the tools shaped the novices’ exploration intent, processes, strategies, and experience in both helpful and unhelpful ways, making them fixate on some parts of the architecture while missing others. For instance, Isopleth’s call graph led P26 to follow arrows between nodes and focus on asynchrony rather than reading detailed code in the update node responsible for DOM updates like image loading.

In contrast, CDT’s easy file system access led P21 to find images, and the breakpoint affordance led them to step through code loading those images but not investigate the asynchronous event architecture. Each tool could have supported the learners, who started at similar points in the pre-test, in discovering what they missed. For instance, a CDT expert could have set breakpoints inside event callbacks to uncover asynchronous relationships, and an expert using Isopleth could have read the update function’s code pane. Instead, the tool affordances shaped the novices’ sensemaking processes and led them to fixate on aspects most readily available in the tool.

3.7 Cyclemon Case Study Findings for RQ2, RQ3

We compare how two novices explored Cyclemon.

P19 successfully explored JS Manipulation of DOM with CDT: Our first case, P19, explored the Cyclemon example with CDT. In the pre-test, P19 thought the system maintains a “*variable*

that keeps track of what section you're in" when, in reality, the system manipulates the down arrow to point to different page sections as you scroll. When investigating RQ2 on exploration, we found that P19 leveraged CDT's inspection affordance to investigate page visuals, linking them to HTML and DOM-manipulating JS. Searching for and inspecting the down arrow in CDT led P19 to its HTML, helping them realize that *"it's an image ... it's just a link to the next section"*. They then clarified that there is only one down arrow element, saying, *"I was thinking each section has its own button but I guess there's only one button that's fixed"*. Mapping the down arrow's HTML with related JS, they reached an "aha" moment, saying the system *"changes the href attribute [of the down arrow] ... to whatever the next section is"*, as you scroll. In response to RQ3 on learning, we found that tool use enabled P19 in the post-test to say, *"if you're scrolling through... it [the scroll callback] would check what section you're in and what div the top of your screen is in, and it would update ... the next one and put it in the anchor href attribute"*. These quotes and interactions show how the mapping between page visuals, HTML, and JS in CDT helped P19 discover the JS DOM manipulations behind the visual effect of the down arrow leading to different sections, depending on your position in the scrollable.

P22 missed JS Manipulation of DOM with Isopleth: Next, P22, explored Cyclemon with Isopleth. In the pre-test, P22 believed JS manipulates large DOM sections, such as scene images, using position information on scroll. In reality, the system elegantly updates only the down arrow pointer on scroll. When investigating RQ2 on exploration processes and strategies, we found P22 focused on asynchrony rather than DOM, following Isopleth's purple arrows that represent asynchronous event callback links, looking at node labels, and deep diving into code panes like that of the scroll callback that updates an anchor tag. However, when mousing near a function call labeled "append," they still incorrectly assumed that the system, *"appends the images as it goes instead of having them all loaded at once"*, perhaps due to less DOM access. In the post-

test, P22 could not equate the anchor tag, which is manipulated by JS, with the visual down arrow icon. When investigating RQ3 on learning outcomes, we found that P22 discussed where the down arrow button leads at different page positions but not how the scroll callback adjusts the down arrow upon hitting certain position thresholds. Perhaps this is because Isopleth showed the scroll callback updating elements but did not link that with actual page visuals like CDT did.

Tool-architecture fit and Tool-strategy fixation, Cyclemon: From the Cyclemon case study, we saw evidence of *tool-architecture fit* in that CDT’s affordances were a better fit than Isopleth’s for uncovering the unique JS DOM manipulations. CDT’s page inspection feature and tabs allowed P19 to identify the down arrow button within HTML and map it to related JS code. For P22, however, Isopleth’s features highlighted asynchronous relationships, which were less important. We also saw evidence of *tool-strategy fixation* when CDT’s inspection affordances and tabs encouraged P19 to adopt strategies such as inspecting page visuals and corresponding HTML and JS. In contrast, Isopleth’s call graph pushed P22 to fixate on asynchronous events. Since Isopleth did not map code with page visuals, uncovering the down arrow effect would have required the novice to have better processes and do more intentional sensemaking.

3.8 Discussion and a Call for Process Management

Implications of JS Level: In the study, we found that the JS level of novices impacted whether they could leverage tools like CDT or even the novice-targeted Isopleth to learn from professional examples. So, tool designers may want to be transparent about the JS level their tools target. Educators can develop criteria to determine whether a novice is ready to benefit from using sensemaking tools in professional examples or whether they should study the basics, such as language syntax and canonical examples. This way, curricula can introduce professional examples, sensemaking tools, and process management scaffolds at an optimal time in the novice developer’s training.

Implications of Tool-architecture Fit: Our findings of *tool-architecture fit* highlight potential downsides of designing tools to address specific sensemaking challenges. As we saw and as the literature suggests, novices may understand specific aspects but still struggle to abstract to a higher level, making sense of whole codebases and their architectures [19], [20]. More fundamentally, our findings reveal that novices do not possess effective sensemaking strategies independent of what tools are providing, leading them to focus on aspects that are most readily accessible through the tool’s interface rather than strategically approaching the code to understand embedded concepts. This dependency results in fragmented understanding where novices may grasp individual components but fail to see how they work together to create the overall system behavior. To support holistic learning, tool designers should be transparent about the architectures their tools highlight. They can even combine affordances to highlight many architectures with a single tool. We could also teach novices how to pick appropriate tools to investigate all the critical aspects of a codebase. And, while current tools do not teach sensemaking, future tools can perform process management by scaffolding novices of varying JS levels to develop strategies for making connections and abstractions, setting holistic learning goals, connecting theory with practice, making tradeoffs, and sensemaking entire web architectures effectively.

Implications of Tool-strategy Fixation: Our findings of *tool-strategy fixation* suggest that tools alone may bias novices towards certain aspects, influencing rather than helping them develop their own sensemaking processes and strategies. This finding aligns with a gap in the literature, where program comprehension emphasizes holistic behaviors such as drawing connections and abstraction, but actual JS sensemaking tools promote specific behaviors and foci (e.g., feature location, page inspection, etc.) [18], [19], [20], [21]. Moreover, when learning from professional code, novices may have their own intentions, interests, and prior knowledge. So, we may want to perform process management and teach them to intentionally select sensemaking strategies for

investigating an entire web codebase, given their hypotheses and architectural learning goals. In this way, later, they will be well equipped to develop their own processes. Such strategizing may prevent them from fixating on some architectures yet missing others that are still possible to uncover with given tools but more deliberate sensemaking. To help, tool designers can be transparent about the sensemaking styles their tools promote, and they can even design process management tools that scaffold novices with relevant activities that link code components, theory, and practical examples. Finally, educators can teach meta strategies to mix and match tools that promote different types of holistic sensemaking to fit each new situation.

Threats to Validity: Our case study intentionally focused on nuanced stories of novice sensemaking of professional examples rather than generalizable claims. However, we acknowledge that the cases only covered two code examples, tools, and types of architectures, and limited pairs of users per tool when many of each exist in the real world. Despite its nuance, its generalizability is limited, as it only tells these individuals' stories. We cannot guarantee all novices will explore or learn in these ways. We would need a more extensive study to make predictive, statistical claims about differences in conditions or tools. Moreover, since most novices lacked the strategies and JS level for learning outcome success, in hindsight, though difficult without tools advertising a target level, studying more advanced novices or scaffolding holistic sensemaking could have yielded more data and insights. Our learning outcomes could have been more comprehensive, including patterns such as lazy loading discovered by P21. Also, our interview questions around implementation aspects, while helpful for eliciting detail, could have influenced learner thoughts. Finally, study time constraints may have influenced the novices' exploration strategies.

Future Work: In summary, this work provides examples of how CS novices explored and learned from professional examples with sensemaking tools, drawing implications of *tool-architecture fit* and *tool-strategy fixation* for CS Education researchers and tool designers. Future studies can

cover more examples, tools, and users, evaluating knowledge transfer with a programming design task. In-the-wild Chrome extensions tracking users as they study examples could trace a more self-paced and longitudinal novice-to-expert journey. Designers and educators could also develop process management tool scaffolds and curricula to teach novices of various JS levels the sense-making strategies of setting goals, drawing connections, and making abstractions. Chapters 4, 5, and 6 of this dissertation each describe one such process management tool for sensemaking.

Linking Observed Challenges to PM Themes: Our pilot cases revealed that novices often do not know how to *navigate code complexity* (e.g., given CDT, they dove into files that appear first while ignoring asynchrony), exhibit *surface-level focus* via local debugging or single-feature attention (e.g., image loading), fail to *transfer* conceptual insights from one part of the codebase to another (e.g., given Isopleth, they noticed a scroll callback but did not see how it applies to elegant DOM [href] updating), and lack broader *awareness* of architectural or design relationships (e.g., given CDT, ignoring asynchronous callbacks). These tool-architecture fit and tool-strategy fixation manifestations match the four core sensemaking struggles outlined in Chapter 1. For example, participants working with CDT would fixate on the *visible DOM manipulations* and overlook deeper, asynchronous event chains. This illustrates both a surface-level approach and limited awareness of asynchronous interconnections across code components. Meanwhile, Isopleth users recognized some hidden callbacks but generally lacked an overall strategy for relating them to intricate visual updates. Such omissions underscore the importance of *structured exploration prompts* and *reflection steps* advocated by the PM framework, which helps novices incrementally conquer complexity, avoid tunnel vision, connect lessons across modules, and maintain a broader perspective on the entire architecture.

CHAPTER 4

SCAFFOLDED EXERCISES (SE), A PM TOOL FOR JS VALUE TRANSFORMATIONS

The concepts of fit and fixation from our needfinding case study in chapter 3 contributed to our team’s running hypothesis that learners struggle to make sense of professional codebases holistically when they focus on isolated execution details without structured exploration. In this chapter, we study how to create process management scaffolds for novices to overcome such challenges in the context of learning JS variable transformations.

4.1 The Need for Process Management in Learning JS Value Transformations

¹It is challenging for JS novices to understand interconnections and data flow, such as JS variable transformations that drive visual updates. In production code, a single visual effect emerges from callbacks, asynchronous handlers, and chained computations dispersed over many files. Novices cannot easily determine starting points or hold all intermediate state in working memory. Yet, understanding variable-level data flow matters as it lets developers diagnose UI bugs quickly, reason about performance, and extend features confidently—skills critical for becoming an engineer who can maintain complex front-end systems.

Ideally, front-end developers should be able to (a) trace a user-facing DOM change back through every intermediate JavaScript variable transformation that causes it, and (b) explain in plain language *why* each variable exists and how it fits the data-flow chain. Such knowledge is

¹This chapter includes some starter work and writing done in conjunction with Maxine Whitely, Jonathan Liu, Natalie Brewster, Yabi Ayele, Nell O’Rourke, and Haoqi Zhang. The author of this dissertation has done their best to indicate sections with significant collaborative writing contributions from SE team members. Source code contributions can be found at <https://github.com/gobid/scaffolded-exercises> and <https://github.com/gobid/scaffolded-exercises-interface>.

rarely taught however. Textbooks and online tutorials tend to showcase isolated snippets or toy widgets; they rarely follow value flow across dozens of functions in complex professional code. Also, debuggers surface everything at once, overwhelming novices with low-level detail.

As a focal example, consider a novice trying to inspect the “drag map” interaction on the XKCD comic shown in Figure 4.1. In early needfinding, we found that novices using Chrome DevTools (CDT) commonly stumble into a chain of function calls that looked unrelated: the code references *X position*, *tile center coordinates*, or *offsets*, each living in a specific section of the code. Without a structured exploration strategy, they typically open random files or sections in the *Sources* panel, place breakpoints, see that *X position* changes, and then lose track of how that leads to new image loads. As a result, novices *fixate on local data changes* (e.g., seeing the value for *X position*) but do not connect them to deeper transformations (e.g., *tile center coordinates* → *tile image URL*) in other parts of the code. This confusion also occurs when they jump to the *Elements* tab to review DOM changes but fail to see how those changes link back to the original *mousemove* event. In pilots, learners routinely missed tracing events in CDT, discovered some DOM updates at the surface-level, but did not connect them deeper in the complex call stack. As a result, they missed essential data flows and could not reflect on and integrate pieces of their knowledge into a cohesive story, leaving them with a fragmented sense of how the map reloading logic worked end to end.

Problem: As we have seen, in the domain of Javascript, existing tools like CDT often leave novices unsure of where to start, leading them to trace **complex code** in the order of execution, without **awareness** about how changes across multiple code components impact the DOM. This leads them into a chain of confusing function calls with variable values, which at the **surface-level** seem loosely semantically related to the system’s functionality/domain. For example, in the XKCD Comic (Figure 4.1), learners struggle to connect a map drag gesture and final DOM updates with

the deep variable transformations of mouse positions and tile centers that trigger image reloading [14]. This highlights the pitfalls of unstructured sensemaking, which make it difficult to link generic conceptions of data flow with example specific coordination.

Embedded in this thought space, we iteratively developed Scaffolded Exercises—a process management system that incrementally scaffolds learning about variable value transformations (i.e., one variable affects the value of another, which affects another, and so on). Provided with a demonstration of an interaction on a website that a novice wants to learn about (e.g., dragging the XKCD comic map to load new content as in Figure 4.1), SE auto-generates a sequence of exercises that guide the user through exploring and learning about code that produces a visual effect on the front end. Through a sequence of design iterations, we refined how SE’s structure could help learners make observations, break down complex code, map code execution to visual changes, and incrementally build conceptual models. This led to features such as better resurfacing of notes taken in earlier exercises to help learners build on prior knowledge. The resulting system has since been automated and refined, enabling controlled evaluation and broader scalability.

4.2 How SE enacts PM: Ordered Exercises, Live Variable Values, Reflection, and Resurfaced Prior Notes

SE supports process management through two key approaches: (1) reverse call stack sequencing that guides learners from intuitive DOM outputs to deep variable transformations and (2) structured reflection questions tied to interactive variable tracking and code comprehension. This helps learners trace functionality backwards through the call stack and in a conceptually chunked way.

System Overview: To manage this process, SE follows a sequence of exercises that reflects the structure of the value transformations being taught and the PM framework. SE (Figure 4.2) begins with a user (1a) observing a DOM-level effect, (1b) inspecting the live values of the driving

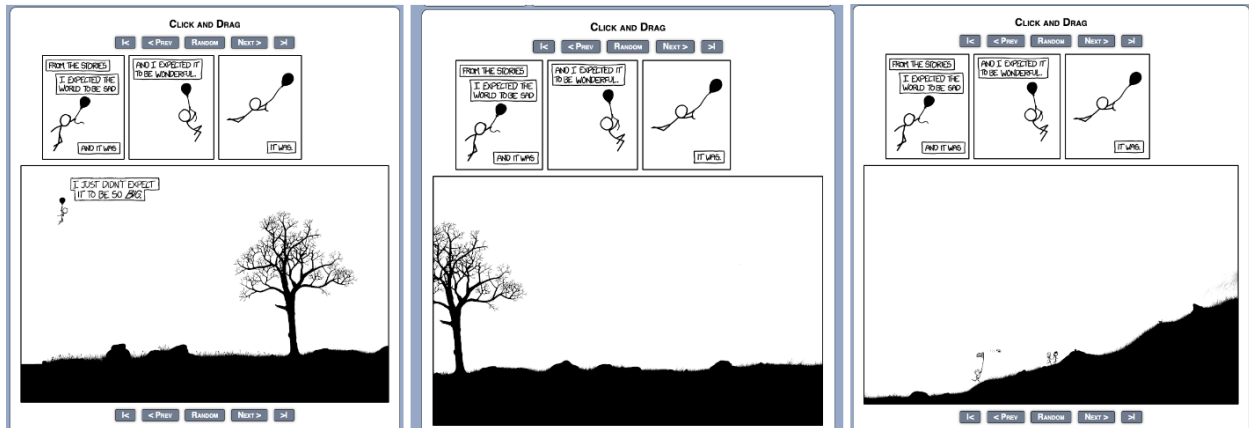
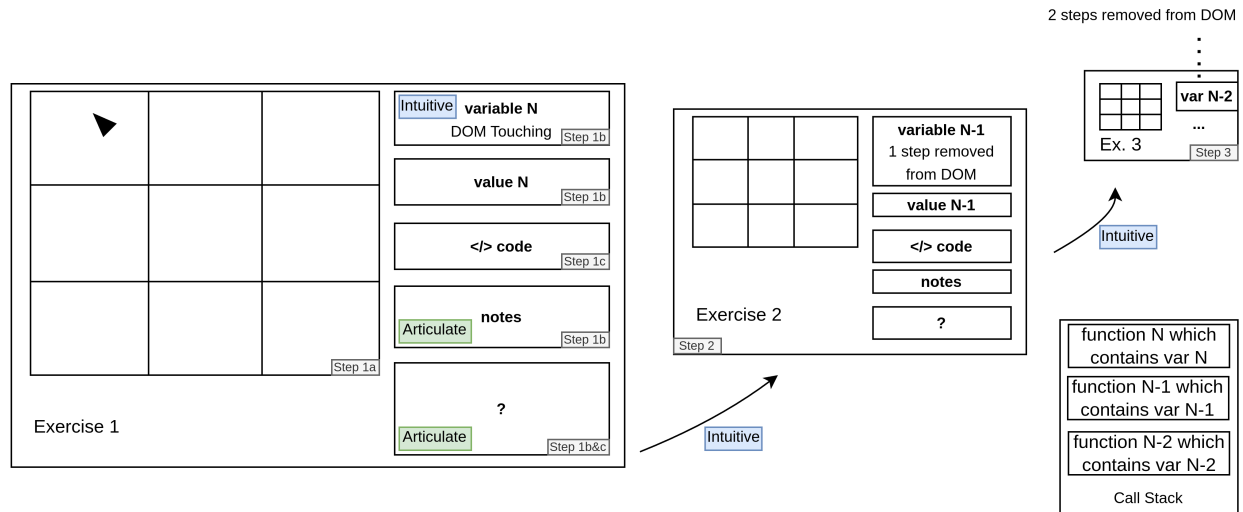


Figure 4.1: The XKCD Click and Drag example presents a draggable map of a comic world. In its implementation, mousemove events trigger the system to calculate a new page position for the user, which in turn is used to update the CSS positioning of a larger map div. The mouse position changes are also used to calculate a new value for a *centre* variable, which in turn is used to determine a *name* variable, which specifies which image tiles to load into the DOM.

variable, and (1c) answering reflection prompts about the surrounding code and related variables; it then (2) traces one level back and repeats, progressively (3) deep-diving into earlier computations (e.g. a *position* var affects *centre* which affects *tile*) and ensuring (4) full coverage of all DOM elements and variables. In addition to enacting PM, SE enacts Quintana’s heuristics scaffolding sensemaking because their suggestions around intuitive entry points, articulation, bound setting, diverse representations, and knowledge organization are known to significantly improve inquiry and sensemaking [26].

PM Solution: SE (Figure 4.3) is a PM system that provides learners with an ordered set of scaffolded exercises for sensemaking, where learners can progressively build on their intuitive, prior understanding to advance further understanding. The three core PM principles behind SE that shape learners’ sensemaking pathways are:

1. **Reverse Call Stack Sequencing for an Intuitive and Followable Sensemaking Order:** SE leverages *reverse call stack sequencing*, where learners start with exercises covering **visually**



SE Steps and Features:

Step 1a - DOM: The first exercise should start with intuitive aspects like inspecting visual DOM interactions and DOM updates, apparent to the end user.

Intuitive

Step 1b - Variables, Prompts, Notes: Then the exercise should present learners variables and values that directly affect the DOM, and ask about them. It should let learners take notes on variables for future reference.

Bounds Articulate

Step 1c - Selective Code Reading and Prompts: Next, the exercise should present learners with code containing those DOM touching variables and values, and ask about said code.

The code snippets (annotated with syntax definitions or "Tutorons"), variable values, and visual site (DOM elements) comprise multiple views that help the learner make sense of the example.

Multiview Articulate Organize

Step 2 - Move on to other exercises: SE should then move to exercises that follow the same structure as the first exercise described in Step 1, except around different variables / code.

These subsequent exercises will cover code involving precursor variables whose values the DOM-touching variables of step 1b depend on. SE should present said precursor variables, values, and related code snippets and ask about them through reflection questions.

For instance, SE should ask how precursor variables transform into DOM-touching variables. It should allow learners to enter answers, while referencing notes from previous exercises.

Intuitive Organize

Step 3 - Deep Dive: SE should move deeper in the call stack, repeating step 2 for variables that impact step 2's variables, and so on.

Intuitive

Step 4 - Coverage: SE should move to other DOM elements than the one we started with in Step 1, until the learner has studied all interactive DOM elements on the webpage. This is so they will have organized and linked their knowledge around all aspects of the codebase, from surface level DOM interactions to variable value transformations deep in the call stack.

Intuitive Organize

Heuristics from the Learning Sciences:

Quintana's Scaffolding Design Framework Heuristic 1b (Intuitive)

"Use descriptions of complex concepts that build on learners' intuitive ideas." Build on prior knowledge. Provide an intuitive starting point.

Quintana's Heuristic 7c (Articulate)

"Facilitate articulation during sensemaking."

Quintana's Heuristic 4a (Bounds)

"Restrict a complex task by setting useful boundaries for learners."

Quintana's Scaffolding Design Framework Heuristic 3b (Multiview)

"Enable learners to inspect multiple views of the same object or data."

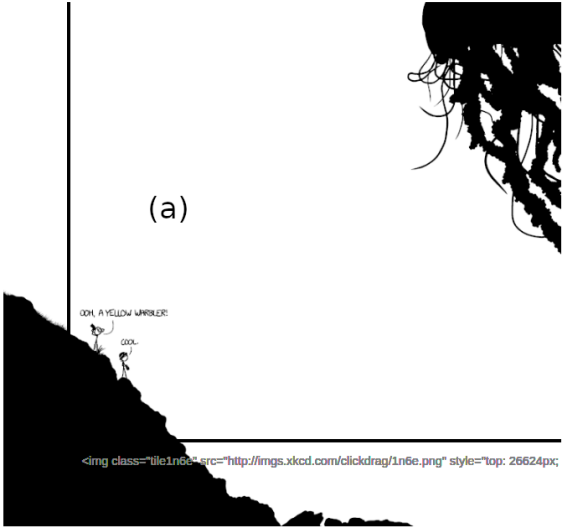
Quintana's Scaffolding Design Framework Heuristic 6b (Organize)

"Facilitate the organization of work products."

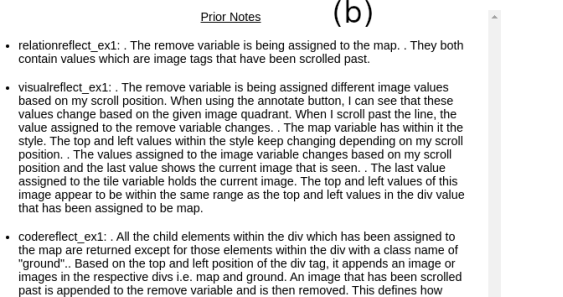
Figure 4.2: Here we present a conceptual diagram of how SE works and enacts Quintana's heuristics, as well as several PM heuristics. The system presents exercises to the learner, starting with intuitive (PM1) DOM interactions (Step 1) and moving to variable value transformations deep within the call stack (Steps 2 and 3). The chunked exercises (PM3, PM4) provide learners with live updating variable values as they interact (PM5) with the webpage as an end user. The system prompts learners to articulate (PM8) their understanding of the semantics of variables and relevant code snippets while referencing their prior knowledge (PM2) in the form of their notes from previous exercises. This process repeats until they achieve coverage of all important variables and code (Step 4).

Scaffolded Exercises

DOM Elements:



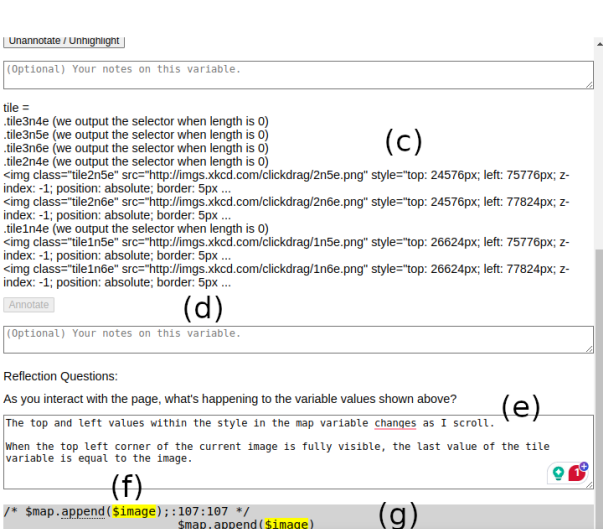
(a)



(b)

Prior Notes

- relationreflect_ex1: The remove variable is being assigned to the map. . They both contain values which are image tags that have been scrolled past.
- visualreflect_ex1: The remove variable is being assigned different image values based on my scroll position. When using the annotate button, I can see that these values change based on the given image quadrant. When I scroll past the line, the value assigned to the remove variable changes. . The map variable has within it the style. The top and left values within the style keep changing depending on my scroll position. . The values assigned to the image variable changes based on my scroll position and the last value shows the current image that is seen. . The last value assigned to the tile variable holds the current image. The top and left values of this image appear to be within the same range as the top and left values in the div value that has been assigned to be map.
- codereffect_ex1: All the child elements within the div which has been assigned to the map are returned except for those elements within the div with a class name of "ground". Based on the top and left position of the div tag, it appends an image or images in the respective divs i.e. map and ground. An image that has been scrolled past is appended to the remove variable and is then removed. This defines how



(c)

(d)

(e)

(f)

(g)

(h)

(i)

Figure 4.3: SE prototype on XKCD: It features (a) annotatable elements on the page, (b) resurfaced prior notes, (c) live variable values and DOM element tracking, (d) variable-specific notes, (e) reflection questions on DOM elements and variable values, (f) code views where live variables can be highlighted, (g) Tutorons, or hover-over definitions of key syntaxes, (h) reflection questions on code snippets, and (i) reflection questions on relationships among DOM elements and variable values. The code shown is dependent on which exercise within the overall ordering that the user is on. This figure's data was taken from P3's tool use on Ex. 2.

intuitive DOM updates (Figures 4.2 1a and 4.3 a), such as a tile reloading on map drag, and systematically trace backward from the surface-level to understand underlying value transformations deep in the call stack² (i.e., how code changes internal variable values that eventually affect the DOM). This structured ordering prevents novices from exploring arbitrarily and losing context. It helps prepare them to build on an initially intuitive picture of the system’s functionality.

2. **Linking Prior with Current Knowledge:** Each exercise references variables or functions discovered in previous steps, prompting learners to recall **earlier insights** and build cumulative understanding of the complex knowledge structure (e.g., variable value transformations in this case) that SE targets. This prevents disconnected “one-and-done” observations (i.e., seeing some complex variables too early and missing others in the data flow) and fosters an integrated mental model (i.e., by surfacing variables at the right level of depth and complexity at the right time).
3. **Connecting Conceptual Insights to Concrete Values:** SE shows live variable transformations alongside reflection prompts and the code itself, ensuring that learners repeatedly verify their conceptual inferences (e.g., *position* to *centre* to *\$image* updates). By comparing real-time variable values with their conceptual models, novices gain clarity on how abstract ideas **map** to actual example-specific code behavior. Doing this helps them uncover the knowledge structure being taught, namely, variable value transformations.

Beyond the application of these core principles, by chunking each step into an exercise, SE aims to **reduce cognitive overload** (PM4) and supports learning within one’s zone of proximal development [35]. Each step surfaces only the relevant variables and transformations (Figure 4.2

²The original (non-reversed) execution orders starts from the bottom of the call stack, stacking on new function calls, and moves up to the top of the stack.

1b/2-4). Moreover, the progression aims to help users make semantic sense of internal variables, program state, and how the two drive visual effects—insights that traditional debugging tools such as CDT do not explicitly scaffold [26].

Unlike traditional debugging tools, SE provides ordered, goal-driven exercises that begin with **intuitive visuals** (PM1) and enable *value transformation exposure*, the study of changes in variable values across code components. SE surfaces **interactive live variable values** (PM5) (Figure 4.3 c), **salient code snippets** (PM3) (Figure 4.3 f), and hover-over syntax explanations (Figure 4.3 g) that intend to **scaffold away complexity** (PM4). Each exercise includes structured prompts promoting **reflection** (PM8) and note-taking [26] (Figure 4.3 d) to support incremental sensemaking of variables. These notes resurface across the ordered exercises (Figure 4.3 b), aiming to help learners link **prior and current knowledge** (PM2), reinforcing their architectural understanding of data flow and relationships (Figure 4.2 2).

Additionally, when **comparing related variables** (PM6) in SE, users can see how live values change in tandem, often **mapping** (PM7) from one representation (e.g., pixel coordinates) to another (e.g., image tile coordinates), and from generic conceptions of data flow to example-specific manifestations in key variables (e.g., *position*, *centre*, *name*, *image*, etc.). This process-oriented guidance aims to prevent unhelpful exploration, such as struggling with overwhelming, scattered, and seemingly unlimited program state information. It scaffolds novices in adopting expert-like sensemaking strategies, helping them build architectural understanding for future web development work.

Structured Reflection: SE’s questions (Figures 4.2 1b/1c, 4.3 e, h, i) prompt reflection on: (1) changes in state and visuals as live variables update on interaction; (2) code-to-output mapping, with cues to annotate DOM elements, highlight relevant variables in the code, and view syntax definitions (Figure 4.3 a, c/f, g); (3) connections to prior exercises and their insights. This aims to

reduce unfocused exploration and helps learners develop accurate mental models incrementally.

Real-Time Value Transformation: SE averts the need for setting breakpoints, which is non-salient work when one’s goal is to learn about value transformations rather than to debug code [26]. Instead, SE auto-tracks and surfaces live variable updates, both local and global (Figure 4.3 c). This allows users to observe transformations dynamically as they interact, reinforcing their understanding of data flow through code components.

By integrating these features, SE enacts process management, scaffolding learners toward expert-like sensemaking strategies and helping them move beyond isolated execution details to a conceptual understanding of full-stack architectures (Figure 4.2 Steps 2-4). While we have described how SE enacts PM, Figure 4.2 shows how SE also enacts several of Quintana’s heuristics directly, including ones not present in PM, such as providing multiple views (e.g., code, variables, DOM) of objects and data. In this way, we can consider SE doubly compliant [26].

4.3 SE System Implementation: Reverse Call Stack Sequencing

We now review techniques for automatically³ creating exercises:

Implementation Approach: First learners interact with an instrumented copy of a professional website, and then SE generates process management exercises based on those interactions. To support this sequence, SE software employs two core mechanisms: (1) code instrumentation that enables program state and function call capture, and (2) an exercise generation algorithm that orders sensemaking exercises in reverse of the original call stack⁴. To enable live variable tracking

³There are currently a few hardcoded XKCD-specific directory paths that create new files in the right spots that would have to be changed to run SE on a new site. Some static file links and endpoints for storing collected data from instrumentation require manual adjustment or setup. Future work can generalize these paths and/or create a config template file for practitioners to recreate for every new website they want to run SE on. The rest of the system is sufficiently general to work on any site.

⁴Equivalent to reverse function run log order, in non-async cases.

and runtime function call logging, SE instruments a target website’s JavaScript using a tracing engine that wraps DOM interactions and variable assignments with callbacks. These callbacks populate a centralized run log and update a global state manager that classifies variables as DOM-touching. Using this run log of function calls, SE automatically generates interactive exercises, starting from the most recent DOM-touching operations (e.g., visual changes). Then, SE walks backwards in time through other code components in the run log that reference the same DOM elements or precursor variables (variables that the previously generated exercise depends on). Each selected snippet becomes a scaffolded learning step, showing variable transformations, enabling note-taking, and resurfacing prior observations. Details follow.

Code Instrumentation: To create exercises in reverse of the code’s original execution order, SE instruments over all the JavaScript for a given website and wraps key variable updates, method calls, and function calls in a stack tracing call (e.g., *StackTrace.instrument(() → \$map.css(...)* , *callback(“%map.css:51:57”))();*).

StackTrace attaches the instrumented code to callback functions, which are triggered whenever the instrumented code is invoked. The callback functions are fed the line numbers of the instrumented code to ensure that we consider each code snippet unique, even if that code content appears in multiple places in the codebase. While the purpose of *StackTrace* is to build a function call run log, we use similar techniques to surface the live variable values.

When SE instruments a site’s JavaScript to trace execution, it injects a global state manager object (hooking into each variable declaration and assignment so it always has the latest value, keyed by scope location to avoid collisions), thereby enabling retrieval of any program variable regardless of scope. This state manager stores information about which variables are DOM objects.

Run Log Creation: When the instrumented code runs as users demonstrate an interaction on a website of choice, the system returns the complete trace or run log of the function calls and variable

updates in order of execution. This instrumentation works for any JS program, demonstrating SE’s potential to work across diverse websites in the wild.

The DOM-touching First Exercise: SE then goes through the run log from the most recent call backward until it finds one that directly updates the DOM. It selects this code as the first exercise.

Exercise Ordering Algorithm: We leverage Abstract Syntax Trees (AST) to analyze function call code snippets or “nodes”, and build ordered exercises around each. For the subsequent exercise after the first, we keep our “reference” as the DOM object that the previous exercise’s code modified and focused on. We iterate through function calls in the run log from the most recent backwards. This works because the run log order and call stack order are effectively the same in our case, with the most recent DOM-changing events at the top of the stack, the variable value transformation sequence in the middle of the stack, and the initial user input detection at the bottom of the stack. We iterate until we find an unvisited node that initializes, modifies⁵ (e.g., `document.getElementById('square').width = '150px';`), or uses (e.g., reading a div’s width without changing it) the reference or other DOM objects present in the previous exercise. We then generate an exercise for that node. If there is no such unvisited node, we iterate through the reversed run log, look for a new “unconnected” node that modifies or uses a DOM object, and repeat all the above. After exhausting the DOM-touching nodes, we look for precursor variable-touching code snippets in the reversed run log. Visiting these is key for identifying intermediary variables and ensures full code coverage. After exhausting these, we are done as we have covered all the DOM object-touching code and their precursor variables in an optimal, interrelated sequence.

Code Classification Methods: To determine if a code component is an initializer rather than a modifier or user, SE marks the time DOM content was fully loaded and records the number of

⁵The first exercise’s code/node already modified the reference, so this refers to yet another node that modifies the reference DOM object.

times each code component is called. If a code component is called only once and before the DOM is fully loaded, then it is an initializer. To classify code components as modifiers or users, we used the state manager to check whether they changed any DOM variables. If they did, they are labeled modifiers; if they did not, users⁶.

Interface Implementation: We also developed the UI for SE (Figure 4.3). We automated⁷ the process of importing generated exercise orderings, including exercises’ code snippets, relevant variables, and DOM elements. Our system automatically produces the relevant JavaScript (React) and HTML for the exercises displayed on the SE interface. Each exercise consists of one JavaScript / HTML file pair that implements the features of Figure 4.3.

Running SE on the XKCD Comic Map: For the XKCD example, the exercise ordering algorithm means that exercises start with the CSS positioning of the map *div* (a surface-level DOM update) and the removal and display of image tiles (also surface-level DOM updates). They then proceed to the causal factors behind appending and loading of images, expanding the code context to include the conditionals and loops over a grid in an “update” function, whose logic governs how a “centre” variable is calculated from “position” and then transformed into “image tile names” (deeper value transformations). Together, these form eight auto-generated exercises, which are presented in the SE user interface.

⁶We acknowledge that some DOM elements (e.g., dynamically added XKCD map images) may be created post-load, so we may not find initializer nodes for them. While DOM-object resolution and modifier/user classification are imperfect and non-exhaustive, this does not affect our design claims: exercises still present DOM access before internal value updates, and modifier/user ordering has minimal impact.

⁷Our exercise generator is general, but we passed in some XKCD-specific variables and elements to handle skips, unfurling, and aesthetics. These can likely be automated (e.g., via LLM-based classification). While our instrumentation supports functional code, current support for developer-defined classes and nested calls is limited but addressable in future work.

4.4 Controlled Think-Aloud Evaluation of SE to Gauge Learning

To evaluate SE’s process management and gauge learners’ exploration and conceptual learning about variable value transformations, we conducted a between-subjects, think-aloud evaluation. The study comprised 10 advanced novices, five using our system and five using a control. We measured the effectiveness of the PM affordances, such as SE’s incremental exposure through ordered exercises.

Participants were recent university graduates and students involved in Computer Science, IT, engineering, etc. To recruit them, we sent public blasts in tech-focused Slack organizations and channels⁸. Participants were clearly informed that the study was educational and voluntary, with the right to opt out at any time. The sessions were conducted and recorded using Zoom, leveraging video, audio, chat, screen share, and remote control, as needed. In some cases, when participants had internet issues, we leveraged chat-based communication rather than audio, but we still used the screen-sharing facilities of Zoom and followed the same study procedures.

Screening: To ensure we were testing on advanced novices who understand JS syntax and constructs, we screened for individuals who reported having at least two web development experiences from coursework, projects, internships, or jobs. Additionally, SE participants (both conditions) needed to have (a) written 200-5000 lines of JS and (b) passed at least two-thirds of the questions on the SE screener’s quiz covering loops, forms, positioning, and other general JS topics. Questions featured multiple screenshots of websites, asking candidates to pick the one generated by a particular code snippet. As a quick *spot-check* during the study, we asked participants to re-answer one or two of the screener questions to confirm their authenticity. Inability to articu-

⁸Recruitment: The experimenter has hired part-time contractors through such channels and/or holds leadership roles there. To mitigate potential conflicts of interest, participation was voluntary, recruitment was conducted via a neutral public blast, and the opportunity was presented separately, unrelated to any contract work. Participants were informed during the study that they could opt out at any time. Learning sessions were neutral and standardized, with study materials kept confidential and feedback collected at the end of each session.

late a coherent explanation—regardless of correctness—would have raised a red flag suggesting potential dishonesty. However, the experimenter observed no such concerns.

Conditions and Study Phases: Participants were randomly assigned to one of two groups: experimental or control. Both groups participated in three phases: a pre-test, an interactive learning activity with the SE tool, and a post-test (identical to the pre-test).

Control Details and Training: To isolate the effects of process management, we designed the control tool (Figure 4.4) as a version of SE without key PM affordances but with state-of-the-art alternatives. This version had live variable values but none of SE’s core features for making sensemaking intuitive, such as ordered exercises based on chunked code snippets, each with exercise-specific reflection questions and resurfaced prior notes. They also had access to CDT, the current state-of-the-art JS inspector, and a bulleted list of activities to consider, such as inspecting page visuals, noting variable values, linking code with visuals, and investigating relationships between variables. The activities cover everything SE aims to highlight, but in a static rather than process-managed manner. Control users were trained through a video on how to use CDT, including features like inspection, setting breakpoints, and viewing live variable values. Control users were asked to replicate the actions of the video on a demo example and reference an accompanying CDT cheatsheet to ensure they understood how to use the tool and were well prepared for the session.

On the other hand, the experimental tool, SE, was generally self-explanatory and did not require training.

Protocol: Participants were asked to explore XKCD as an end user and take a pre-test, a written web form consisting of open-ended questions about the examples’ implementations. The SE pre/post-test covered the XKCD comic map (xkcd.com/1110), focusing on JS implementation, data flow, and intermediary variable values. It asked (a) *What user inputs change the DOM?*, (b)

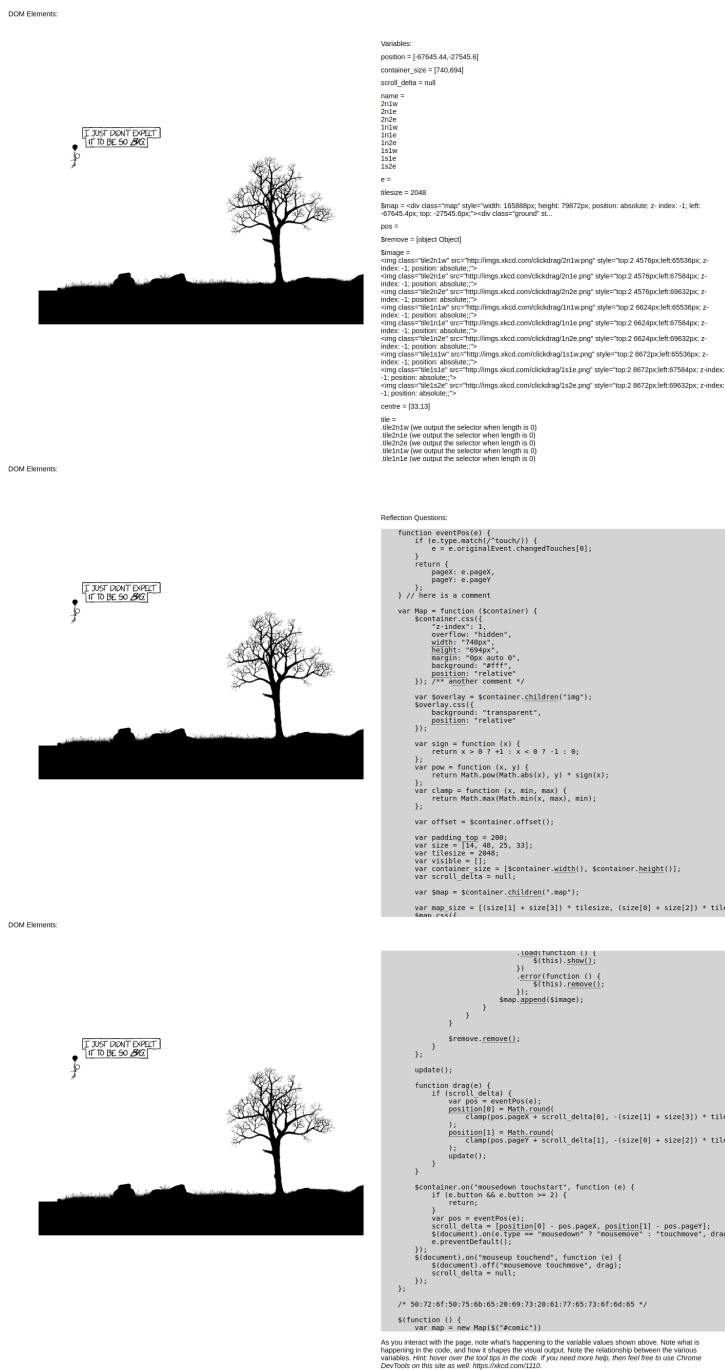


Figure 4.4: To isolate the effects of process management, we designed the control tool as a version of SE without key PM affordances but with state-of-the-art alternatives. This version had live variable values but none of SE’s core features for making sensemaking intuitive, such as ordered exercises based on chunked code snippets, each with exercise-specific reflection questions and resurfaced prior notes. They also had access to CDT, the current state-of-the-art JS inspector, and a bulleted list of activities to consider, such as inspecting page visuals, noting variable values, linking code with visuals, and investigating relationships between variables.

How is the interaction implemented?, and (c) *Can you write pseudocode to build the interaction?*. Additionally, the tests included a diagramming activity to *explain the example's implementation visually*.

Tool use: Participants then used either SE or its control. While they interacted, we paid attention to their exploration process and which tool features helped them overcome misunderstandings or led them to “aha” moments around learning outcomes (Table 4.1).

Nature of Studies: To support authentic learning, participants were permitted to look at the code and use search engines during tool use but not during the pre and post-tests, which were taken from an end-user perspective. All participants had as much time as they liked in each part of the study and were compensated at 20 USD per hour for time spent actively participating in study activities.

Post-test and survey: Finally, participants completed a post-test identical to the pre-test and a post-study survey during which they reflected on how useful the overall tool and core tool features (e.g., exercise ordering, prior notes, etc.) were for reaching their learning goals. They described their tool use and provided ratings on a 5-point Likert scale (1-5 or *not helpful* to *extremely helpful*). They were asked to be honest in rating the systems' effectiveness.

Inactivity: The experimenter made a best effort to exclude periods of inactivity or natural distractions/breaks such as phone calls and used best judgment to capture and correct study or tool use timings due to unintended/accidental delays in starting/stopping recordings. All participants had the ability to contact the experimenter with feedback.

Timing: Participants were informed upfront that the study would take several hours, engaged in informed consent, and were explicitly told they could withdraw at any time. To ensure participant well-being, we provided flexible scheduling, modular study itineraries, and both short and multi-day breaks, allowing participants to explore concepts at their own pace without fatigue. Given

the complexity of professional programming patterns (as discovered in pilots), we intentionally allowed participants as much time as they wished to engage in meaningful, structured learning.

Breaks: Participants in both conditions could take as many breaks as they wanted for as long as they wished but were asked not to refer to study materials during breaks. Participants were free to choose when to take breaks and could split the study across multiple days, similar to a self-paced learning assignment. These breaks were natural and did not impact study integrity, as participants received a recap after extended breaks as needed to ensure continuity. The recaps involved showing participants the last few minutes of their past session’s recording (or screenshots).

Learning Orientation: Participants were told the study aimed to evaluate the tool, not their performance.

Wizarded Interventions: Interventions the experimenter made to ensure participants in both conditions had a fair chance to benefit from the learning experience included encouraging participants to explore the example(s) thoroughly, to tinker liberally (e.g., moving around the comic map), to focus on web interactions rather than the content of the comic itself, to use the features available to them fully, especially when stuck, to strictly follow the exercise order as applicable⁹, and to *say more* (up to three times) for the questions. Further assistance included acquainting the participants with the UI at hand, refreshing participants on where they left off if they took a break, restating insights they had already reached to follow up on the meaning, asking who/what/where/when/why/how follow-ups to elicit more detail, prompting them to reread the question carefully, using encouraging language (e.g., “good work/progress”), asking them about which variables each one affects if they don’t make it clear, reminding them of definitions, asking about anything internally inconsistent in their language, and correcting one-off UX errors. Participants in both conditions were instructed to heed variable value transformations across connected

⁹Experimental condition users of SE were expected to proceed linearly through activities.

Num	Learning Outcomes Rubric Item
1	<i>\$map</i> has images (multiple HTML <code>< img ></code> instances) appended / loaded onto it
2	Each image tile has a unique name - <i>name</i> is being used to determine the image's src
3	The names are labeled in some sort of coordinate system (nswe) - all <i>name</i> variables are combinations of integers and n,s,e, or w
4	Image <i>name/tile</i> coordinates are based on the <i>centre</i> variable
5	<i>centre</i> refers to the center tile that the user is seeing
6	There is a double for-loop that iterates through a 3x3 grid of images that are appended to the <i>\$map</i> div.
7	Once some threshold is passed (i.e., the <i>centre</i> changes), then a new set of PNG images loads in. It's okay if they say new images are loaded "upon need."
8	<i>centre</i> coordinates are proportional to <i>position</i> coordinates.
9	As you move your mouse left / right, up / down, just as <i>position</i> changes, <i>centre</i> also changes accordingly, direction-wise and magnitude-wise.
10	The left and top CSS attributes of <i>\$map</i> are set by the <i>position</i> variable. Even if they mix up left and right or top and bottom, it is acceptable.
11	<i>position</i> , <i>centre</i> , <i>tile</i> or <i>name</i> , <i>\$image</i> all change as you "drag" ("mouse-move" while "mousedown", or just "mouseover", or some mouse moving related event) through the comic <i>\$map</i>
12	<i>position</i> contains pixel coordinates that track user mouse input

Table 4.1: SE pre/post-test learning outcomes grading rubric. Each item corresponds to one point. Partial credit details omitted for brevity.

code components.

Interventions the experimenter made during the pre/post-test to ensure accurate assessment included reminding the participants to explore the examples' UX thoroughly, prompting them to share more about how/why the example works $\sim 1-3$ times for the questions, restating insights they already reached, asking them to recall their tool use, asking who/what/where/when/why/how questions to elicit more detail, having them write what they said aloud, and asking them if they considered everything the question covered (e.g. adding arrows to diagrams to annotate relationships and data flow).

Additional SE-specific Interventions: We also encouraged participants to move around in the comic world extensively while interacting (automatable in the future with rules triggered by site analytics), to focus on new information upon entering each subsequent exercise (automatable using interface prompts), to use variable notes and refer to prior notes upon starting a new exercise (automatable using interface prompts), and to use affordances they hadn't used yet before moving on, such as hover-over tooltip definitions and highlighting code values on the code snippet-related question, and annotation affordances when exploring elements and variables (rules can trigger learners to use these features when they aren't doing so).

Future work can explore how LLMs can potentially automate experimenter interventions in production settings [128], [129].

Scoring and IRR: To evaluate whether SE can help learners understand JS variable value transformations, we coded the pre and post-test according to a rubric of reachable learning outcomes around variable value transformations (e.g., how mouse positions transform into tile centers which influence image loading) (Table 4.1). Points were summed to obtain pre and post-test scores, which were used to measure knowledge gains (post minus pre-test score). We also recorded participants' tool interactions to track tool usage. We followed a formal qualitative coding procedure,

with two scorers independently coding the pre and post-tests to ensure consistency.

Inter-rater reliability (IRR) was measured and substantial agreement was reached. Our IRR process followed standard qualitative practices using a mixture of percent agreement, weighted kappa, and case-by-case discussion. We established the codebook using 10–40% (up to 60% for some codes) of pre/post data, and then independently coded the rest in small batches with check-ins and field notes. Sparse codes (predominantly 0s/1s) and ones with too little data left after establishing the code used percent agreement only, not weighted kappa, in order to avoid statistical distortion. Our protocol excluded circumstances requiring case-by-case discussion on every code from IRR metrics, as they reflected collaborative resolution. Minor inconsistencies (4%) (e.g., clear typos, section reference errors, rubric mislabel of a correct explanation) were resolved or excluded based on protocol notes and coder intent.

We report (a) percentage agreement where possible (83.5%, $\sigma = 12\%$ over 12/12 codes), (b) Cohen’s weighted kappa on the subset of codes (0.75, $\sigma = .19$ over 12/12 codes) from (a) where the weighted kappa metric could be applied, and (c) the remaining codes that required case-by-case discussion on all data (0 here for SE).

4.5 SE Evaluation Results: Deconstructing the Effect of Scaffolded Process Management in SE

Through a 5v5 evaluation, we found that SE with process management helped learners achieve greater knowledge gains than control participants. SE users had an average gain of 8.2 ($\sigma = 2.0$) out of 12, while control users had an average gain of 2.9 ($\sigma = 2.5$) out of 12 (see Table¹⁰ 4.2, Figures 4.5, 4.6 for participant-specific data points).

¹⁰For “discussed live variable values” and “fraction of vars in code discussed” (Table 4.2), we verified field notes against tool use recordings and study logs. Due to occasional recording delays or missing some text areas, we report a conservative lower bound in the table, spot-checking raw data where needed.

Learning Outcomes and Benchmarks	P2	P3	P5	P7	P10
pre-test score	1.5	3.5	2.5	0.5	4
appending images	E1-v	E1-v	E1-v	E0-c	E1-r
unique tile naming	E4-r	E1-v	E7-r	E4-c	E2-v
coordinate system		E7-c	E4-v	E5-c/E7-r	E7-cr
image/centre relationship		E7-c	E7-vr	E7-vc	E7-cr
loading 9 images		E4-c/	E7-v	E7-c	E7-cr
thresholding	E7-c	E1-c	E2-v	E1-c/E7-c	E1-r
centre proportional to/dependent on position	E7-v	E7-v		E7-c	E7-r
centre changes based on position	E7-v	E7-c	E7-r	E1-v/E7-r	E7-r
JS sets top/left of map	E1-vr	E1/2-v	E1-v	E1-v/E5-c	E5-r
changing values as you drag	E7	E7-c	E7-c	E5-c/E7-r	E7-r
position has pixel coordinates	E8	E7-v	E8-c	E7-v	E7-r
post-test score	9.5	9	11	11.5	12
knowledge gains	8	5.5	8.5	11	8
Num times resurfaced prior notes	4/8	3/8	3/8	5/8	2/8
fraction of vars in code discussed	22/27	20/27	20+/27	26+/27	27/27
discussed annotated elements	10/12	9/12	11/12	11/12	8/12
discussed live variable values	26+/32	28+/32	30+/32	30+/32	29+/32

Table 4.2: Table showing SE users' pre/post-test scores and tool use metrics such as the exercises (E0-E8, E3 skipped, 8 total) and contributing subsections (v=variables, c=code, r=reflection) if we could pinpoint them, where users reached specific learning outcomes. In most cases, participants eventually reached the full learning outcome. In some cases, we included an exercise that got them to partial understanding or helped them build understanding before the '/'. For cases where participants did not reach the learning outcome or where the contributing exercise could not be clearly identified, we left the table cell blank.

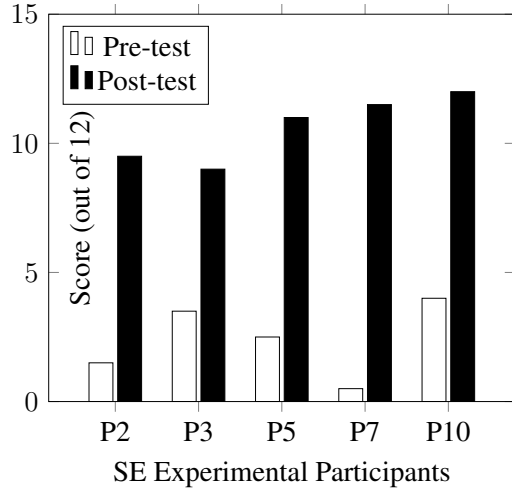


Figure 4.5: Comparison of pre-test and post-test scores for SE experimental users.

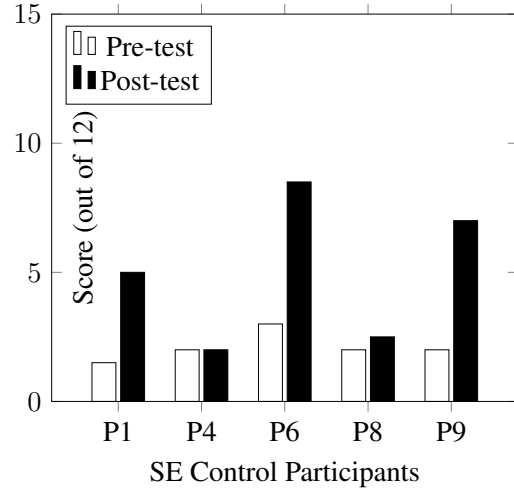


Figure 4.6: Comparison of pre-test and post-test scores for SE control users.

Quantitative Analysis: With 10 participants (5 in each condition), this study is necessarily exploratory, yet the effect size was large enough to yield a statistically significant result. We conducted a two-tailed Mann-Whitney U test (using *method="exact"* in *scipy*), which can be applied to small data samples that do not necessarily follow normal distributions. The Mann-Whitney U statistic was 0.5 ($p = 0.016$), indicating a significant difference between the groups. The effect size demonstrated a substantial difference ($r = 0.76$, $rb = 0.96$). To assess robustness and sensitivity, we confirmed via 10K power simulations that our observed effect sizes return a power > 0.8 under several plausible distributions¹¹. Still, future evaluations of more users and

¹¹We computed this through an analysis of 10,000 simulated 5v5 experiments and their Mann-Whitney U statistics for each distribution. Since the underlying distribution is unknown, we conducted the simulations using several commonly used distributions for independent test score gains, including normal, gamma, Weibull, and lognormal. To meet the strictly positive support requirements of the gamma, Weibull, and lognormal distributions, we shifted all test score gains upward by a constant ($number\ of\ codes + 1$), a transformation that preserves the rank order and, therefore, does not affect the Mann-Whitney U test outcomes. Prior to data generation, these distributions were fit to our observed data and effect size, which we argue are representative given the practices followed in our experiment design. We also performed bootstrapping with replacement. In all five cases (four distributions and bootstrapping), we found that the simulated power was > 0.8 . Still, this study was intended as exploratory, and future studies should use a priori power analyses, additional sensitivity checks, and confidence intervals for power.

more in-the-wild production-grade PM systems¹² can support our arguments and back stronger claims around SE and PM’s generalizability.

Qualitative Mechanism Analysis: We found SE users built upon knowledge from prior exercises and reached “aha” moments around variable value transformations in 4 out of 8 exercises on average. 4 out of 5 SE users described the chunked exercise ordering as helpful (*extremely* or *somewhat*). While there was no substantial difference between how the users rated¹³ the experimental ($\mu = 4.2, \sigma = 0.62$) and control ($\mu = 4.5, \sigma = 0.51$) tools overall, the participants in the experimental condition seem to have engaged in sensemaking more effectively and learned much more about variable value transformations. The control participants focused mainly on events and line-by-line control flow, while the experimental participants explored data flow between intermediary variable values. Given the data, we can now answer our research questions on exploration and learning with SE.

We now examine exploration (RQ1) and learning (RQ2) of SE and control participants.

Experimental Participants’ Exploration (RQ1): The experimental participants highlighted the significance of the exercise ordering, chunked code snippets, reflection questions, and annotations in building their understanding step-by-step. P7 emphasized how the structured progression allowed them to deepen their comprehension incrementally, stating, “*The gradual introduction of each part of the code and the functions for updating the map followed a logical progression that allowed me to build my understanding.*” This understanding progression is visible in their pre (Figure 4.7) and post-test (Figure 4.8) diagrams, which demonstrate an improved grasp of variable value transformations (e.g., *centre* $\rightarrow$ *name* $\rightarrow$ *tile* $\rightarrow$ *image*). Participants also found chunked

¹²Given the extended periods learners require to learn these expert concepts, further manual prototyping is not a feasible solution.

¹³We asked them at the (a) beginning and (b) end of the post-study survey how helpful the tool was in helping them understand (a) the example’s implementation and (b) its variable value transformations. These four answers were averaged across all the participants in a condition.

snippets particularly helpful for avoiding cognitive overload. As P2 noted, *“If the whole code was presented to me at a go, it would have been overwhelming, and I wouldn’t have been able to figure out what portion of the code is leading to what.”* Moreover, reflection questions and note-taking affordances made building on prior knowledge possible. P2 shared, *“At the very last exercise, which contained the overall code, I did refer to the notes from previous exercises that included smaller snippets.”* Participants reported that the reflection questions helped guide their focus. As P7 explained, *“These questions prompted me to think more deeply about the code and its components like variables and relationships.”* **Interactive annotation and highlighting features** (PM5) also played a critical role by offering clarity, with P10 adding, *“Annotation helped me figure out image boundaries, which simplified my understanding.”* The combination of progressive exercise ordering, chunked code, annotations, and reflective questions may have enabled participants to build a cohesive understanding of the example, transforming fragmented learning into a meaningful exploration of code and value transformations. In the end, among the experimental condition participants, 4/5 said the **exercise order** (PM1) was helpful (extremely or somewhat), 5/5 said **annotation/highlighting** (PM3, PM4) was helpful, 4/5 said the **reflection questions** (PM8) were helpful, and 5/5 reported referring to their **prior notes** (PM2) at least twice. These self-reports align with the experimental participants’ heavy use of these features, as shown in Table 4.2. The table also shows how nearly all exercises¹⁴ contributed to participants reaching learning outcomes, with earlier ones generally helping with intuitive DOM touching activity and later ones helping with intricate intermediary variable relationships.

Control Participants’ Exploration (RQ1): The control participants focused on events and individual code components but struggled to capture the broader data flow between variables. P1

¹⁴A few SE participants completed parts of exercises they were told they could skip due to redundancy (e.g., Ex. 3, or variable sections of Ex. 4/5, or relationship sections of Ex. 6), likely due to missed reminders. We posit these redundancies likely did not affect the results or pose validity concerns.

variables like *position*, *centre*, *tile/name*, and *image*. This finding suggests that without structured guidance, participants tend to focus on the most visible aspects of the code (e.g., events) rather than the underlying value transformations.

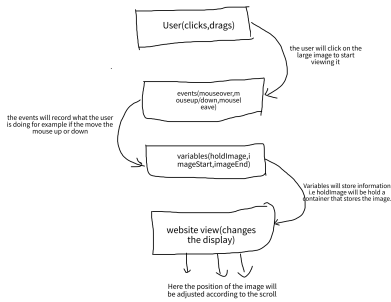


Figure 4.9: This diagram shows P8's understanding of XKCD before control/CDT use.

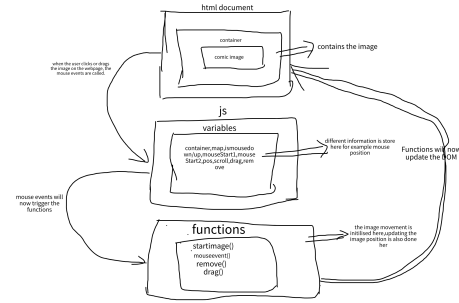


Figure 4.10: This diagram shows P8's understanding of XKCD after control/CDT use.

Experimental Participants' Learning (RQ2): The experimental participants appeared to demonstrate learning gains, especially in understanding the *position* → *centre* → *tile/name* → *\$image* data flow. P2 initially focused on basic interactions, noting, “*I was using the mouse... to shift the content,*” but later realized, “*The value of the position determined the value of centre and centre_last... any one value going wrong [would break] the whole chain.*” Similarly, P3 transitioned from broad descriptions like “*the system track[s] different coordinates and generate different graphics*” to a deeper understanding, explaining, “*The position variable had two values that change on scroll... The tile name is made up of two letters and two numbers... calculated*” dynamically based on the user's movements. P5 and P10 expanded on this, **mapping and comparing variables** (PM6, PM7), saying that as we update the *position* elements, the UI updates as well, and that the code calculates the *centre* and *tile name* to manage what images are displayed. P5's pre (Figure 4.11) and post (Figure 4.12) tests accordingly show improved understanding of the *position* → *image update* data flow in the XKCD example. Generally, experimental condition par-

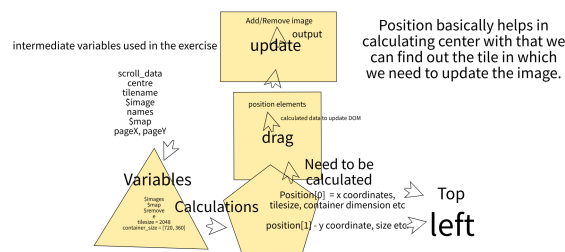


Figure 4.12: This diagram shows P5's understanding of XKCD after SE use.

Control Participants’ Learning (RQ2): Despite some improvements by the post-test, control participants generally struggled with grasping intermediary variable values and data flow. P1 mentioned, “*Intermediate variables: size, container_size, tile_size, tile_name, position, scroll_delta, name,*” but their **surface-level** description of how these variables interacted offered less clarity, making it seem as if position updates jumped directly to image updates. Similarly, P4 referenced “*X position and mouseY*” and “*deltaX and deltaY*” for tracking mouse position but focused more on event handling than variable transformations. P6 discussed how the update function “*checks*

the position of the tile and displays the correct image,” yet they did not fully explain the role of variables like *centre* and *tile/name* in determining which images to load. While P9 traced some value transforms well, they only demonstrated partial understanding. They successfully described interactions through event handling and positional updates, even noting that *centre* represents a central tile. However, they failed to fully grasp how intermediary variables like *position* affect *centre*, which affects the tile naming, which affects the image loading. Even by the post-test, many participants continued to miss some deeper insights into intermediary variables and their flow.

4.6 SE Discussion - Practical Learning and Technical Contributions

Our evaluation of Scaffolded Exercises shows that novices gained a clearer grasp of how seemingly simple user interactions, like dragging the XKCD map, emerge from multiple layers of JavaScript variable transformations—explicitly tackling the **code complexity** obstacle (e.g., multiple, seemingly disconnected variable updates, such as ones about position and others about image updates, coordinating together). Compared to the debugger-based control, SE users achieved higher post-test scores and could articulate how “position,” “center,” and “tile” variables interplay to load new images as the user navigates. This underscores SE’s success in guiding learners beyond **surface-level** mouse event handling, through an *intuitive sensemaking order*, toward the “invisible” data flows often missed by novices.

SE builds on existing code-comprehension and debugging research—such as visualization tools for novices and professional source explorers—by automating an approach that begins at final DOM modifications and works backward through each causal step (cf. CDT, WebCrystal, Whyline, and others [23], [24], [26], [44], [48], [49]). Instrumentation injects small hooks into the target JavaScript to capture live variable values, then produces short, chunked code “exercises” in reverse call-stack order. Although the required setup is minimal (the user points SE to a reposi-

tory), our data shows this approach effectively steers novices through the code in a more logical progression than conventional line-by-line tracing.

By surfacing transformations “from effect back to cause,” SE helped learners connect each observable UI change to specific variable updates—an otherwise daunting task in large codebases [18], [19], [20], [39]. Participants consistently cited how moving from obvious DOM elements to deeper program variables prevented getting lost, enabling them to progress within their zone of proximal development, *link their prior knowledge* (e.g., knowledge of *position* influences understanding of *centre*), and discuss value transformations in detail [35]. These findings align with other efforts showing that focused, interactive scaffolds can improve sensemaking in code comprehension tasks [57], [58]. SE’s backward segmentation of the call stack is particularly valuable for revealing where and how each key variable evolves, so novices gradually build conceptual links interactively rather than fixating on single lines or events. As a result, learners not only identified individual snippets of code but developed a clearer, end-to-end understanding of the value transformations and data flow responsible for dynamic features, successfully *mapping this structure* to XKCD specifics (e.g., *centre* $\rightarrow$ *name* $\rightarrow$ *tile* $\rightarrow$ *image*).

In guiding users through layered transformations, SE illustrates the broader importance of structuring a “process” around code exploration, which enables data flow-related learning that might otherwise be more challenging. The idea of chunked, stepwise exercises—each anchoring to a known outcome—aligns with the PM framework’s emphasis on mapping concepts to example-specific instances, reducing cognitive load at each stage, incrementally revisiting prior knowledge, and supporting reflection. Our results suggest that carefully designed PM tools can help novices handle learning from code at scale by shaping how they traverse complex logic. SE thus adds empirical weight to the notion that tool-assisted process management fosters more thorough program comprehension, encouraging further integration of chunked exploration flows in automated tools

for software engineering professional development.

CHAPTER 5

KNOWLEDGE MAPS (KM), A PM TOOL FOR GRANULAR CSS TECHNIQUES

¹Professional front-end developers routinely develop production-grade UIs whose layouts must behave gracefully across a wide range of screens and settings. Accomplishing this work requires insight into detailed visual layout features and the ability to apply situation-appropriate and nuanced CSS techniques, including media-query-driven sub-properties, fractional units, overflow rules, and subtle grid/flex interactions. Tutorial resources may teach these techniques, but in isolation and with theoretical or canonical examples rather than real-world ones. Often, developers can only apply such techniques in practice once they have pieced together assorted tutorials, scattered blog posts, and years of professional source code inspection, the last of which is one of the best ways to learn CSS but a time-consuming and error-prone process.

To master CSS, a learner has to (a) decompose an unfamiliar page into its granular visual-layout features, (b) pinpoint the exact CSS properties and values responsible for each feature among dozens of interacting rules, and (c) reason about why specific implementations yield specific layouts (e.g. *grid-template-columns: 1fr 1fr 1fr 1fr* yields a rigid four-column layout while *grid: auto-fill* adapts responsively). In terms of learning outcomes, one should be able to recognize an assortment of layouts, such as (A) ones where equally sized content overflows into new columns dynamically (as in Flat Icons, Hero Icons in Figure 5.1), (B) ones following a fixed table-like structure with sidebars and assorted rectangular sections (as in Italic, Masterclass in Figure 5.2), (C)

¹This work was a collaboration with Roxy Wilcox, Fardeem Munir, Daniel Zhu, Salome Kariuki, Ava Robinson, Alexandra Andreiu, Mieraf Mulat, Lev Rosenberg, Nell O'Rourke, and Haoqi Zhang. Some initial writing may be borrowed from our prior collaborations. Source code contributions can be found at <https://github.com/gobid/km-css-techniques>.

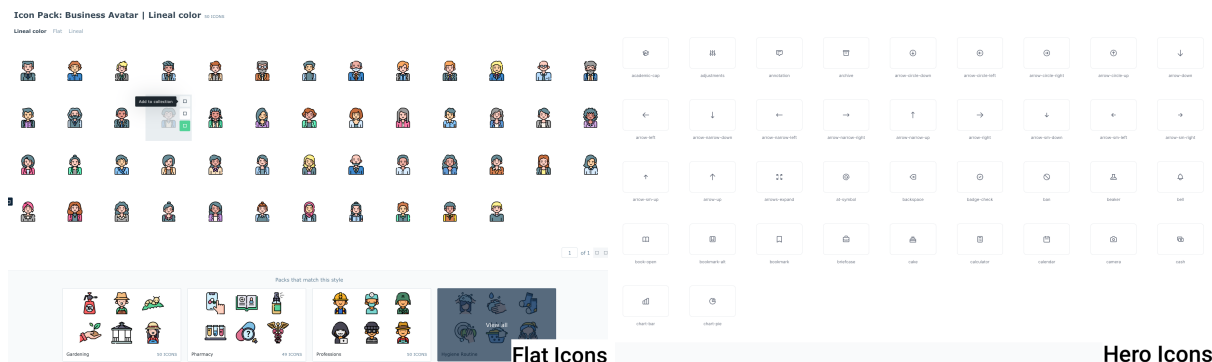


Figure 5.1: KM Examples: Learners should also be able to recognize examples with (A) layouts where equally sized content overflows into new columns dynamically (e.g., Flat Icons, Hero Icons).

ones with responsive layouts but without clear table-like structures (as in CSS Tricks, Smashing Magazine in Figure 5.3), and (D) ones with specific overflow behavior (as in CSS Tricks in Figure 5.3), as well as their underlying CSS techniques. We chose these learning outcomes and layout examples² because they cover a wide range of layouts and CSS techniques that are commonly used in professional web development.

Additionally, there may be nuances within each layout category that the learner needs to understand. Consider the scenario of a beginner replicating the Italic and Hero Icons galleries (Figure 5.1, 5.2). Both appear to use the same multi-column grid, so a learner who is looking to adapt their design may simply copy *display: grid* and press ahead. However, they miss that Hero Icons relies on *auto-fill* whereas Italic hard-codes four equal columns using *grid-template-columns*. So, their subsequent Hero Icons-inspired implementation looks fine on a small laptop but wraps incorrectly on mobile and larger screens, leaving the learner unsure why.

In early pilot sessions, we observed this confusion firsthand. Novices given several professional sites, such as Italic and Hero Icons, skimmed or tinkered arbitrarily, often analyzing one page at a time without deeper reasoning. They drew surface-level conclusions (e.g., “both use grid”), fixated

²Available on magenta-daffodil-7c0588.netlify.app.

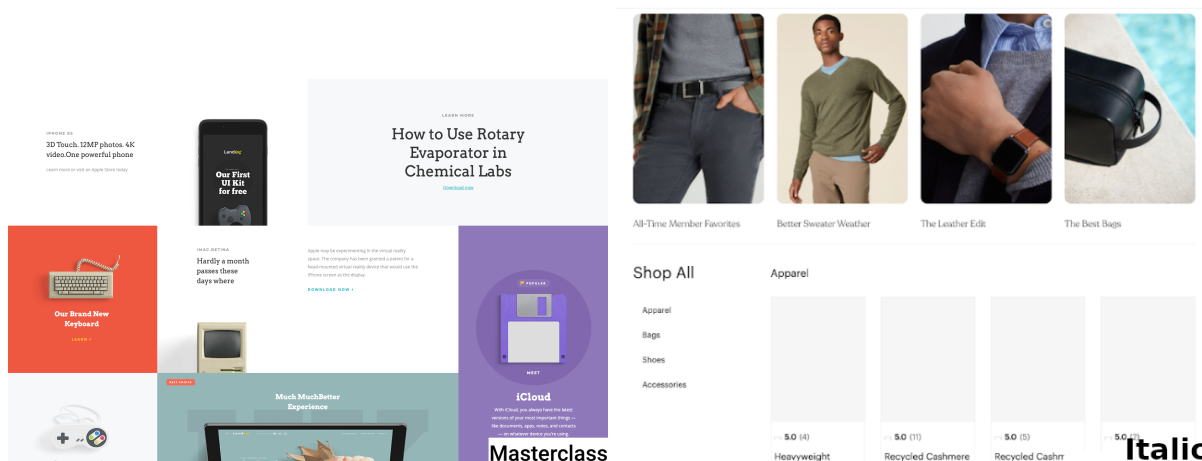


Figure 5.2: KM Examples: Learners should also be able to recognize examples with (B) layouts following a fixed table-like structure with sidebars and assorted rectangular sections (e.g., Italic, Masterclass).

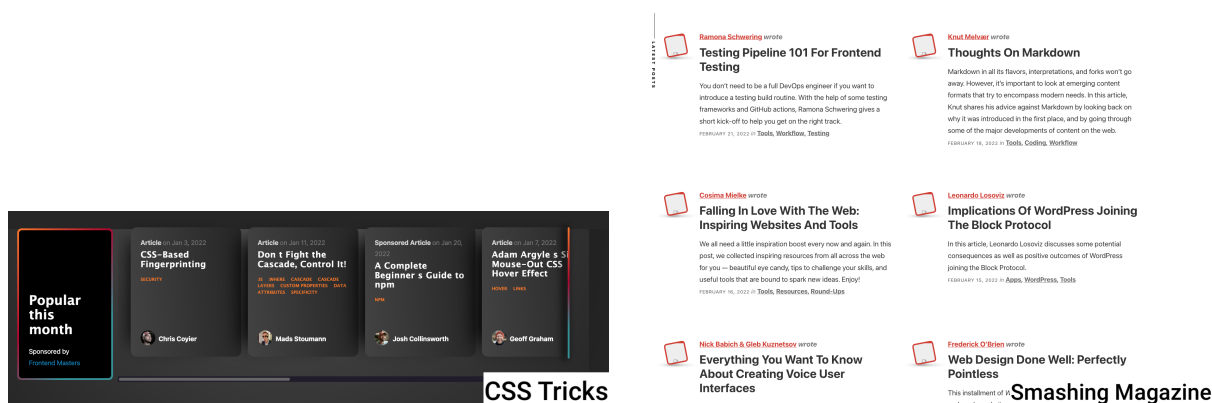


Figure 5.3: KM Examples: Learners should be able to recognize examples with (C) responsive layouts without clear table-like structures (e.g., CSS Tricks, Smashing Magazine); (D) layouts with specific overflow behavior (e.g., CSS Tricks).

on cosmetic details like color, and neglected complex nuances (e.g., layout changes under different viewport sizes). Sub-property distinctions such as *grid-template-columns* versus *grid: auto-fill* also slipped by. Lacking a systematic progression, learners are left with only partial, non-transferable insight into why each layout choice exists.

In this way, intermediate-level developers face a gap: they can reproduce tutorial snippets, yet cannot confidently adapt professional-grade layouts. Modern sites combine dozens of interacting CSS rules, and without guided comparison across multiple exemplars (recommended for priming learners about intuitive differences before exposing deeper implementations [29]), novices struggle to notice which lines of code actually matter for distinct layouts and may instead group sites superficially (e.g., by theme color or font type). This lack of deep, transferable, implementation-level understanding hinders their ability to diagnose layout bugs quickly, justify design choices to collaborators, and reuse expert patterns and concepts in new contexts.

Problem: As we have seen, learners **struggle with knowledge transfer** of professional CSS techniques because they view examples in isolation, lacking **awareness** of other examples, their nuances, and knowledge of alternative implementations against which to compare. Hence, they miss subtle but architecturally significant distinctions. For instance, the *grid-template-columns: 1fr 1fr 1fr 1fr* vs. *grid: auto-fill* distinction is not apparent unless learners compare multiple implementations, resize the page actively, and consider both **complex visuals and detailed CSS** together. Without structured mapping and guidance, they may group examples by superficial, **surface-level** details (e.g., rounded corners, syntax) rather than make conceptual tradeoffs. Moreover, without an organized process [25] of grouping, contrasting, and tinkering with the code of examples, learners may not achieve significant coverage of the granular visual outcomes and their underlying CSS, or link the two together in practical contexts.

We iteratively designed Knowledge Maps (KM), a tool to overcome these obstacles. KM has

learners explore CSS-heavy websites and map out their understanding into a “knowledge map” of layouts and their underlying CSS techniques. These layouts include overflowing columns, fixed table-like structures, sidebars, assorted rectangular divs, responsivity, etc. To investigate the obstacles, we adopted a design-based research approach. We have completed three iterations of KM, each pushing us toward a better understanding of the obstacles novices face when learning about granular visual outcomes and techniques. Through early designs, we discovered that despite features for interactive comparison and structured reflection, challenges around transfer, particularly of granular CSS sub-properties (e.g., *grid-template-columns: 1fr 1fr*, not just *grid*) persisted. This would push us to think more about designing a structured process for learning from diverse examples and heeding details.

5.1 How KM (Iteration 3) Enacts PM: Cycles of CSS Similarities and Differences

KM’s early iterations helped learners grasp general CSS techniques, their rationale, and visual outcomes. However, these iterations did not guide learners toward understanding granular differences between examples. For instance, while learners recognized that CSS grids can create table-like layouts, they struggled with more nuanced concepts like fractional units and *repeat autofill*. Such subtle differences are crucial in the fields of CSS and UI Design because even small CSS properties can significantly influence a website’s layout and user experience [131], [132]. Throughout our research, learners faced three main obstacles: code and example complexity (e.g., witnessing five examples each with its own collection of CSS subproperties at once), surface-level focus (e.g., focusing on color, styling, and font), and lack of transfer (e.g., example-specific insights around grid producing table-like layouts without notice of how media queries shape nuanced behavior at different screen sizes). To address these high-level obstacles, we reinvent the system through process management via prompts, activities, and an integrated map-process complex, in which

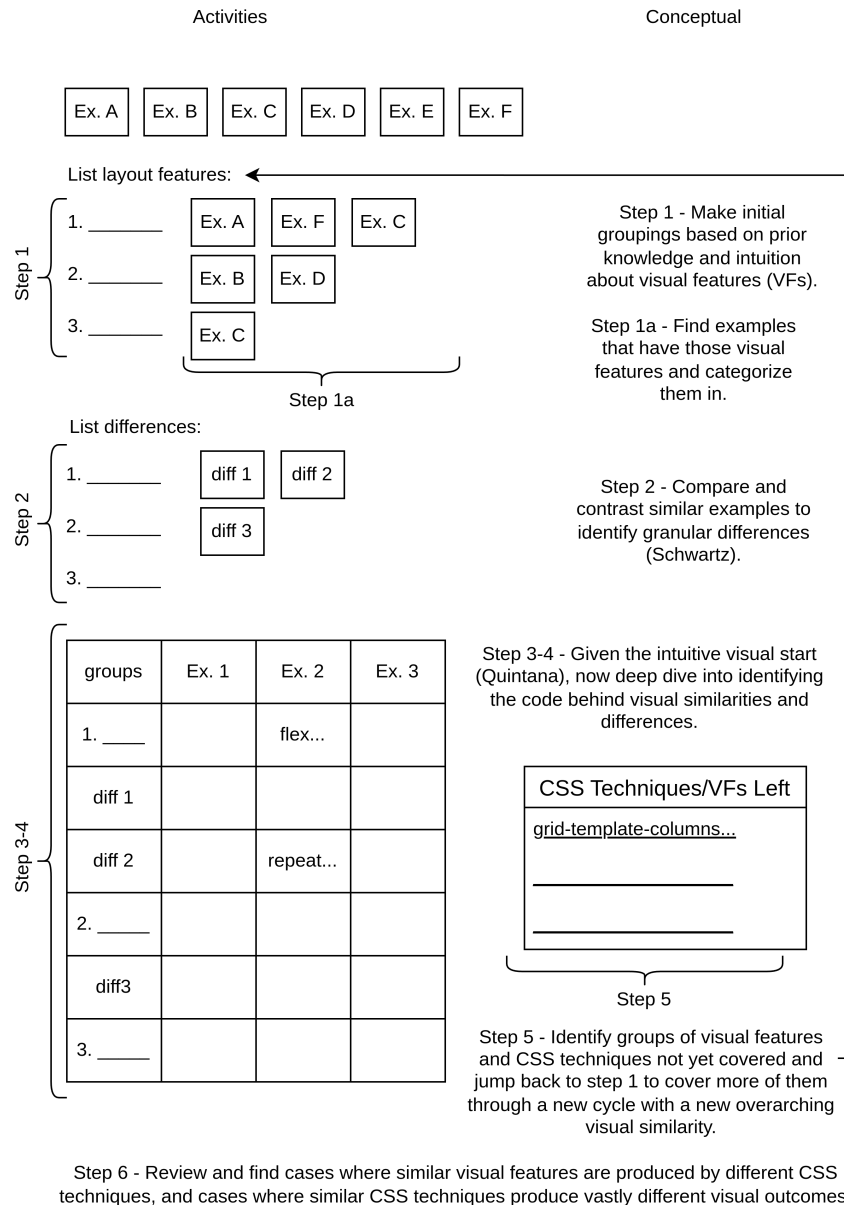


Figure 5.4: Here we present how KM helps users progressively build (PM2) a knowledge map of CSS concepts. Learners begin with the intuitive exercise of writing down a common overarching visual feature (PM1) in a subset of professional examples. They then categorize which examples have that common feature (step 1). They compare and contrast (PM6) examples to identify granular differences within each overarching similarity (step 2). Finally, learners dive deeper by looking at the code through the comparison tool, mapping (PM7) specific syntaxes with specific visual outcomes they have listed (steps 3-4). They answer reflection questions (PM8) [36] on how and why the syntaxes lead to specific visual outcomes. They then return to Step 1 and repeat these feature mapping cycles until they cover as much CSS layout code as possible (Step 5). Finally, they attempt to find instances of similar syntax leading to different visual outcomes and different syntax leading to similar visual outcomes (step 6). They also check to understand any lingering CSS code they did not cover.

learners iteratively group and contrast examples in order to better understand critical CSS details (Figure 5.4). We argue that KM can now enable learners to transfer nuanced CSS concepts from training examples to new applications.

System Overview:

KM (Figure 5.4) guides the learner through repeated *cycles* of exploring examples of professional CSS webpages, their visual features, and underlying CSS. The learner first clusters diverse examples by an intuitive visual feature (e.g., “multi-column grid”) to encode their prior knowledge, and next compares differences within each cluster (e.g., spacing, wrapping nuances). They then inspect CSS that explains the shared feature and the observed differences. Finally, the learner repeats this cycle multiple times for new visual features and solidifies insights by reflecting on cases of similar visuals stemming from different code, or vice versa, building ever-richer and increasingly comprehensive mappings between visuals and code.

This structure ensures learners proceed from intuitive visual groupings to increasingly sophisticated CSS understanding, aided at each stage by interface features that guide attention, comparison, and reflection. We now describe each phase in more detail, while explaining how KM enacts process management and prepares learners for transfer.

KM (Figure 5.4) consists of stages of grouping, comparing, and mapping using diverse professional examples, in order to help learners develop transferable conceptual knowledge.

1. **Group visually similar sites.** Learners cluster examples by an intuitive visual layout features based on end-user observation before reading code (e.g., *Flat Icons*, *Hero Icons*, *Italic*, *Masterclass*, and *Smashing Magazine* from Figures 5.1, 5.2, 5.3 all share a 2D grid-like interface).
2. **Compare within the group.** Learners resize and scroll to surface nuanced visual differences (e.g., *Masterclass* has 0 gap; items in *Flat Icons* and *Hero Icons* overflow while *Italic* and

Smashing Magazine cap at 4 and 2 columns, respectively). Such comparison prevents siloing, but operating at the end-user level prevents premature code complexity overload.

3. **Map the shared feature.** Inspecting code side-by-side, learners identify the high-level technique(s) that implement the shared feature (e.g., all five sites rely on *grid* or *flex*).
4. **Map the differences.** Learners connect observed differences to specific sub-techniques and property–value choices (e.g., *grid-template-columns: 1fr 1fr 1fr 1fr* vs. *grid: auto-fill* explains how *Italic* has a maximum of four columns while *Flat Icons/Hero Icons* stretch and wrap incrementally; *grid-gap: 0px* shapes *Masterclass*’s density; larger gaps govern spacing in the others). Such mapping allows learners to build on their prior insights around nuanced visual outcomes and thereby deepens their understanding of both visuals and code.
5. **Cycle to a new overarching feature.** Learners repeat the observe → compare → map routine for additional features and examples (e.g., a “non-traditional grid layouts” group with *CSS Tricks* for overflow scrolling and *Masterclass* for stitching together *divs* of assorted sizes).
6. **Reflect and consolidate.** Learners synthesize across cycles (e.g., similar four-column visuals can arise from fixed counts or auto-fill depending on the screen size; two sites using *grid* can still yield vastly different outcomes due to subproperty distinctions such as *grid-gap*). These reflections are logged in the knowledge map and signal when to end (i.e., coverage has been reached) or restart a new cycle. This step also aligns with the best practices around knowledge organization and synthesis that prepare learners for transfer.

This example-driven progression shows what a learner produces after each stage: an evolving set of visually grounded clusters, explicit CSS annotations for both commonalities and differences, and a growing collection of transfer-ready insights.

PM Solution:

Like SE, Knowledge Maps (KM) is a *process management* tool, with a cluster, compare, and map progression that is structured around 3 core sensemaking approaches:

1. **Begin the Sensemaking Order with Intuitive Visual Clustering:** KM prompts learners to group examples by **visually intuitive features** (step 1) before viewing code. This reduces complexity and serves as a natural entry point, ensuring novices begin the process at a comfortable place instead of diving directly into dozens of CSS files. Once they have this intuitive anchor, they can proceed to build on and deepen their understanding.
2. **Link Prior and Current Insights:** As learners create and refine these clusters, KM systematically resurfaces previous observations (e.g., previously noted similarities when identifying differences in step 2, or previously noted visual features when identifying their underlying CSS techniques in steps 3-4), encouraging them to integrate new findings with **existing knowledge**. This prevents disjointed, one-off, or surface-level analyses (i.e., seeing some complex syntax too soon without understanding it, and missing others) and builds a cumulative mental model (as in steps 5-6), i.e., approaching complex syntax after understanding the nuanced visual differences between examples.
3. **Compare and Connect Conceptual Insights to Code Differences:** Once users clearly see visual groupings (step 1) and sub-group distinctions (step 2), KM directs them to the underlying CSS that explains those differences (steps 3-4). Moreover, by contrasting code snippets side by side, learners are primed to notice differences and learn about their example-specific code (e.g., *grid-template-columns: 1fr 1fr 1fr 1fr* vs. *grid: auto-fill*). This comparing and **mapping** process fosters deeper understanding of professional layout techniques and reveals the rationale behind each variation, which in turn can facilitate transfer.

Collectively, these steps highlight how KM addresses the same core sensemaking barriers

found in other domains (e.g., SE with JS data flow), but tailored here to granular CSS differences.

5.1.0 *How KM's Interface Implements KM's Process Management Steps*

The following content details how KM's interface (Figure 5.5) implements each approach in practice through the lens of PM. The interface serves learners investigating CSS layout techniques.

Comparison: The (a) steps of the KM process and tool center around comparison and (b) reflection. Learners use KM's (c) side-by-side comparison tool to (d) resize and (e) analyze website pairs in order to group and **compare** (PM6) them, at first without even looking at their code. This can help them identify subtle differences between sites with similar visual features [29], better priming them to learn concepts, i.e., code in this case. This approach builds on their visual intuitions [26]. For example, when comparing the layouts of Flat Icons and Hero Icons (Figure 5.1), users may initially assume they are the same but later discover intricate differences, such as how Flat Icons adjusts both icon widths and heights while Hero Icons modifies only widths. Upon resizing the window, they may also notice differences in the minimum number of displayed columns. This visual-first approach gives learners a practical starting point and can help them avoid surface-level thinking (e.g., grouping sites by superficial similarities or shared syntax). In fact, by grouping visual outcomes across several examples, learners implicitly predict which may share similar code, but through active comparison/contrast and (later) mapping, they may improve the transfer of their understanding of said visuals and underlying CSS [25], [29].

Mapping: The KM interface (f) **highlights salient property and property/value pairs** (PM3) shared by the two examples under comparison. The CSS displays enable users to (g) interactively toggle CSS properties on and off, (h) experiment with different values (tinkering), and observe the resulting changes in the layout in real time. The (i) side-by-side HTML snippets and (j) hover-over syntax definitions intend to reduce **extraneous cognitive load** (PM4) during mapping. These

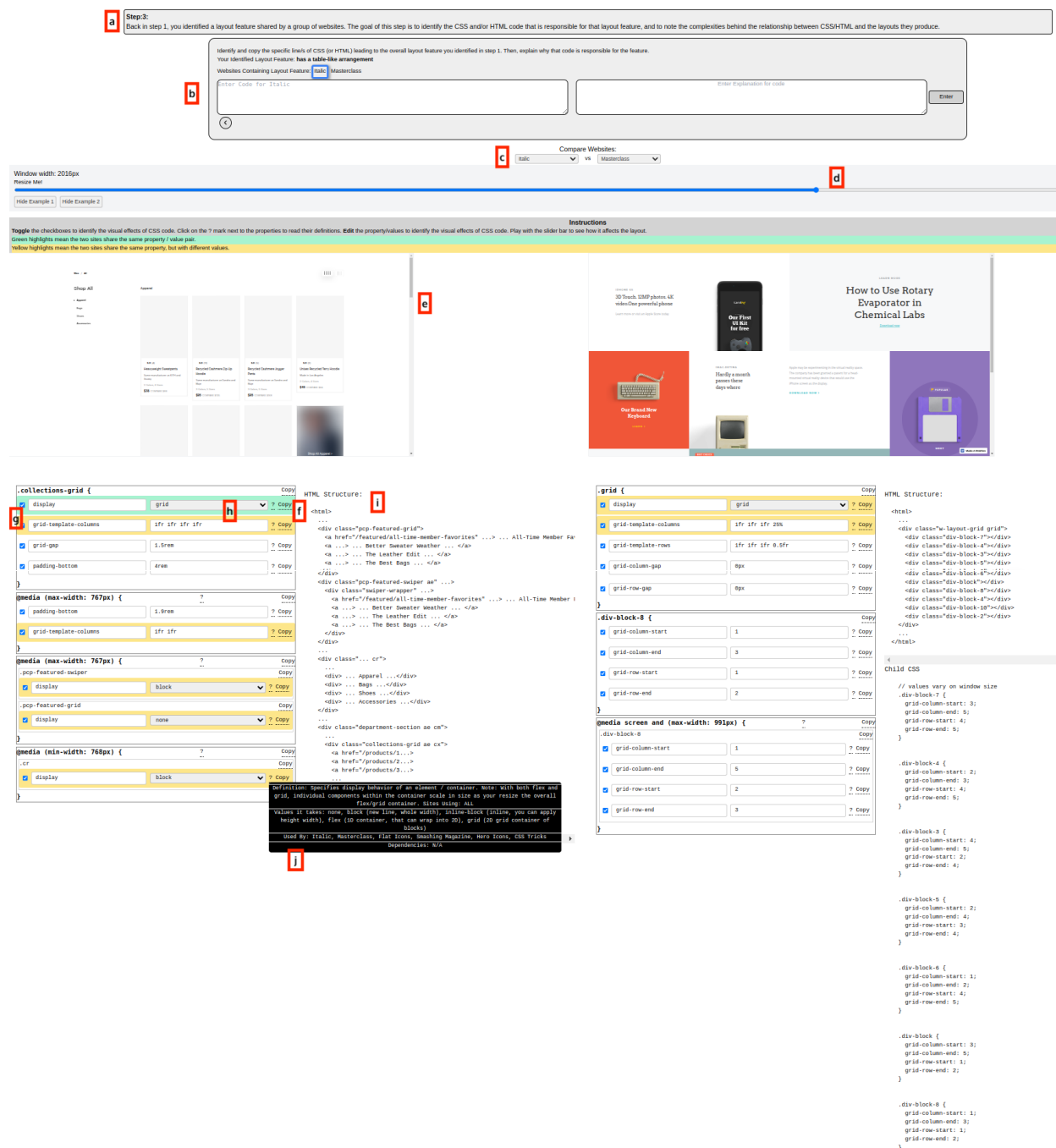


Figure 5.5: KM walks users through (a) a sequence of steps (step 3 shown here as an example), each with (b) reflection inputs shown at the top of the webpage. Users can also select (c) a pair of websites to compare. Below the inputs, KM provides users the ability to (d) interactively re-size examples' widths (to assess responsiveness) and (e) compare example pairs side-by-side. (f) Highlights also draw user attention to sites that share properties and property/value pairs. They can also (g) toggle (turn off and on) and (h) tinker with (change) the values of CSS properties. (i) Side-by-side HTML snippets and (j) hover-over definitions provide further context. These features help learners map granular visual outcomes to specific CSS techniques.

features work together to make the process of **mapping** (PM7) granular visual outcomes to specific code both intuitive and concrete, supporting deeper conceptual understanding.

Cross-Cutting Affordances: PM affordances that further drive learning include knowledge maps (Figure 5.11) for organizing insights from comparison and mapping, capturing granular differences, and prompting learners for further detail (e.g., What CSS or visual features are missing from the current map that we can cover in a next cycle?). They also include reflection prompts (Figure 5.5 - b) during comparison that remind learners to **interactively resize the page** (PM5) and discover easily overlooked visual and CSS differences.

We argue that chunking the process [26], by starting with intuitive visual comparisons to tease out differences [29], and prompting learners to tinker, reflect, and map code and rationales [36], enhances learning by providing a structured pathway for understanding CSS details. By actively modifying CSS properties—in line with Quintana’s “multiple views” and “malleable representations” [26]—learners observe how small code adjustments impact layout outcomes.

Together, these repeated tool use interactions aim to scaffold expert-like processes, integrating exploration and map construction into one continuous workflow. Figures 5.6 - 5.12 show how KM creates a seamless interplay between noticing, testing, and formalizing CSS insights.

Now that we have covered the arguments for KM’s core features, we proceed to detail its interfaces for process step-by-step.

Grouping Visually Similar Websites: To begin this process, learners compare pairs of websites, focusing only on page visuals without looking at the code (Figure 5.6). This aims to reduce complexity and their **extraneous cognitive load** (PM4). They identify one **intuitive visual layout feature** (PM1) common to multiple websites (a group), such as centered tables or responsive layouts. Once they identify it, they share why they picked it (i.e., how exploring it will advance code comprehension and coverage). Figure 5.6 shows the interface for Step 1, where the learner

Step 1:
The goal of this step is to identify a common layout feature between 2-5 example websites. First explore the layouts of the different websites. Then, identify and describe a specific shared feature.

List one layout feature shared by any 1-5 websites, and mark the corresponding checkboxes.
Your Identified Layout Feature: **they have a table like layout**

they have a table like layout

☒ Italic
☒ Masterclass
☒ Flat Icons
☒ Smashing Magazine
☒ Hero Icons
☐ CSS Tricks

Enter

Compare Websites:

Italic

vs

Masterclass

Window width: 1471px
Resize Me!

Hide Example 1

Hide Example 2

Shop All

Apparel

3D Touch, 12MP photos, 4K video. One powerful phone.

Learn more or visit on Apple Store today

Our First UI Kit for free

How to Use Rotary Evaporator in Chemical Labs

Learn More

Figure 5.6: Step 1 - Identify Common Visual Layout Features (Sample Data)

is prompted to explore several example websites and identify a shared visual layout feature. The figure highlights how the learner records their observations after comparing the visual previews of featured sites. The goal at this stage is to encourage broad, intuitive observation of layout patterns before any code is revealed, setting the foundation for deeper analysis.

Comparing within the Group: They then progress beyond the surface-level, **comparing and contrasting** (PM6) examples within that previously noted visual layout group (Figure 5.7) to build awareness of and tease out as many subtle differences and details as possible (e.g., slight variations in spacing, overflow behavior, or alignment). These differentiated groupings (Figure 5.8) serve as **prior knowledge** (PM2) for future steps and form the outline of their knowledge map. Figure 5.7 depicts Step 2, where the learner, having grouped sites by a common feature, is asked to list differences in how each site implements that feature. The figure shows the annotation area and the comparison interface, which guide the learner to record subtle distinctions. This step is crucial for moving from surface-level grouping to a more nuanced, comparative understanding. Figure 5.8 shows how the learner can collect all the differences within a visual layout group, with each site's

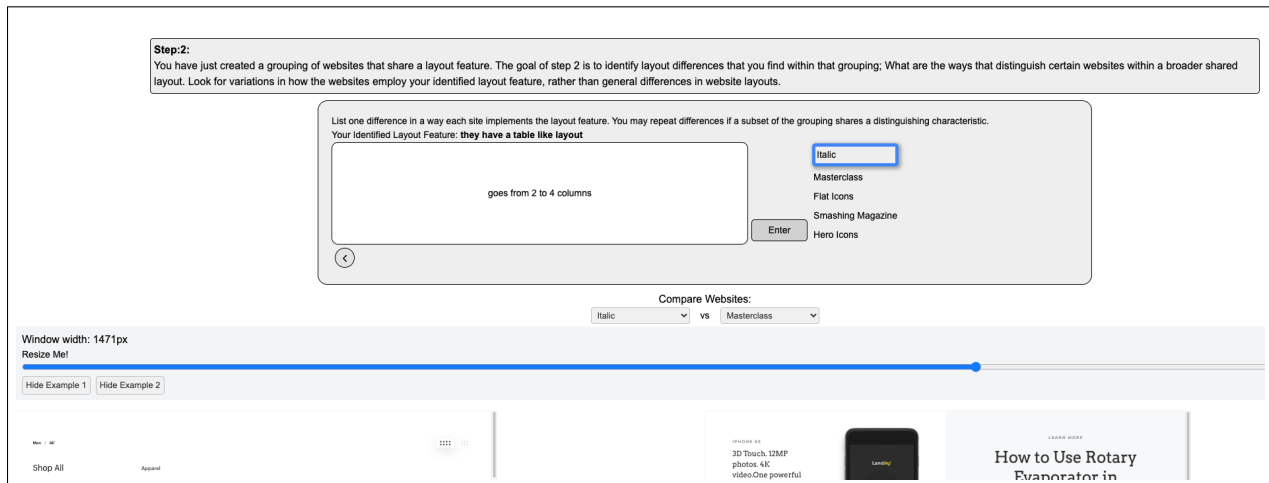


Figure 5.7: Step 2 - Find Differences within Group (Sample Data)

unique implementation highlighted. The learner's annotations accumulate and together help them form a detailed map of variation across examples.

Mapping the Shared Features to Code: Attuned and primed to learn from code, learners are asked to identify the code each example uses to implement the common visual layout feature by examining property/value pairs (such as display: grid) across examples and **mapping** (PM7) them to the shared feature (Figure 5.9). They can refine their understanding using **interactive** (PM5) features such as property toggling and value tinkering in the code view (Figure 5.5). At this stage, KM users may also speculate why said CSS properties were chosen. This step bridges the gap between visual intuition and technical understanding.

Mapping the Differentiating Features to Code: Figure 5.10 presents Step 4, focusing on the code behind granular differences. Here, the learner is asked to map the subtle distinctions identified in Step 2 to their underlying CSS, and to articulate why those code choices produce the observed differences. Again, the interface's annotation and mapping affordances help the learner make these connections explicit.

Cycling and Reflecting: At the end of these cycles of steps 1-4 (grouping → comparing →

Step 2:
You have just created a grouping of websites that share a layout feature. The goal of step 2 is to identify layout differences that you find within that grouping; What are the ways that distinguish certain websites within a broader shared layout. Look for variations in how the websites employ your identified layout feature, rather than general differences in website layouts.

List one difference in a way each site implements the layout feature. You may repeat differences if a subset of the grouping shares a distinguishing characteristic.
Your Identified Layout Feature: **they have a table like layout**

starts at 1 col and goes to inf - height of icons doesn't change but width stretches

Enter

Italic	goes from 2 to 4 columns
Masterclass	blocks of assorted sizes in tight table like grid
Flat Icons	goes from 3 to inf columns
Smashing Magazine	2 column news layout down to 1 on mobile
Hero Icons	starts at 1 col and goes to inf - height of icons doesn't change but width stretches

Compare Websites:
Italic vs Masterclass

Window width: 1471px
Resize Me!

Hide Example 1 Hide Example 2

Figure 5.8: Step 2 - Collect all Differences within Group (Sample Data)

Step 3:
Back in step 1, you identified a layout feature shared by a group of websites. The goal of this step is to identify the CSS and/or HTML code that is responsible for that layout feature, and to note the complexities behind the relationship between CSS/HTML and the layouts they produce.

Identify and copy the specific line(s) of CSS (or HTML) leading to the overall layout feature you identified in step 1. Then, explain why that code is responsible for the feature.
Your Identified Layout Feature: **they have a table like layout**

Websites Containing Layout Feature: Italic Masterclass Flat Icons Smashing Magazine Hero Icons

display:grid

this creates the grid for the table like layout

Enter

Compare Websites:
Italic vs Masterclass

Window width: 1471px
Resize Me!

Hide Example 1 Hide Example 2

Instructions

Toggle the checkboxes to identify the visual effects of CSS code. Click on the ? mark next to the properties to read their definitions. Edit the property/values to identify the visual effects of CSS code. Play with the slider bar to see how it affects the layout.

Green highlights mean the two sites share the same property / value pair.

Yellow highlights mean the two sites share the same property, but with different values.

Figure 5.9: Step 3 - Find Code behind Common Visual Layout Features, and the Rationale behind that Choice of Code (Sample Data)

Step 4:
Back in step 2, you identified differences between the group of websites that shared your identified layout feature. The goal of this step is to identify the CSS and/or HTML code that is responsible for those differences in layout, and to note the complexities behind the relationship between CSS/HTML and the layouts they produce

Identify and copy the specific line/s of CSS (or HTML) leading to each layout difference you identified in step 2. Then, explain why that code is responsible for that difference.
Your Identified Layout Feature: **they have a table like layout**
Your Identified Layout Difference for Italic: **goes from 2 to 4 columns**
Websites Containing Layout Feature: **Italic** Masterclass Flat Icons Smashing Magazine Hero Icons

At lower **widths** we shift down to two columns with the 1fr 1fr but at high **widths** we have 4 columns with the 1fr 1fr 1fr 1fr

Compare Websites:
Italic vs Masterclass

Window width: 1471px
Resize Me!

Hide Example 1 Hide Example 2

Instructions
Toggle the checkboxes to identify the visual effects of CSS code. Click on the ? mark next to the properties to read their definitions. Edit the property/values to identify the visual effects of CSS code. Play with the slider bar to see how it affects the layout.
Green highlights mean the two sites share the same property / value pair.
Yellow highlights mean the two sites share the same property, but with different values.

Shop All

3D Think 12MP photos. 4K videos One powerful phone

How to Use Rotary Evaporator in

Figure 5.10: Step 4 - Find Code behind Granular Differences, and the Rationale behind that Choice of Code (Sample Data)

Step 5:
Congrats! You just finished a cycle of the RM process! Below you can see the knowledge mapping you created through out this process. Click finish to answer some final reflection questions. Or, you can also continue your learning by starting another cycle by clicking the "Start Another Cycle" button below.

Layout Feature	Websites	Code driving layout feature	How these CSS techniques drive your identified layout feature	Differences	Code driving difference	How do these CSS techniques drive your identified difference
grid-like layout	Hero	display: grid;	The display: grid is used to create a grid layout that allows the user to view content in an organized manner of columns and rows. Grid also allows the site to place items in a similar manner.	When resized from medium to maximum, Italic only shows 4 columns and shows two columns on the smaller screens.	grid-template-columns: 1fr 1fr 1fr 1fr; @media (max-width: 767px){ grid-template-columns: 1fr 1fr; }	The grid-template-columns directly define four fractional units and 2 fractional units when the screen is 767px or less hence tables only show 4 columns on bigger screens and 2 columns on smaller ones.
Masterclass	display: grid;	Grid allows the user to place items in a precise order which makes it easier to develop.	There is no use of spaces between the different columns / cards.	grid-template-columns: 1fr 1fr 1fr 1fr; @media (max-width: 767px){ grid-template-columns: 1fr 1fr; }	The grid-template-columns and the grid-row-gap are both set at 0px ensuring no spaces are added.	
Flat Icons	flex;	Since there is no code provided for the different spacing, flex has an initial space distribution. I believe Flat Icons goes down to 2 columns on small screens and can have up to 11 columns on high screens.	There is a lot of use of space between the different cards. When resized to smaller screens, it displays only one column and only two on medium and larger screens.	flex-wrap: wrap; @media (max-width: 480px){ flex-wrap: wrap; @media (max-width: 480px){ flex-wrap: wrap; }	Flat icons use flex-wrap which makes the content responsive according to the screen size. The icons on the small screens (480px and below) have a minimum width of 50% ensuring that only 1 would be displayed.	
Smashing Magazine	display: flex;	flex has an initial space distribution. I believe that display flex is being used. flex wrap is being used to there is a lot of use of space between the different cards. When resized to smaller screens, it displays only one column and only two on medium and larger screens.	It shows one column on smaller screens with great use of space between the different cards.	@media screen and (min-width: 480px){ flex-wrap: wrap; @media screen and (min-width: 480px){ flex-wrap: wrap; }	Smashing Magazine displays only one column on smaller screens because the flex basis for the col-md-12 is set to 100% when the screen size is 480px or smaller.	
Hero Icons	display: grid;	Grid allows us to have more control over the precise placement of the different items.		grid-template-columns: repeat(auto-fill, minmax(120px, 1fr));	The gap being used is significantly higher than the grid-template-columns is what is being used to create the one column simply because it only uses 1 fractional unit with a minimal size width of 120px, meaning that as long as there is space, the columns will be auto-filled or multiplied ones. They will show only one column.	

Layout Feature: **Appearing and Disappearing Website Hero**

Code driving layout feature: @media (min-width: 768px){ .col-display: block; } } The .col is the class of the sidebar and when the screen is 768px or lower, the sidebar display is set to block. The sidebar is hidden when the screen size is reduced.

How these CSS techniques drive your identified layout feature: Some of the new items are hidden when the screen is reduced to 768px or lower.

Differences: @media (min-width: 768px){ .col-display: block; } } The sidebar (.col) disappears on the smaller screen 768px or lower when it is set to block.

Code driving difference: @media (min-width: 768px){ .col-display: block; } } The overflow, when set to scroll will enable the user to be able to view the hidden items as they scroll.

How do these CSS techniques drive your identified difference: The overflow, when set to scroll will enable the user to be able to view the hidden items as they scroll.

Compare Websites:
Hero vs CSS Tricks

Window width: 768px

Figure 5.11: Step 5 - Review Notes from Cycles of Finding an Overarching Visual Feature → Finding its Granular Sub-Features → Mapping its Code → Repeating the Process (P4 Data)

Can you find cases where you have the same visual feature in 2 examples that are implemented in different CSS techniques/properties? Add your findings in the left input box below.

Hero Icons and Flat Icons both have a clean, symmetric table-like structure with auto-growing rows and columns. However, the first uses grid and the second uses flex.

Masterclass and Hero Icons both use grid but the first has table elements of assorted sizes and the second has table elements of uniform size and is more symmetric.

Figure 5.12: Step 6 - Find cases where different CSS techniques produce similar visual features and cases where similar CSS techniques produce vastly different visual outcomes. (Sample Data)

mapping similarities → mapping differences → repeat), learners **reflect** (PM8) on an evolving “knowledge map,” consolidating insights from their comparisons and code mapping (Figure 5.11). During this reflection while looking at their knowledge map, they analyze patterns such as how similar code can lead to different visual features or vice versa (Figure 5.12). They also identify any remaining CSS properties missed earlier and map them to visual features. This process-managed cycle aims to promote conceptual learning and transfer to real-world scenarios through metacognitive reflection. Learners who complete sufficient cycles of KM across diverse websites should understand a breadth of visual features and existing techniques, both broad and subtle, and why professionals use them in specific contexts.

5.2 KM System Implementation

KM is implemented as a React-based web application with a Node.js backend that enables interactive CSS learning through embedded website comparison. The system uses iframes to embed professional websites side-by-side, allowing learners to resize, compare, and manipulate CSS webpages and their properties in real-time. The frontend provides a structured interface for the six-step learning process described earlier, and our system handles dynamic updates of CSS properties from input fields onto the embedded sites. This architecture supports the core PM affordances of visual grouping, comparison, mapping, and reflection through interactive features like property toggling, value tinkering, and responsive resizing.

The KM system implementation is simpler than SE and RM. Aside from the hard-coded placement of wizarded HTML snippets, the KM system is automatically customizable: it can embed new websites, leverage results from Ply [13], a tool that identifies influential CSS on the page, to obtain³ relevant, visually impactful CSS property-value pairs, and automatically update the page

³Note, we did not need to rerun Ply since that was already done in iteration 2 for each of these sites. Our prototype

based on user tinkering, toggling, etc. Under the hood, KM structures a user’s CSS inputs and leverages JavaScript that uses the browser’s *postMessage API* to update the websites’ visuals.

5.3 Controlled Think-Aloud Evaluation of KM to Gauge Learning

To evaluate KM and its instantiation of PM, we followed the same 5v5 setup as SE (see Chapter 4’s System Evaluation section for more details), recruiting, screening, spot-checking (no red flags), and assigning conditions similarly. The nature of the study, timing, breaks, learning orientation, scoring, interventions, and phases of pre-test → tool use → post-test (same as pre) → survey were also identical. The studies measured the effectiveness of the PM affordances, such as KM’s example-contrasting exercises, in helping learners understand and transfer granular CSS knowledge. We note some system specifics:

Screening: We needed to test on intermediate-level CSS developers with knowledge of broader CSS properties (e.g., *grid/flex* though perhaps not how to apply *grid-template-columns* in new settings). Hence, we ensured KM participants (both conditions) scored above 60% on the screener’s CSS quiz. Screener questions were multiple choice. They had learners pick the CSS property or code responsible for a particular described behavior (e.g., *z-index* helps set the stacking order of elements, *display:flex*; *align-items:center* on a *div* lets you align its children to its center).

Definitions Provided: To further prepare KM participants (both conditions) for the interactions, we defined “visual layout features” and “CSS techniques” for them after the spot check. A visual layout feature is a feature pertaining to the layout details of a website, such as the arrangement of child elements, responsiveness, spacing, overflowing/scrolling, transformations, translations, animations, transitions, sizing, appearing/disappearing, rotation, etc. It excludes styling of

from iteration 2 did not have all the necessary layout syntaxes because Ply may require certain fixes and updates. Instead of side-tracking to modify Ply, we “wizarded” the selection of these syntaxes that have visual impact, as an automated solution should be possible (i.e., an updated Ply).

text, color, etc. By CSS technique, we mean the actual code, or in other words, the properties and values they can take on, i.e., using the CSS grid technique or the CSS flex technique.

Pre/Post-test: The KM pre/post-test asked participants to describe the CSS techniques they thought were responsible for the layouts of three example websites that use grid, media queries, spacing, etc. (Figure 5.13). These three websites differed from those used during the tool-use phase to prevent bias. For each site, it asked users to (a) *describe the site’s visual layout features*, (b) *predict their underlying CSS techniques*, and (c) *explain why those CSS techniques were chosen for the example at hand*. We prompted participants for more details on the visual layout, encouraging them to resize the page and cover all sections of the web pages. Participants were instructed to focus on visual layout features rather than aesthetics like color or typography.

Category	Handsome Frank	Alphabet Icons	Rocksteady
Media Queries	~4	~13	~2
Grid and Sub-techniques	~6	~12	~3
Flex and Sub-techniques	~4	~8	~1
Transitions	~3	0	0
Scrolling/Overflow	~4	~3	0
Spacing	~3	~14	0
Total per Example	24	50	6
Grand Total: 80			

Table 5.1: Point Allocation by Category of the Learning Outcomes Rubric for Scoring Visual Layout Features and Underlying CSS Across Sites (much improved from previous KM iterations): Note we use ~ because some rubric items (codes) awarded a point for mentioning one of the several possible techniques (e.g., *flex* or *grid*) behind visual features (e.g., a table has a minimum of x columns), or some codes could go in multiple categories. Each rubric point was assigned to one of the possible categories it belonged to in the table above. Examples of techniques include *grid*, and examples of sub-techniques include *grid-template-columns: 1fr 1fr 1fr 1fr*, which produces a four-column layout, and *grid-template-columns: 1fr 1fr*, which produces two columns and is often used inside media queries for small widths. Our rubric contained many granular specifics to check if participants were noticing, conceptually understanding, and applying as many details as possible.

Control Details and Training: We aimed to isolate the effects of PM while still providing the

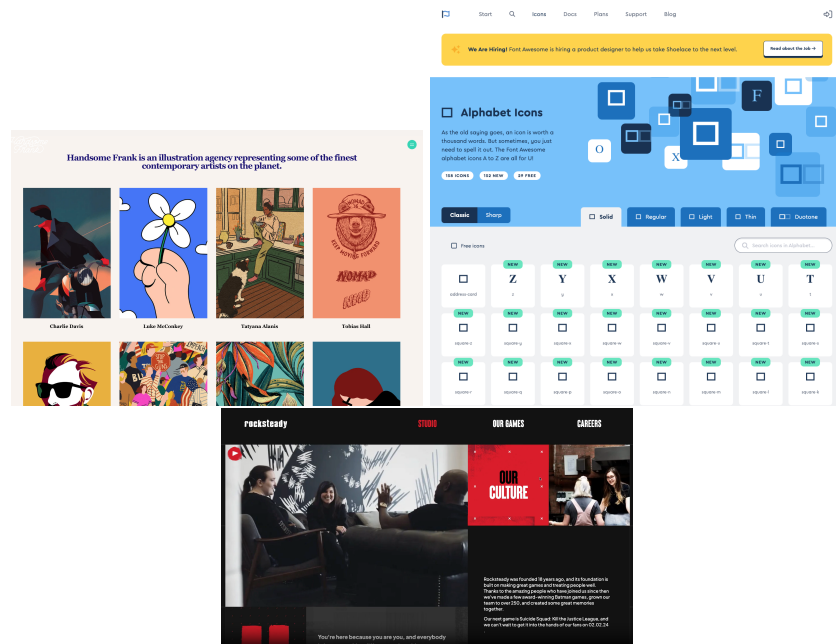


Figure 5.13: Three websites rich in granular CSS techniques used in the pre and post-tests. Handsome Frank (handsomefrank.com) is rich in grid techniques, media queries, transitions, and scrolling nuances. Alphabet Icons (fontawesome.com) adds stretchable flex implementations to the equation. Rocksteady (rocksteadyltd.com) has tight spacing and custom grid arrangements. Together, these sites are sufficiently different from the examples used in tool use, so we could use them to test for transfer while still covering all the techniques taught during tool use.

1. **display** specifies display behavior of an element/container. → Values can be **grid** (2D grid container of blocks), **flex** (1D container that can wrap into 2D), **block** (new line, whole width), **inline-block** (inline, you can apply height/width). *Src: W3Schools / The Tech Platform. Sites Using: ALL.*
2. **grid-template-columns** specifies the number (and the widths) of columns in a grid layout. The values are a space-separated list, where each value specifies the size of the respective column. → Values can specify **pixels**, **percentages**, **fractions** (i.e. 1fr), or **auto-fill**. With auto-fill, by default you go down to 1 column at the smallest widths. *Src: W3 Schools / The Tech Platform. Sites Using: Italic, Masterclass, Hero Icons. Implicit Dependencies: must use display:grid.*

Figure 5.16: A sample of the glossary terms available to both KM conditions (delivered via Tutorons for the experimental condition).

control a fair state-of-the-art learning experience. KM's control group used CDT and online CSS resources to explore the same six CSS-heavy websites that the experimental group did. Immediately before tool use, they were provided an interactive CDT tutorial⁴ to familiarize themselves with CDT features, such as inspecting elements and manipulating CSS properties. During CDT tool use, they filled out a spreadsheet (Figure 5.14) detailing their insights on the code behind visual features they could spot. They had access to the same websites (via links, not iframes) and the same CSS/HTML snippets available to KM users (Figure 5.15) including a glossary of syntax definitions (Figure 5.16). This setup ensured that control users focused on learning CSS techniques rather than finding relevant code, which is handled by existing tools [13]. However, the control did not experience KM's interactive features, such as code highlighting, tinkering, reflection prompts, and structured comparisons. For instance, they could only view one website at a time, rather than two side-by-side.

Experimental Tool Training: KM users watched a video demo of KM use immediately before using the tool.

⁴<https://developer.chrome.com/docs/devtools/css/>

Scoring and IRR: To evaluate whether participants⁵ could understand granular CSS techniques, we scored the tests according to a rubric of learning outcomes, or codes (e.g., sub-techniques of grid, flex, media query, spacing, etc.), described in Table 5.1, which, inspired by Ply and Iso-pleth, collates criteria for success [13], [14]. We followed the same IRR process as SE (see Chapter 4’s System Evaluation - ‘Scoring and IRR’ section for more details), including the same style of scoring, summing points, establishing the codebook, handling sparse codes, and accounting for case-by-case discussion, etc. Like with SE, we report (a) percentage agreement where possible (92%, $\sigma = 8.2\%$ over 77/80 codes), (b) Cohen’s weighted kappa on the subset of codes (0.79 ($\sigma = 0.19$) over 31/80 codes) from (a) where the weighted kappa metric could be applied, and (c) the remaining codes that required case-by-case discussion on all data (3 for KM).

Evaluation of Participants’ Performance During Tool Use To ensure robustness in our tool use metrics (i.e., per-example scores of participants while using the tool or control), we developed a tool use rubric similar to Table 5.1. The experimenter scored completed cycles of KM and control tool-use data (i.e., filled out spreadsheets), and had an LLM re-score them. This approach is loosely inspired by recent developments in human-AI collaboration [133], [134]. Discrepancies greater than one point between the experimenter and LLM scores were scrutinized and resolved through manual review, either by correcting errors or making judgment calls. Since our rubric for these learning outcomes was objective and primarily checked for specific CSS techniques, this approach double-checked our mechanism measures and helped identify any inconsistencies.

Additional Interventions: Beyond relevant interventions described in the SE study, we asked participants to keep track of examples they already explored (can be enforced in the interface in the future), resize the page (automatable through rule-based checks), and note definitions (e.g., visual layout feature/CSS Technique).

⁵The tests of one KM participant were coded only by the experimenter due to time/technical constraints, using established rubric standards.

We also encouraged participants to strive for coverage of granular layout features and underlying CSS at the start of each new cycle (auto-detectable in future with rules since Ply [13] provides relevant CSS of each example), to not enter CSS where visual features are required and vice versa (a classifier or zero-shot prompt could do this), to ensure the code behind visual differences and similarities are distinct (a rule-based check could do this), to cover all sections of the web pages in their answers if they were not doing so (a computer vision-based classifier could break down sites in the future), and to focus on visual layout features rather than aesthetics like color or typography if they were discussing the latter (a classifier or zero-shot prompt could reference the list of visual features and Ply-extracted CSS to determine whether the user is listing visual layout features and even whether suggested overarching features would cover a large number of not-yet-covered sites each cycle).

5.4 KM Evaluation Results: Process Management’s Role in Attention to Detail

We found that KM users achieved greater knowledge gains around granular CSS techniques than control participants. KM users had an average gain of 25.2 ($\sigma = 6.5$) out of 80, while control users had an average gain of 5.6 ($\sigma = 4.8$) out of 80. Experimental participant data points are presented in Table 5.2 and Figure 5.17, while control participant data points can be found in Table 5.3 and Figure 5.18.

Quantitative Analysis: To analyze KM results, we leveraged a two-tailed Mann-Whitney U analysis identical to the one performed on SE data. The Mann-Whitney U statistic was 0 ($p = 0.01$), indicating a significant difference between the groups. The effect size ($r = 0.8$, $r_b = 1.0$) also demonstrated a substantial difference, and the sensitivity/robustness check yielded a simulated power > 0.8 . Although these results are encouraging and pass the same simulated power check as SE, they stem from an exploratory 5v5 design; replication with larger, “in-the-wild” samples is

Learning Outcomes and Benchmarks	P2	P4	P5	P7	P9
pre-test score	13	5	17	18	20
Italic, During Tool Use (DTU)	7/11	6/11	9/11	11/11	9/11
Masterclass (DTU)	4/11	3/11	10/11	11/11	11/11
Flat Icons (DTU)	5/7	5/7	7/7	7/7	5/7
Smashing Magazine (DTU)	3/11	8/11	9/11	9/11	5/11
Hero Icons (DTU)	8/9	7/9	8/9	6/9	6/9
CSS Tricks (DTU)	1/9	2/9	6/9	6/9	5/9
post-test score	27	34	45	43	50
Num Differences Found (DTU)	13	9	17	14	16
Num Same VF, Diff CSS (DTU)	2	4	6	4	5
Num Diff VF, Same CSS (DTU)	2	2	3	4	5
knowledge gains	14	29	28	25	30

Table 5.2: Table showing tool use statistics for KM users, including the number of granular visual layout differences found over ~ 3 cycles, as well as the number of user-discovered cases of similar visual features arising from different CSS techniques and vice versa. The table also shows learning outcomes KM users reached during tool use and their pre/post-test scores. The tool-use-specific learning outcomes included diverse aspects such as (a) flex nuances and transformations in Flat Icons and Smashing Magazine, (b) sophisticated transitions and scrolling behavior in CSS Tricks, and (c) intricate grid and spacing properties in Italic, Hero Icons, and Masterclass.

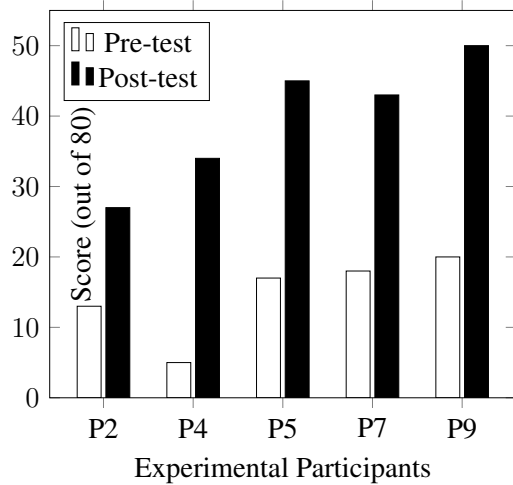


Figure 5.17: Comparison of pre-test and post-test scores for experimental (KM) users.

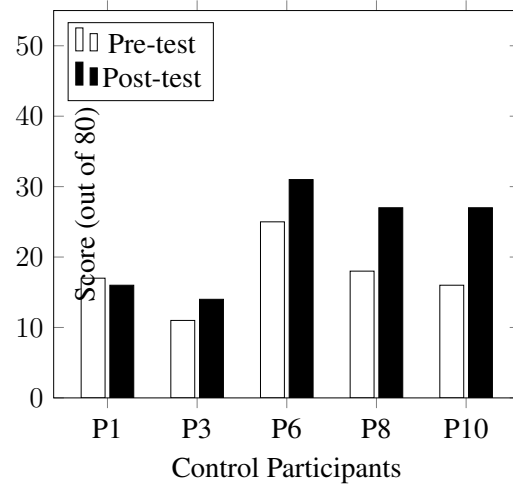


Figure 5.18: Comparison of pre-test and post-test scores for control users.

Learning Outcomes and Benchmarks	P1	P3	P6	P8	P10
pre-test score	17	11	25	18	16
Italic, During Tool Use (DTU)	6/11	5/11	4/11	6/11	7/11
Masterclass (DTU)	4/11	10/11	6/11	10/11	11/11
Flat Icons (DTU)	3/7	3/7	2/7	4/7	2/7
Smashing Magazine (DTU)	1/11	4/11	0/11	6/11	2/11
Hero Icons (DTU)	8/9	4/9	0/9	4/9	7/9
CSS Tricks (DTU)	1/9	1/9	4/9	4/9	0/9
post-test score	16	14	31	27	27
knowledge gains	-1	3	6	9	11

Table 5.3: Table showing tool use statistics for control users, such as learning outcomes reached during tool use, along with pre/post-test scores (out of 80).

still needed (as noted in the SE analysis).

Qualitative Mechanism Analysis: We found KM participants contrasted examples to discover, on average, ~ 14 granular differences across 3 cycles of tool use. All KM users tinkered and toggled with properties and values and found, on average, 3-4 cases of different CSS leading to similar visual features and vice versa.

Users described KM's process management as helpful, providing a rating of 4 or 5 (out of 5) for the **visual grouping** (PM1), **comparing/contrasting** (PM6), **code mapping** (PM7), **reflection** (PM8), and **tinkering/toggling** (PM5) activities. While there was no substantial difference between how the users rated the experimental ($\mu = 4.9, \sigma = 0.32$) and control ($\mu = 4.7, \sigma = 0.48$) tools, the experimental condition participants seemed to have engaged in more effective sensemaking and learned more about granular CSS techniques. The control participants mainly focused on code reading/copying, overfitting to particular techniques (flex, CSS overriding, etc.). In contrast, the experimental participants explored the niche CSS techniques behind diverse layout differences across the examples. With the data in mind, we can now answer our research questions.

We proceed to examine the exploration (RQ1) and learning (RQ2) of KM and control participants.

Experimental Participants' Exploration (RQ1): Participants effectively utilized the tool's features—such as grouping, comparing and contrasting, code mapping, tinkering, toggling, and reflection—to deepen their understanding of granular CSS techniques. Grouping websites by shared visual layouts, KM users identified similarities and differences in both appearance and underlying code. P4 noted, *“The grouping helped me realise that though some sites looked alike they used completely different techniques,”* suggesting that comparing and contrasting sites within a group, **based only on salient visuals and not code** (PM3, PM4), revealed differentiated sub-features, which eventually mapped to different CSS implementations. Participants found mapping **previ-**

ously identified visual features (PM2) to code particularly enlightening. P5 stated that *“diving deep into each feature really improved my understanding of the CSS,”* indicating that such mapping enhanced their comprehension. Tinkering and toggling with code allowed participants to see the impact of specific CSS properties firsthand. P2 reflected, *“I was able to see deeply the importance of a particular attribute in CSS, if the attribute is missing, the code won’t work,”* emphasizing the hands-on experience in understanding the functionality of CSS attributes. Similarly, P9 mentioned that *“mapping visual similarities and differences to CSS code”* helped in understanding the different ways layout and items in a website are arranged. This finding aligns with how the tool facilitated the identification of similar code leading to different visual features, and vice versa. Overall, the tool’s structured approach and interactive features appeared to enable participants to effectively explore and understand granular CSS techniques through practical application and critical analysis (see Table 5.2 for tool-use-specific scores).

Control Participants’ Exploration (RQ1): Participants in the control group primarily used CDT features such as inspection tools, grid/flex badges, and debugging features for overridden code. However, they often tended to focus on specific techniques like flex/flexbox and CSS overriding, which may have hindered their understanding of diverse granular CSS techniques compared to participants in the experimental condition. P1 highlighted the usefulness of CDT, stating, *“using the computed tab to check the overall styles that are applied to an element by the browser [helps you] know if the style declaration you are looking for’s... effect was actually applied.”* Despite this, P1 focused excessively on page inspection and CSS overriding without trying to identify granular visual features and map them to CSS, suggesting a superficial, **surface-level** engagement with the material. P3 and P6 similarly missed important concepts, spending too much time mechanically copying code from the cheat sheet or fixating on flex. P6 leveraged grid and flex badges for property identification but expressed a desire for more intuitive tools, saying, *“I wish this tool had more*

customization options for displaying code properties and layouts other than [what] CDT [offers].”

This reliance on CDT features without deeper analysis led to overfitting on particular techniques. P8 extensively read **complex syntax-heavy code** and used reference materials but overemphasized flex/flexbox as a solution, mentioning that tinkering with code properties *“did impact my understanding of the code properties and ... [how] they would affect the visual layout of a website,”* yet primarily focusing on flex/flexbox adjustments. Overall, while these participants effectively used CDT for code inspection and manipulation, they tended to overfit to certain techniques and did not delve into mapping granular visual features to diverse CSS implementations. Such exploration resulted in a less comprehensive understanding of niche CSS than participants in the experimental condition (see Table 5.3 for tool-use-specific scores).

Experimental Participants’ Learning (RQ2): We measured whether learners could move beyond recognition and transfer CSS concepts into new applications, i.e., examples different from those dissected in tool use [135]. KM users appeared to demonstrate significant transferable knowledge gains from the pre-test to the post-test and effectively reached learning outcomes. Initially, they focused on basic aspects of website design, such as general layout and simple CSS properties. For instance, P2 noted, *“The website is well responsive, the width for content elements are well arranged for both desktop and mobile views,”* highlighting their practical grasp of responsive design using flex and grid layouts. By the post-test, they showcased a more profound comprehension of granular visual layout features and underlying CSS techniques. P2 described *media queries* more clearly, stating, *“Some elements on desktop view are displayed in a grid list mode, while on mobile view are displayed horizontally,”* illustrating their ability to translate theoretical knowledge of CSS into practical solutions across multiple screen types. They also predicted the code (e.g., *“transition: all .2s”*) behind advanced concepts such as animations, transitions, and hover effects—showing how CSS ideas from tool-use examples can be reused to enhance the user

experience in new web contexts. Moreover, they specified techniques such as “*grid-template-columns: 1fr 1fr 1fr 1fr;*” and “*overflow-x:*”, further demonstrating detail-orientation. Similarly, P4 moved from high-level observations to detailed knowledge of specific CSS properties, explaining, “*Display grid was used because it creates a grid-like structure where the dev... has control over both the rows and columns...*” with “*grid-column-start set[ting] where a certain grid item starts from*”. P5 expanded their understanding to include grid layouts and media queries, recognizing that “*The grid layout provides a feature to control items individually and hence creates dynamic layouts...Media query helps to build responsive layouts, and for this website, we had multiple breakpoints.*” These descriptions are examples of applying complex design logic to the new examples in the post-test. P9 connected how specific CSS dependencies, like *media queries* shaping *grid-template-columns*, work to create visual outcomes, demonstrating a comprehensive understanding of how advanced CSS techniques influence both layout and interactivity. Finally, in their post-tests, several KM users also demonstrated knowledge of intricate techniques like *overflow: scroll/hidden* and *flex-basis*. These progressions indicate that participants effectively reached the learning outcomes in the post-test. Given the study setup, it is plausible that learners transferred their nuanced understanding of CSS code and visual layout features from the tool-use examples to the new applications.

Control Participants’ Learning (RQ2): Control participants exhibited limited progression in their understanding of granular CSS techniques from the pre-test to post-test, occasionally overfitting to specific methods like flex/flexbox, **unaware** of more advanced sub-properties or alternatives. Initially, they recognized general layout structures using basic CSS properties; for instance, P1 observed, “*The site has a mixture of different layout techniques such as Grid and Flex...*” and provided simple code examples for grid layouts. P3 mentioned, “*They choose to use display: flex for [the] layout structure of element[s] in rows or columns,*” focusing on high-level properties. By

the post-test, their comprehension had not significantly deepened. P1 continued concentrating on basic layouts without discussing advanced techniques, stating, *“There are no specific transitions and transformations for the website layout.”* Similarly, P3 still emphasized basic flex layouts and general animations, saying, *“They use these CSS techniques because of transition and animation of moving elements.”* Still, they missed the precise location and code behind some transitions. P6 mainly described layout features without consistently detailing advanced CSS techniques for all the sections where they were relevant (e.g., applying *flex* but missing sub-properties like *flex-basis*). They noted, *“These CSS techniques, specifically grid and flex-box, are responsible for the layout because they provide a flexible and powerful way of arranging elements.”* P8’s understanding remained somewhat superficial, focusing on general aspects such as *“Transition, Transformation and Animations: These techniques enhance interactions of the site”* and *“Grid and Flexbox: These layouts improve the flexibility and responsiveness of the site,”* while sometimes missing specific CSS sub-properties/values or advanced techniques that produce what we see (e.g. *grid-template-columns 1frx6* or the use of media queries to switch layouts at lower screen widths). Or, they mentioned detailed syntaxes for sections where those syntaxes were inapplicable. This performance indicates that even by the post-test, their understanding of granular CSS techniques was not substantially improved. They tended to over-rely on specific methods like flexbox without achieving more in-depth comprehension or **knowledge transfer**.

5.5 KM Discussion - Practical Learning and Technical Contributions

The results of our controlled study indicate that Knowledge Maps (KM) significantly enhances learners’ ability to identify, understand, and transfer granular CSS techniques across different websites. Compared to the control group—who relied on standard DevTools and static code snippets—KM participants showed markedly larger post-test improvements, discovering subtle differ-

ences in professional layouts and applying CSS sub-properties more effectively in new contexts. Qualitative evidence also suggests that KM’s comparison-heavy workflow motivated participants to systematically explore the nuances of multiple complex examples (e.g., media queries shaping grid or flex behavior, intricate subproperty use), leading to deeper insight into both common and divergent CSS strategies. KM, like SE, is structured around three core organizing principles: intuitive visual clustering of websites, linking prior and current insights (e.g., visual clusters influence nuanced differences discovered), and comparing/mapping conceptual insights to code differences.

On the practical and technical fronts, KM adopts a straightforward yet powerful approach for exposing and comparing relevant CSS features and code, bundling them with reflective prompts in an iterative mapping process. Specifically, this stepwise workflow mirrors the three principles: visual clustering reduces cognitive load and primes attention; linking prior and current insights builds a cumulative mental model in a focused manner; and comparing/connecting conceptual insights to code differences solidifies transferable understanding. This design approach, while simple in its use of existing technology, proves effective in practice: even small interactive features—like toggling and tinkering with highlighted properties—helped participants see the nuances that drive distinct visual outcomes. The data demonstrate that KM’s streamlined implementation worked because it guided learners through a stepwise process of observation, comparison, and reflection, ensuring that each subtle property or layout mechanic was noticed and conceptually integrated. Such attention to detail and conceptual reorganization are otherwise pain points for non-expert learners [29], [43].

From a learning perspective, KM’s compare-and-contrast exercises, rooted in prior work on the right “time for telling” and conceptual priming [29], appear to foster deeper conceptual knowledge than more open-ended approaches. By first grouping sites by apparent visual similarities, KM users build an intuitive foundation on which to notice small differences. This initial visual

clustering lays the groundwork for the subsequent contrasting and code mapping cycles, presented as the user is ready for them. Repeatedly mapping these differences back to their precise code triggers a cycle of hypothesis, experimentation, and reflection that is hard to replicate with ad hoc surface-level browsing (e.g., noticing a particular syntax such as `grid`). The interactive toggles further amplify this effect by letting learners directly see how changing a value (e.g., switching *grid-template-columns: 1fr 1fr* to *auto-fill*) affects responsiveness, spacing, or overflow—all essential yet easily overlooked aspects of professional CSS. Over time, these iterative cycles enabled learners to recognize advanced sub-properties—such as *flex-basis* and nuance in *grid-template-columns*—and effectively apply them to new layouts, resulting in significantly deeper and more transferable CSS proficiency (e.g., seeing use cases of media queries during tool use reappear in post-test examples).

Finally, these findings hold important implications for addressing the practical problem of teaching advanced CSS. Our study suggests that novices can master nuanced layout techniques if supported by processes that highlight relevant examples, compare them systematically, and connect each visual observation to the underlying code. By explicitly guiding learners through the three organizing principles (i.e., intuitive sensemaking order, prior knowledge linking, comparison and code mapping), KM parallels the processes embodied in SE, reinforcing the broader value of process-management frameworks for educational tools. In other words, when learners are guided to contrast multiple examples in detail—rather than passively reading or copying code—they not only map CSS techniques to the visuals they afford but also build transferable knowledge of said techniques. This evidence further supports the broader value of process management frameworks for educational tools, illustrating that even intricate web development concepts and knowledge transfer become accessible when embedded in an intentionally structured learning process.

CHAPTER 6

RALE MODULES (RM), A PM TOOL FOR PYTHON DESIGN PATTERNS

The exploratory case study and early iterations of SE and KM increased our confidence in process management for helping learners build knowledge by exploring complex professional source code. However, applying diverse design patterns and architectures is an unaddressed gap that differentiates novices from experts. Like with SE and KM, we investigate how intermediate developers can learn about these patterns from professional code. To do this, we developed RALE Modules (Figure 6.1), a system that manages intermediate-level developers' sensemaking process to help them see how and why design patterns are used in professional Python examples. The system aims to help learners overcome the complexity of professional codebases, where it is difficult to know where to start or what to do, let alone how concepts behind design patterns are instantiated and what alternatives could have been considered. To do so, RM maps mechanism and rationale instantiations across theory, a tutorial example, and a professional example. The system offers a holistic perspective by surfacing design patterns sourced from the entire professional codebase, yet remains specific by presenting only the information necessary to understand these patterns and architectures.

Professional Python developers are expected to be able to (a) recognise the *core schema* of a design pattern (e.g., a Mixin or an MVC/MVT triad), (b) trace *how* each piece of that schema is realised across tutorial code and a production-scale repository, and (c) articulate *why* the pattern fits the professional domain by citing concrete goals, mechanisms, and trade-offs. Concretely, a developer should be able to open an unfamiliar file in a professional codebase and immediately spot pattern instantiations. For example, they may see how various mixins are being “mixed-

and-matched” by adopter classes that present domain-specific data. They can explain, in prose or by sketching, how different mixins’ respective methods fire in sequence to achieve distinct functionalities for their adopter classes.

Novices often struggle with these skills. Such domain-specific application knowledge is rarely covered through textbooks and tutorials, as they present design patterns in isolation through pristine UML diagrams or toy examples, but rarely the messy, cross-file reality of production systems. Tutorials also seldom ask learners to reason about *rationales*, such as why one pattern defeats its alternatives in a given domain. RM fills this gap by situating theory, tutorial, and professional code side-by-side, and encouraging mapping and reflection. Normally, teaching learners how to apply such patterns in practice is challenging because production repositories are large, and pattern instantiations are scattered and intertwined with domain logic. Novices do not know where to begin in their self-directed exploration, cannot follow data flow across layers, and often miss the architectural motivation behind code. Without guidance, they fixate on syntactic details and fail to see trade-offs. Yet this matters because pattern-level reasoning lets developers debug faster, maintain applications, weigh architectural alternatives, and communicate design intent with teammates.

However, in the context of design patterns, transfer of architectural knowledge across examples may be difficult because learners: (1) may not “spontaneously” activate relevant prior knowledge; (2) may possess procedural knowledge of patterns without understanding deeper features (what to do but not why); (3) may fail to abstract general principles and apply them across varied examples (knowing the abstract principle can help you adjust to new contexts); or (4) may not know how to apply concepts in the given context. Even when learners develop a mental model of a pattern, they may struggle to extend it beyond isolated examples, failing to connect definitions, tutorials, and professional applications.

In early study sessions, we observed this in that novices exploring real-world Python code-

bases featuring design patterns (e.g., mixins, MVC variants) often fell into ad-hoc browsing of individual files or classes, without recognizing how those pieces united into a coherent, although complex, architecture. For instance, they might have noticed a *Mixin* class but failed to see how it connects to specific *ViewSets* or integrates with multiple classes inheriting it in the same codebase. They might have *fixated on local, surface-level code details* (like a single method override) without understanding why a pattern exists or what design tradeoffs it addresses. Without structured guidance, they rarely looked for parallel patterns in other modules or considered alternatives (e.g., inheritance vs. mixins), resulting in knowledge silos. As a result, they *struggled to transfer* the design pattern concepts to new contexts, including in the code of professional examples, missing the architectural implications (e.g., domain constraints, maintenance tradeoffs). This left them with only a partial sense of how and why the design pattern worked, illustrating the challenges of **code complexity**, **surface-level focus**, and limited **awareness** of possible alternatives in production-scale Python applications. Abstracting from these sessions helped us formulate the problem at hand conceptually.

Problem: Learners often struggle to recognize structures, patterns, and their rationale in **complex** professional codebases. They have limited working memory and schemas during program comprehension [21], [136], [137], and hence may suffer from high cognitive demand when attempting to keep track of multiple coordinating components (e.g., complex files, patterns, business logic, functions/methods, etc.). Even if learners have some sensemaking processes and consider important example specifics, they may **struggle to transfer** design pattern theory into professional practice. They do not know where to start (e.g., in a UI with heavy REST API usage, in CDT's network tab to see how data is loaded), how to chunk the problem, which tools to use (e.g., CDT and its console logging to print data, VS Code with breakpoints), or what to do (e.g., reverse engineer a URL response to uncover data flow). In this way, they often fail to uncover different layers

within the codebase, from the database to the frontend, or how design patterns manifest across them. Also, by default, learners are **unaware** that they must consider the application domain and stakeholders, such as end users and other programmers. These considerations are often helpful for understanding tradeoffs of why a pattern is chosen over its alternatives, e.g., choosing MVC (model-view-controller architecture) over no framework for a complex user analytics codebase so that frontend and backend developers do not tamper with each other's code. Many of these tradeoffs may not be in the learner's purview. Yet, such specifics are helpful for linking the general rationale for a pattern with the professional use case.

Moreover, learners may struggle to integrate theoretical, tutorial, and professional contexts because they view them at a **surface level**, as isolated constructs rather than manifestations of deeper concepts. For instance, though they may see many classes adopt mixins in a professional codebase, they may not be able to see why a mixin—which represents a slice of functionality (e.g., an authentication check)—should be mixed and matched rather than inherited, an option they may not have even considered. They may find it challenging to adapt the abstract schema and its rationale and tradeoffs to the professional context. And because learners struggle to understand and apply design patterns in professional contexts, they may have trouble gaining expertise and transitioning from intermediate to expert. So, we considered these obstacles while designing RM. The system strives to scaffold the analogical transfer of Python design patterns across contexts and to manage the learners' process as they explore complex professional code.

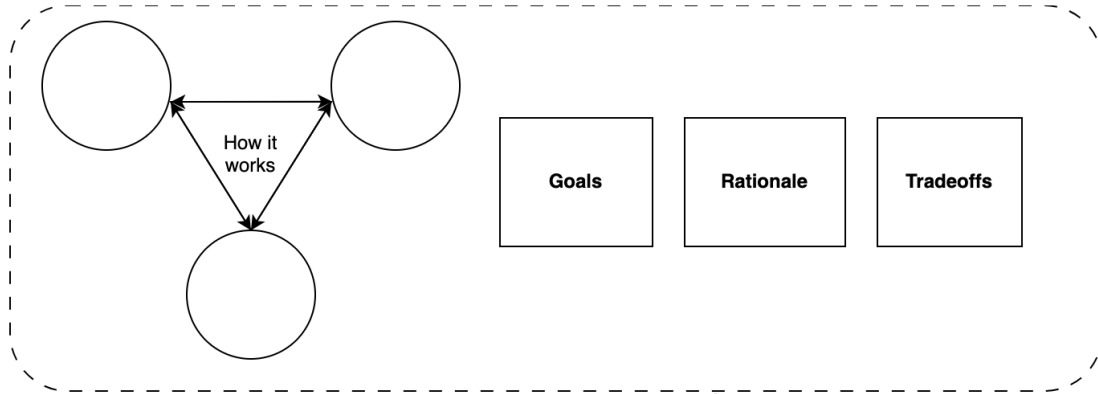
6.1 How RM enacts PM: Intuitive Diagrams, Analogical Mapping, Comparing Tradeoffs, Reflection

System Overview:

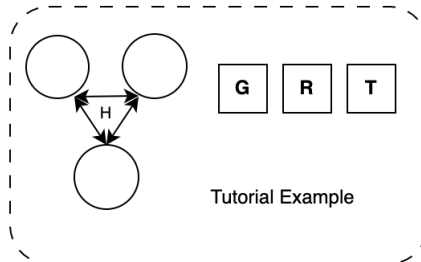
RM supports the sensemaking process of an intermediate developer learning about design pat-

RALE Modules

1. Surface a greater schema: Provide the definition. Pull out and autogenerate a shared visual representation of the pattern itself—a core schema. Augment it with how and why the concept works in theory—i.e. mechanism, goals, rationale, and tradeoffs.

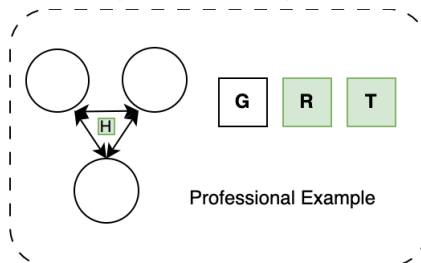


2. Mapping: Scaffold the learner to map the greater schema from the definition / theoretical context to a tutorial example / canonical context.



Analogical Transfer Scaffolds

3. Mapping Again: Scaffold the learner to map the greater schema from the theoretical and canonical contexts into a professional context. So far, all information has been provided or autogenerated for learners. However, in the professional context we will have learners infer and fill in the how, why, and tradeoffs, since such information is difficult to parse as it depends on the domain, and since such articulation activities are good practice for sensemaking and active learning.



Code / Data Flow / State Scaffold: We provide select code snippets and data flow (e.g. inputs / outputs of functions / methods). This constrains sensemaking and helps learners draw on their intuition about program state. It helps them see how the pattern is applied in practice.



Visual Diagrams Scaffold: We autogenerate intuitive visual diagrams of how design patterns work in professional contexts in order to help learners understand complex codebases.

Table Structure Scaffold: RM's activities line up the mechanism (how), rationale (why), and tradeoffs of the definition and tutorial example, in a table-like structure, in order to setup the learner for analogically mapping those concepts into the professional example's domain.

Def.	Tut.	Pro.
How	How	?
Why	Why	?

Custom Activities Scaffold: We autogenerate reflection questions customized to the professional example's variables, classes, and domain, in order to help learners infer and articulate how / why patterns work in the professional context.



Overcoming Mapping Challenges: Mapping into a professional context is challenging. Hence RM provides process management scaffolds.

Process Management Scaffolds

Figure 6.1: Process Management for Transfer in RALE Modules. Learners progress through three stages: (1) conceptual understanding by defining the core schema with its mechanism, goals, rationale, and tradeoffs, (2) mapping (PM7) the concept to a structured tutorial example, and (3) applying it in professional contexts where learners are prompted to reflect (PM8) on how [H] and why [R] it works, comparing and evaluating tradeoffs [T] (PM6). Diagrams (PM1), code snippets, and data flow (e.g., logs of specific data and program state, or function/method inputs/outputs) support each stage. By aligning the conceptual, tutorial, and professional cases, the model intends to foster deep understanding and robust knowledge transfer across the three contexts.

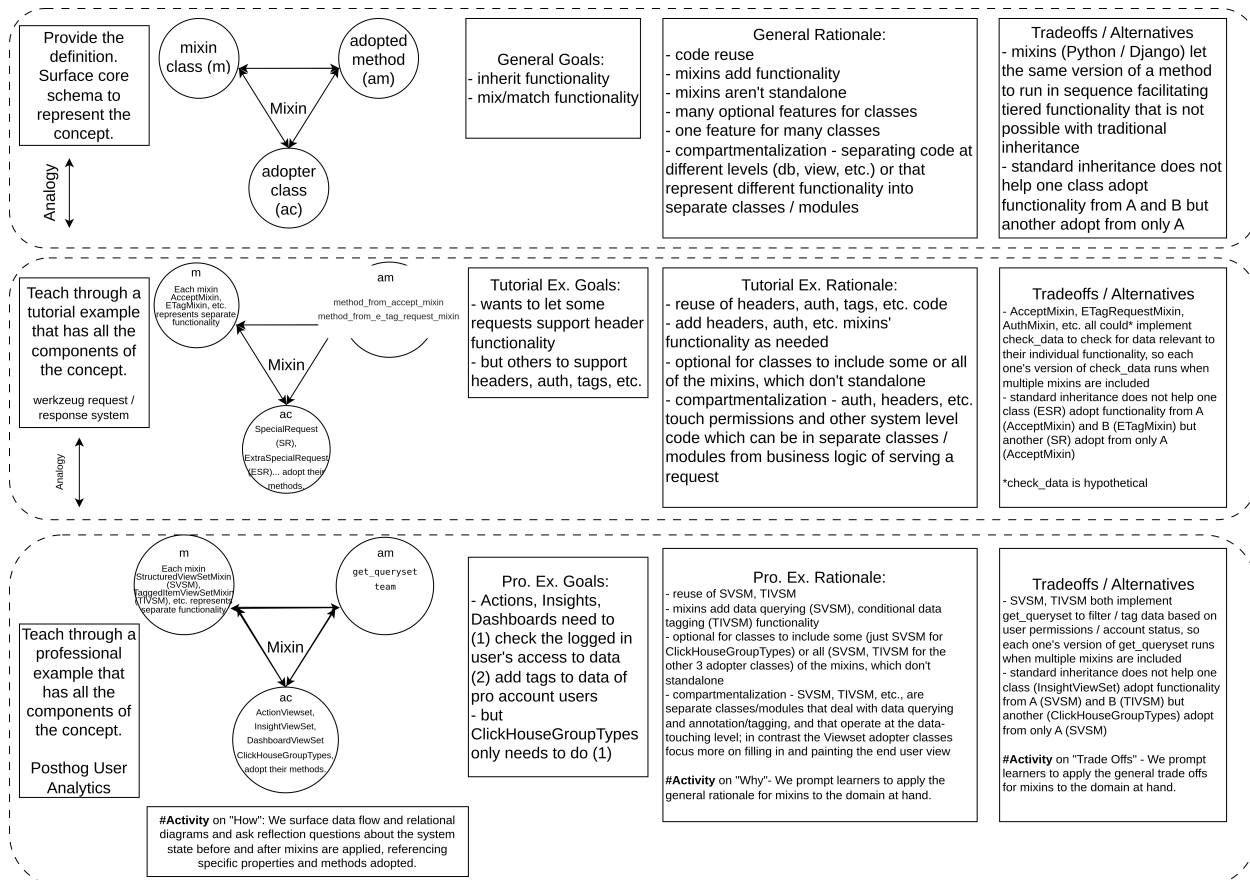


Figure 6.2: Knowledge Taught by RALE Modules. This figure demonstrates the application of RALE modules across three stages, using the pattern of mixins in Python/Django: (1) *Core Schema*: Introduces the concept of mixins (e.g., *Mixin*, adopter classes that inherit it, and associated adopted methods) to show how functionality can be inherited and combined across multiple classes. (2) *Tutorial Example*: Uses a request/response system with classes like *SpecialRequest* and *ExtraSpecialRequest* to explain how mixins (e.g., *AcceptMixin*, *ETagMixin*) provide reusable functionality, such as headers and tags. We highlight the tradeoffs of mixins versus traditional inheritance, showing how mixins allow flexible class combinations while avoiding rigid class hierarchies. (3) *Professional Example*: Applies mixins to a user analytics system, where different dashboards (e.g., *ActionViewSet*, *ClickHouseGroupTypes*) require specialized querying functionality. Learners using RM are guided to identify patterns, evaluate tradeoffs, and understand when to adopt each mixin based on their specific goals (e.g., filtering data or conditional tagging). Each stage includes scaffolds or prompts to explore this knowledge, aiming to foster deep understanding and transfer.

terns and architectures from professional code. Since learners lack intuitive starting points, explicit pattern schemas (mental models), or the ability to map those schemas into professional contexts, our system has learners proceed row by row through a table of activities mapping definitions, goals, mechanisms, rationales, and tradeoffs across theoretical, tutorial, and professional domains. Notably, however, the learners are not provided the professional example's mechanisms (*how*), rationale (*why*), and architectural tradeoffs. Instead, they are provided with autogenerated scaffolding prompts that challenge them to map from the theoretical and tutorial domains into the professional context as an exercise of transfer. In this way, RM users proceed through three high-level steps as shown in Figure 6.1. The steps reflect the PM framework and the structure of the analogies being taught. These steps, figures, and themes are explained in more detail below. Throughout, we leverage the Posthog user analytics dashboard as the professional example because it is a case of code that is adopted widely by large organizations and yet is open source, making it a prime target for demonstrating industry-grade design pattern schemas.

- **Step 1: Surface a greater schema.** Learners begin with a high-level definition of the design pattern and its abstract visual schema (e.g., mixins are structured around methods and adopter classes, all of which interact, Figure 6.2 (1)). The system provides a UML-inspired visual diagram showing the pattern's mechanism and aligns it with its goals, rationale, and tradeoffs. These conceptual elements give learners a theory-based mental model to anchor their understanding.
- **Step 2: Map the schema to a canonical example.** Next, the system scaffolds learners to connect the abstract schema to a tutorial or simplified code example (e.g., Werkzeug, a simple utility, uses mixins to make some HTTP requests accept headers but make other requests accept both headers and etags, Figure 6.2 (2)). Here, learners see how mechanisms, rationales, and tradeoffs from theory are implemented in an example. This explicit mapping is designed

to support analogical transfer and prepare learners for the next step.

- **Step 3: Map the rationale to the professional context.** Finally, learners are guided to apply the design pattern schema to a complex professional codebase (e.g., Posthog uses mixins to allow some accounts to access data tags but not others, Figure 6.2 (3)). In this stage, they are no longer shown the rationale directly. Instead, they must reflect on how and why the pattern works in the example, drawing analogies to the theory and tutorial. This sensemaking process is supported by targeted activities and auto-generated diagrams that help learners articulate mechanisms, rationales, and tradeoffs specific to the domain and stakeholder needs. This is meant to help with transfer, engaging both analogy and reflection.

Figure 6.1 visually summarizes this three-stage mapping which is at the heart of RALE Modules. At the top, the figure shows how learners begin by defining the core schema of a design pattern, including its mechanism, goals, rationale, and tradeoffs. The central section illustrates the mapping of this schema to a structured tutorial example, highlighting how each conceptual element is instantiated in sample code. At the bottom, the figure depicts the application of the pattern in a professional context, where learners are prompted to reflect on how and why the pattern works, and to compare tradeoffs. The flow from top to bottom, with supporting diagrams and code snippets, makes clear how RM scaffolds the transfer of knowledge from theory to practice, and how each stage builds on the last to deepen understanding.

Next, Figure 6.2¹ demonstrates the application of RM's process to the mixin pattern in Python/Django. The top section introduces the core schema, with labeled boxes for mixins, adopter classes,

¹Our actual prototype adheres to these figures' steps with a few minor deviations due to practical implementation constraints: (i) the design pattern/tutorial mechanisms were presented with the definitions, (ii) goals were presented between definitions and diagrams, (iii) some basic scaffolding questions were presented before or with the code/data, (iv) some mechanism questions were in the same section as the rationale ones, (v) some mechanism and rationale questions were presented in a non-tabular structure (for MV*, not mixins in the Posthog example). Also, while the key professional example sections were autogenerated, the context, goals, code/data, and some Django-related footnotes were wizeded.

and adopted methods, visually showing how functionality is inherited and combined. The middle section walks through a tutorial example, illustrating how mixins provide reusable functionality in canonical contexts. The bottom section applies these ideas to a professional analytics system, with arrows and annotations guiding the reader to see how different dashboards adopt specialized querying and tagging functionality via mixins. The system's structure makes the analogical mapping across the three contexts explicit, helping the learner see both the general principle and its domain-specific instantiation. This mapping structure is embedded in the RM system deployment that we describe in detail next.

Since the most effective analogies emphasize shared relationships, particularly higher-order ones that govern simpler ones [34], the RM system's table of activities has rows of (i) explanations and goals (Figure 6.3), (ii) diagrams (Figure 6.4), (iii) code snippets/data flow (as applicable, Figure 6.5, 6.6), (iv) mechanisms and rationales (Figure 6.7), and (v) tradeoffs (Figure 6.8) of design patterns, and columns representing definitions/theory, tutorial examples, and professional applications. The entirety of i-v (rows) is mapped across the three domains (columns). While the tool offers detailed instantiations of these five components in the definition/theory and tutorial columns, the professional example phase involves prompting articulation.

PM Solution

Like SE and KM, RM and its steps also provide structured process management centered on three core sensemaking approaches:

1. **Start the Sensemaking Process with Conceptual Pattern Definitions and Diagrams:** RM begins by presenting overarching definitions (Figure 6.3) and visual diagrams of patterns (Figure 6.4) such as mixins or MVC variants (as in step 1). This aligns novices on the broad idea, ensuring they have an intuitive grasp before viewing complex real-world code.
2. **Link Prior Pattern Knowledge to Concrete Implementations in Professional Examples:**

RM has learners map diagrams from previously studied theory (step 1) and tutorials (step 2) to analogous autogenerated diagrams for the professional example (step 3) in order to drive conceptual links. This approach encourages connecting conceptual knowledge to new practical contexts, preventing disjointed and surface-level file-by-file exploration that is not grounded in theoretical design pattern structures.

3. **Map Example-specifics:** By bridging each pattern’s conceptual definition (step 1) with the relevant segments in the professional codebase (step 3), RM prompts novices to see how multiple code components coordinate to implement a pattern. RM also encourages reflection on design alternatives (e.g., “Why choose a mixin over pure inheritance in this specific example?”), ultimately equipping learners with deeper architectural awareness and transferable insights.

6.1.0 *Illustration Example—Applying RALE Modules for Learning Mixins:*

To illustrate how RM’s steps and its PM approaches work, we consider the case of helping novices learn mixins through the professional example of Posthog, an end-user analytics dashboard.

The RM process begins with a detailed view of the design pattern definition as presented in Figure 6.3, which defines² mixins in the context of a tutorial example and a professional example, namely the Posthog user analytics dashboard (Figure 6.9). The diagram, goals, rationale, and tradeoffs will also be visually organized into the same three columns (theory, tutorial, and professional), thereby giving the learner a clear mental model to anchor their understanding of each. This layout encourages the learner to see how each conceptual element relates to the others (e.g., how mixins work in theory relates to how they work towards Posthog’s selective tagging functionality), and sets up the mapping activities that follow.

²Some mixins-related content from the prototype was adopted from <https://en.wikipedia.org/wiki/Mixin>.

Mixins

Check the boxes after reading the statements. Fill in the text boxes.

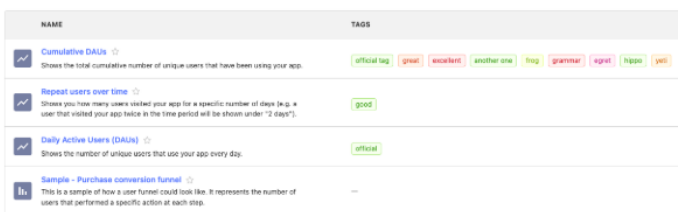
Theory	Canonical / Tutorial Example	Professional Example (posthog)												
<u>Definitions</u> ☐ A mixin is a class from which other classes can adopt methods and properties. ☐ Mixins are used if you don't want a class to "inherit" from another class (i.e. be its child class) but you want it to "include" some attributes / methods. ☐ You can think of mixins as uncles and aunts but not necessarily parents. They help avoid issues and complexities of multiple inheritance. e.g. If class TA has parents Student and Faculty, both of whose parent is Person, then does TA use Student or Faculty's version of any given method, or both's? (src: 1)	☐ For an example, consider werkzeug, which is an HTTP request and response system. Werkzeug needs to support regular HTTP requests, but also: - requests that accept <i>header support</i>, which enables the system to retrieve all the relevant HTTP accept headers, that in turn determine the type of content the client can support - requests that support accept headers, etags (entity tags which help the web cache only request for information it needs), authentication, and user agent support (which lets servers access frontend client data such as the OS, browser, etc.)	 ☐ Recall the video on how Posthog's features work for both regular and enterprise use. ☐ Note that a logged in user is generally part of a team. Depending on which team they are part of, they have access to different data. Depending on whether they are an enterprise user, they can tag insights, dashboards, and actions, but tagging is not applicable to groups. ☐ The application has complex functionality involving insights, dashboards, actions which involve both data and frontend views.												
<u>Goals</u> ☐ inherit functionality ☐ mix/match functionality	☐ wants to let some requests support header functionality ☐ but others to support headers, auth, tags, etc.	Actions, Insights, Dashboards need to: ☐ check the logged in user's access to data ☐ add tags to data of pro account users but ClickHouseGroupTypes only need to do #1.												
<u>Diagrams</u> Mixin → method1 Mixin 2 → method2	<u>Mixins of werkzeug:</u> <table> <tr> <td>AcceptMixin method_from_accept_mxn</td> <td>→</td> <td>SpecialRequest method_from_accept_mxn</td> </tr> <tr> <td>BaseRequest prop_of_basereq_mxn</td> <td>→</td> <td>prop_of_basereq_mxn</td> </tr> </table> AcceptMixin → method_from_accept_mxn ExtraSpecialRequest → method_from_accept_mxn	AcceptMixin method_from_accept_mxn	→	SpecialRequest method_from_accept_mxn	BaseRequest prop_of_basereq_mxn	→	prop_of_basereq_mxn	<u>Mixins of posthog:</u> <table> <tr> <td>StructuredViewSetMixin get_queryset</td> <td>→</td> <td>ClickhouseGroupsView get_queryset</td> </tr> <tr> <td>team</td> <td></td> <td>team</td> </tr> </table> StructuredViewSetMixin → get_queryset ActionViewSet → get_queryset	StructuredViewSetMixin get_queryset	→	ClickhouseGroupsView get_queryset	team		team
AcceptMixin method_from_accept_mxn	→	SpecialRequest method_from_accept_mxn												
BaseRequest prop_of_basereq_mxn	→	prop_of_basereq_mxn												
StructuredViewSetMixin get_queryset	→	ClickhouseGroupsView get_queryset												
team		team												

Figure 6.3: The design pattern definition and goals.

Mixins of posthog:

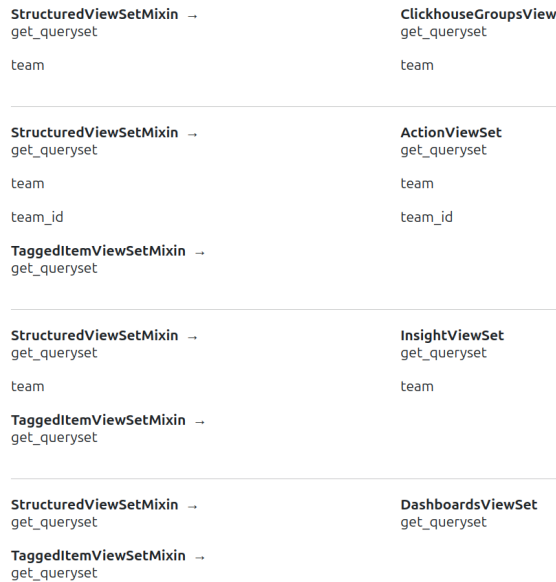


Figure 6.4: Diagram of the professional example's instantiation of the design pattern.

StructuredViewSetMixin code:

```
class StructuredViewSetMixin(_GenericViewSet):
    ...

    def get_queryset(self):
        queryset = super().get_queryset()
        to_return = self.filter_queryset_by_parents_lookups(queryset)
        return to_return

    @property
    def team_id(self) -> int:
        team_from_token = self._get_team_from_request()
        if team_from_token:
            return team_from_token.id
        if self.legacy_team_compatibility:
            user = cast(User, self.request.user)
            team = user.team
            assert team is not None
            return team.id
        return self.parents_query_dict["team_id"]

    @property
    def team(self) -> Team:
        team_from_token = self._get_team_from_request()
        if team_from_token:
            return team_from_token
        user = cast(User, self.request.user)
        if self.legacy_team_compatibility:
            team = user.team
            assert team is not None
            return team
        try:
            return Team.objects.get(id=self.team_id)
        except Team.DoesNotExist:
            raise NotFound(detail="Project not found.")
```

Figure 6.5: A sample of the professional example's code snippets.

Next, Figure 6.4 shows a UML-inspired architectural diagram of Posthog's use of the mixin pattern, with labeled components and arrows, in this case, indicating relationships between classes, mixins, and methods. This visual helps the learner trace how the abstract schema (e.g., of a mixin) is realized in real code in order to learn about the pattern's mechanism and inner workings. Figure 6.5 complements this by providing a sample of Posthog's code snippets, allowing the learner to connect the diagrammatic representation of mixins to actual implementation details, with core architectural components (e.g., mixins and their methods) highlighted. Together, these figures support the learner in bridging the gap between conceptual understanding and practical mechanisms.

Figure 6.6 presents a sample of Posthog's data flow. This is crucial for helping the learner understand not just the static structure of the code, but the dynamic behavior that results from the design pattern's implementation. By following the flow, the learner can see how different components interact to achieve the desired functionality of selective tagging.

Insight data before the TaggedItemViewSetMixin is applied:

This is data for the "Formative project" on the licensed account.

Insight object (21) Daily Active Users (DAUs) from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (22) Cumulative DAUs from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (23) Repeat users over time from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (24) Sample – Purchase conversion funnel from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (25) Users by browser (last 2 weeks) from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (26) Users by traffic source from TIVSM
tagged items: posthog.TaggedItem.None

and if we try to print pre_fetched tags for each of the above,
we get the following, for each of the above insight objects:
AttributeError: 'Insight' object has no attribute 'prefetched_tags'

Insight data after the TaggedItemViewSetMixin is applied:

This is data for the "Formative project" on the licensed account.

tivsm postInsight object (21)
tagged items:
tag: official

tivsm postInsight object (22)
tagged items:
tag: official tag
tag: great
tag: excellent
tag: another one
tag: frog
tag: grammar
tag: egret
tag: hippo

tivsm postInsight object (23), tagged items:
tag: good

tivsm postInsight object (24), tagged items:

tivsm postInsight object (25), tagged items:

tivsm postInsight object (26), tagged items:

Insight data before the TaggedItemViewSetMixin is applied:

This is data for the non-licensed account.

Insight object (21) Daily Active Users (DAUs) from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (22) Cumulative DAUs from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (23) Repeat users over time from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (24) Sample – Purchase conversion funnel from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (25) Users by browser (last 2 weeks) from TIVSM
tagged items: posthog.TaggedItem.None

Insight object (26) Users by traffic source from TIVSM
tagged items: posthog.TaggedItem.None

and if we try to print pre_fetched tags for each of the above,
we get the following, for each of the above insight objects:
AttributeError: 'Insight' object has no attribute 'prefetched_tags'

Insight data after the TaggedItemViewSetMixin is applied:

This is data for the non-licensed account.

tivsm postInsight object (21)
tagged items:

nothing was printed here
tivsm postInsight object (22)

tagged items:
nothing was printed here

tivsm postInsight object (23)
tagged items:

nothing was printed here
tivsm postInsight object (24)

tagged items:
nothing was printed here

tivsm postInsight object (25)
tagged items:

nothing was printed here
tivsm postInsight object (26)

tagged items:
nothing was printed here

and if we try to print pre_fetched tags for each of the above
we get the following for each of the above insight objects:
AttributeError: 'Insight' object has no attribute 'prefetched_tags'

Figure 6.6: A sample of the professional example's data flow.

Rationales

□ R1) A mixin is an object that we can use in order to add reusable functionality to another object or class, without using traditional inheritance. We can't use mixins on their own: their sole purpose is to add functionality to other objects or classes (src: 1, 2).

□ R2) Mixins can be used in this way by multiple classes, for code reusability. They are used to avoid code repetition and promote code reuse, so there is less complexity and room for error. This helps with collaboration.

□ R3) A mixin is a special kind of multiple inheritance. There are two main situations where mixins are used:

- You want to

□ R1 Ex) The difference is subtle, but in the werkzeug example, the mixin classes weren't made to stand on their own. In more traditional multiple inheritance, the AuthenticationMixin (for example) would probably be something more like Authenticator. That is, the class would probably be designed to stand on its own (src: 1, 2)

□ R2 Ex) Here werkzeug's SpecialRequest and ExtraSpecialRequest both adopt AcceptMixin.

□ R3 Ex) Here werkzeug is adding optional features for SpecialRequest and even more optional features for ExtraSpecialRequest.

□ R4 Ex) Here AcceptMixin, ETagRequestMixin, UserAgentMixin, AuthenticationMixin each represent compartmentalized implementations of particular functionalities.

□ R5 Ex) For instance in the werkzeug's request and response example, mixins let you allow more supports as needed, but not by default, to instances of Request. Adding mixins to classes let you layer on additional functionality as needed.

□ R6 Ex) Here, the werkzeug example's mixins (AcceptMixin, ETagRequestMixin, UserAgentMixin, AuthenticationMixin, etc.) respect DI (the dependency inversion principle) because each of their implementations could be changed as long as they implement the methods they promise to implement, which get used by the adopter classes (SpecialRequest, ExtraSpecialRequest, etc.). Similarly the adopter classes can change how they use the mixins and mixin methods they adopt depending only on the interface (i.e. the method signatures) of

StructuredViewSetMixin

get_queryset

Activity 3A: List the classes or class instances that adopt the method **get_queryset** from StructuredViewSetMixin. Hint, look at the "Mixins of posthog" diagram above.

Pacing: Perhaps reserve ~1 min for this question.

Activity 3B: Describe how StructuredViewSetMixin's **get_queryset** method is being used.

Pacing: Perhaps reserve ~2 min for this question.

i.e. What does StructuredViewSetMixin's **get_queryset** do?

i.e. what does the class that adopts StructuredViewSetMixin's **get_queryset** do?

Hint: look at the code of the classes you mentioned in 3A.

Activity 4: **With respect to get_queryset**, explain why mixins, and in particular StructuredViewSetMixin, is used in posthog?

Hint 1: see the general ideas bulleted to the left.

Hint 2: remember the "Multiple Versions of Methods / Props Note" to the left.

Hint 3: then take another look at the posthog application as a user.

Think carefully about the **domain / purpose** of the application you are studying. Why specifically are mixins used for this particular domain?

Pacing: Perhaps reserve ~2 min for this question.

team_id

Activity 3A: List the classes or class instances that adopt the method **team_id** from

Figure 6.7: A sample of the auto-generated activities around the professional example.

Subsequently, Figure 6.7 illustrates a sample of the mechanism and rationale-related activities that RM provides around Posthog. The figure highlights how the activities are tailored to the specific mixins and methods in the codebase (e.g., describing how the StructuredViewSetMixin's **get_queryset** method is used), prompting the learner to reflect on why certain design choices were made and how they relate to the broader architectural goals. The activities have the learner identify key components of a pattern (e.g., specific mixins' adopter classes, adopted methods, etc.), detail what each does, describe how they interrelate, and map each provided reason for why the pattern applies in theory or in a canonical example to a customized (learner-generated) reason for why it applies in the professional example of Posthog. This scaffolding is designed to deepen the learner's understanding of Posthog-specific mixin rationales in the interest of promoting transfer from theoretical reasoning to professional practice. It does so by helping the learner construct a mental model of how patterns work in theory and in practice and then reflect on why they work in

Pattern	Theory	Canonical / Tutorial Example	Tradeoffs
Mixin	□ Mixins let a class adopt methods and attributes of another class. Reasons: (A) code reuse (B) optional methods / attributes (C) compartmentalization □ A mixin class acts as the parent class, containing the desired functionality. A subclass can then inherit or simply reuse this functionality, but not as a means of specialization. Typically, the mixin will export the desired functionality to a child class, without creating a rigid, single "is a" relationship. Here lies the important difference between the concepts of mixins and inheritance, in that the child class can still inherit all the features of the parent class, but, the semantics about the child "being a kind of" the parent need not be necessarily applied. In this way also, mixins allow inheritance and use of only the desired features from the parent class, not necessarily all of the features from the parent class. Most languages don't give a mixin any special status; it's just a class that was designed to be "mixed in", rather than used standalone. It's a matter of the role of the class in the overall domain as to whether a mixin is a "primary" class or simply a mixin. You don't create objects / instances of the mixin classes but rather just mix the mixin classes' functionality into other classes. (taken from src: 1, 2, 3)	□ Werkzeug's Request / Response System Reasons: (A) all Requests (SpecialRequest, ExtraSpecialRequest, etc) can use the AcceptMixin mixin (B) the mixins let you allow more supports as needed, but not by default, to instances of the Request classes (i.e. AcceptMixin for SpecialRequest, 4 mixins for ExtraSpecialRequest) (C) functionality for generally reusable request-related features (AcceptMixin, ETagRequestMixin, etc.) can be separated from ExtraSpecialRequest-specific functionality (i.e. the logic in the <code>esr_method</code>) □ SpecialRequest, ExtraSpecialRequest, etc. are not types of AcceptMixin, ETagRequestMixin, UserAgentMixin, AuthenticationMixin, etc. The request classes ("children" / adopters) use methods of the mixin classes ("parents" / adoptees). The role of these mixins is to provide additional functionality and features to their adopter classes, and we do not create objects / instances of the Mixin classes.	Why were mixins useful for posthog specifically? Hint, think about tradeoffs over inheritance and about traditional multiple inheritance issues that mixins solve. Remember the "Multiple Versions of Methods / Props Note" above. Pacing: Perhaps reserve ~3 min for this question.
Inheritance	□ Reasons: (A) code reuse (B) allows for extending classes □ Inheritance enforces an "is a" relationship. The child class is a kind of the parent class.	□ Inheritance Example of Square / Cube SumComputer: (A) code is reused between Square and Cube SumComputer (B) a more specific class SquareSumComputer or CubeSumComputer can extend a more generic class SumComputer. □ SquareSumComputer is a SumComputer. It is a kind of SumComputer. CubeSumComputer is a SumComputer. It is a kind of SumComputer.	Why would inheritance not work for posthog specifically? Pacing: Perhaps reserve ~3 min for this question.

Figure 6.8: The tradeoffs exercise of the professional example.

theory and in practice.

Finally, Figure 6.8 shows the tradeoffs exercise for Posthog, with a table that prompts the learner to compare competing patterns (i.e., mixins vs. traditional inheritance) and articulate the rationale for choosing one over another. The figure's structure encourages the learner to think critically about the pros and cons of different approaches (e.g., mixins allow Posthog views to selectively include tagging functionality while traditional inheritance is more rigid).

Together, these steps, and the RM system at large, aim to help learners overcome the practical challenges that non-experts face when attempting to learn design patterns in professional contexts. Through these figures, we have seen how learners cover every aspect of a design pattern from its definition to its tradeoffs, and how they map that greater pattern schema from theory into a canonical example into a professional one. Next, we study RM from the lens of our PM heuristics.

6.1.0 Analysis of RM (PM heuristics lens):

Our approaches, viewed from a PM lens, mirror the core PM principles seen in SE and KM, but adapted for structured design pattern exploration in production-scale Python code. For each pattern, RM activities pull in relevant code structures from disparate parts of the codebase (Figure 6.5) and use them to build just-in-time **visual diagrams** (PM1) and **customized reflection questions** (PM8) to help the learner manage complexity based on their progress so far and hence their zone of proximal development [35], [138].

Throughout the process, RM guides the learner to connect the schema (definition/theory), tutorial, and professional examples, breaking down silos between them, while considering the domain and stakeholders. This **mapping** (PM7) spans all aspects—schemas, diagrams, goals, rationales, code/data (as applicable), and tradeoffs—helping the learner transfer their understanding to practical contexts. By **building on prior knowledge** (PM2) [26], such as the theory known to the learner and their answers to previous questions, the system seeks to ensure the learner gradually grasps the mechanics and reasons behind real-world design pattern use.

Diagrams and Activities: We hypothesized that RM’s features would reduce learners’ **extraneous cognitive load** (PM4) (e.g., finding resources, figuring out what to do, etc.) by providing clear visual diagrams and guiding them on what to do and in what order for effective sense-making. By linking otherwise disconnected components within the codebase, offering **relevant code snippets** (PM3), and sharing disciplinary insights on the mechanisms and rationales [26], the diagram-based activities simplify the complex learning process. The UML-inspired diagrams enhance program comprehension [139], helping developers avoid the “Turing tarpit” [140], where everything is technically possible, but abstraction and visualization are elusive. These diagrams provide a high-level architectural view, assisting learners in quickly grasping **the salient parts of the example’s structure** (PM3). They can then investigate the purpose of that structure within the

example-specific domain.

The diagrams and activities showcase professional example details in a way that guides the learner to delve into not only how but also why the pattern applies in the example. For instance, the mixin pattern diagrams (Figure 6.4) highlight how some classes adopt just one mixin, and others adopt multiple, which shows how mixins can be mixed and matched depending on specific needs. Moreover, the theoretical definitions and tutorial content also feature declarative explanations of how and why patterns are used, and learners **interact** (PM5) with the content by ticking checkboxes to denote their progress. The ordered questions and structured activities go further, helping the learner make sense of a professional codebase in the context of its domain.

Throughout the activities, we emphasize the application’s end-user functionalities and domain-specific goals, giving purpose to the architectural investigation [140] and helping prevent learners from fixating on unimportant aspects. Starting with a high-level architecture diagram of the definition and the tutorial example helps learners structure their knowledge [21], [136], [137], building a schema for how the design pattern works. Later, the system’s process management seeks to scaffold analogical transfer [34] by helping learners leverage that schema when they see the diagram structure applied to the professional example and explain the rationale of that schematic choice. Finally, RM prompting learners to evaluate and articulate tradeoffs helps them bridge theory and practice. As learners engage with more professional code examples in this way, we expect them to become increasingly proficient in applying design patterns and architectural principles.

Tradeoffs: The tradeoffs table (Figure 6.8) helps learners **compare competing patterns** (PM6) in real-world domains, such as choosing mixins over inheritance for certain features of Posthog’s analytics dashboard (Figure 6.9). Posthog mixes and matches two mixins: *StructuredViewSetMixin/SVSM*, which filters data based on a user’s access, *TaggedItemViewSetMixin/TIVSM*, which tags/annotates data based on a user’s privileges. To realize these functionalities, both these mixins

implement their own versions of the *get_queryset* method. If one adopts both mixins, then both versions of *get_queryset* run, enabling (a) filtering for user-accessible content and (b) tagging data. However, if one adopts only *SVSM*, then only one version of *get_queryset* runs (*SVSM*'s), and only the filtering functionality activates. On the other hand, if one used traditional inheritance instead of mixins, they would need to combine the two functionalities into one parent class, which compels them to inherit both functionalities even if only one is needed, or none at all. Traditional inheritance cannot support such inclusion and layering of multiple parent classes, sequentially running each of their renditions of a given method. Such distinctions are elucidated by RM's tradeoff tables, where learners should go beyond the surface-level and think deeply. They should compare and contrast [29] similarly applicable but competing techniques, such as mixins and inheritance, to discover the nuanced rationale for using the correct pattern (mixins in this case) in the given real-world situation.

Side-by-side Layout: To prevent **cognitive overload** (PM4), the system presents diagrams, rationales, code snippets, data flow (e.g., function/method input/output as in Figure 6.6), activities, and tradeoffs in a side-by-side layout (as in Figures 6.3, 6.7, 6.8). This unified interface is designed to equip learners to quickly grasp high-level architectures before diving into real-world implementations, and to easily map rationales into practice. Figure 6.10 depicts a learner mapping rationales from theoretical and canonical contexts into the professional context. The figure highlights how the learner draws connections between abstract principles (learned previously, labeled R1-R7) and concrete implementations, showcasing the process of analogical transfer that RM is designed to support. The activities begin with questions about how patterns function (Figure 6.7) and gradually progress to questions addressing the rationale behind their use in the specific example. These guided prompts and opportunities for self-explanation [18] serve to discourage aimless exploration and encourage meaningful inquiry. The visuals and PM activities help intermediate

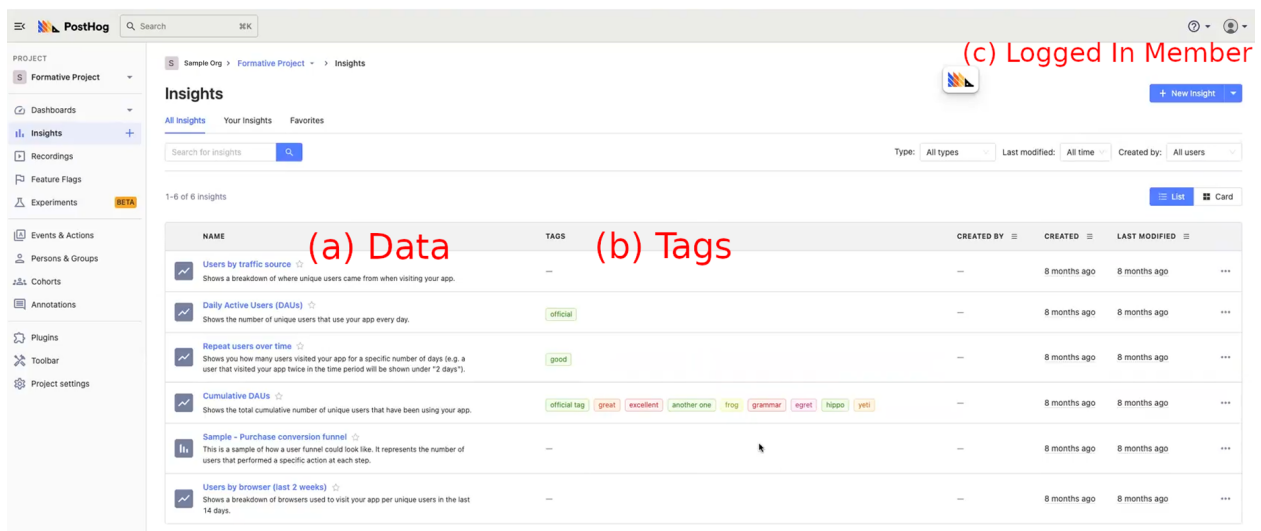


Figure 6.9: Posthog is an open-source product analytics dashboard that helps software development teams analyze their users' behavior. It uses mixins to determine which logged-in team members (c) can access which users' data (a). It also uses mixins to determine team members' Posthog license status, affecting their ability to tag data (b) with labels. Finally, Posthog adopts an MVC structure in Django, with views or templates for different data, controllers that determine the data surfaced for each view, and models to keep track of different types of analytics data.

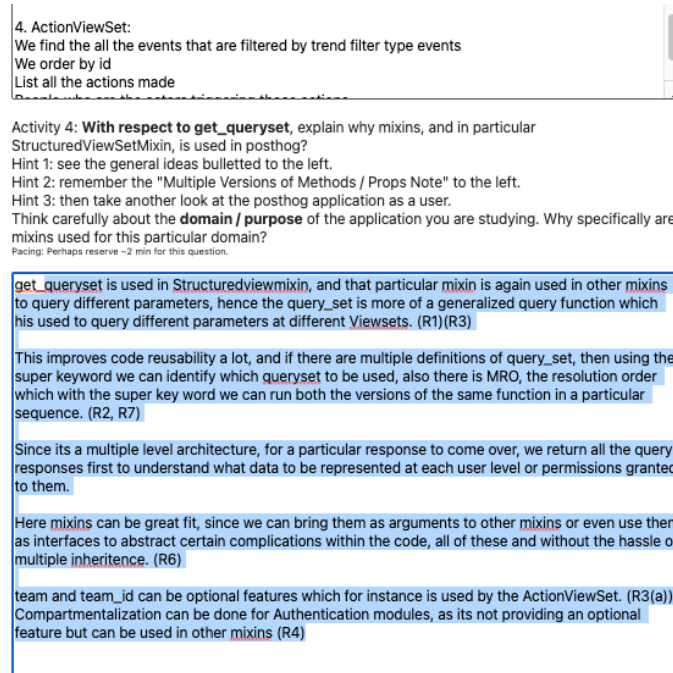


Figure 6.10: Here, we can see P2 mapping rationales from the theoretical and canonical contexts (see Figure 6.7) into the professional context.

developers tackle the otherwise overwhelming task of understanding how and why design patterns from theory apply in complex, real-world codebases.

6.2 RM System Implementation

To implement RM, we begin by parsing a professional Python codebase³ (i.e., Django or Flask⁴), using abstract syntax trees (AST) to identify syntaxes correlated with various design patterns. For instance, in the case of MVC, (a) the classes that extend the Django/Flask base model class would be considered the “models”; (b) URL registrations and associated templates would be considered

³We retrieved code from open-source projects on GitHub. While web source code is often minified, the rich, open-source professional examples on GitHub are not, making them easier to understand and more valuable for learning.

⁴djangoproject.com, flask.palletsprojects.com

the “view”; and (c) Django ViewSets⁵, which perform logic on model data and send refined data to the view, would be considered the “controllers”. Similarly, we flag classes titled with the suffix “Mixin” (e.g. *StructuredViewSetMixin*) as Mixins⁶ and also note which Mixin methods are adopted by other classes.

It should be noted that design pattern detection is non-trivial and an active research area. Techniques that both loosely inspire our approach and/or can be used to refine it include AST-based static detection, dynamic detection leveraging program state sequences, first order logic predicates, [141], UML-based approaches [142], meta-modeling patterns as compositions of simpler components [143], ontology-based semantic rules [144], structural/behavioral/semantic signals [145], machine learning featurization [146], code-related metrics [147], attention-based methods [148], similarity scoring, probabilistic approaches [149], and methods that leverage LLMs to produce embeddings which are then classified [150].

In our case, we extract pattern instances through AST parsing and pre-defined rules, and our system surfaces the patterns in a web view through auto-generated⁷ diagrams, presented alongside pattern definitions and canonical examples. Our system keeps track of the main components of each pattern. It identifies how those components are instantiated in a professional example and generates diagrams for the professional example that are analogous to hard-coded ones from theory or canonical examples. For instance, since our AST was able to detect (a) mixins, (b) their methods, (c) the classes that adopt said mixins, and (d) the methods/properties adopted from the mixin, our system can then create diagrams out of these four components (see Figure 6.4).

⁵django-rest-framework.org/api-guide/viewsets/

⁶It is the convention to have the suffix `mixin` after mixin classes in Python as per pythontutorial.net/python-oop/python-mixin/, coding-guidelines.lftechnology.com/docs/python/classes/, and ridwanray.medium.com/python-mixin-d6f83eaa6951. However, we can extend our system in the future to detect all possible mixins (i.e., classes inherited alongside other classes, classes adopted by views that engage in multiple inheritance).

⁷N.B. While our system is generally automated, one exception is that we wizarded a list of classes to exclude to limit the length of the activities.

Concretely, for each pattern we maintain a pattern schema that enumerates roles (e.g., mixin, adopter, adopted member) and relations among them. Our AST pass applies pattern-specific rules to extract candidate role instances from source files. We then instantiate a diagram template—derived from the canonical/theoretical diagram for the pattern—by filling in the placeholders. The layout itself is a simple text-based arrangement with unicode arrows.

In terms of limitations, heuristic, static AST detection of patterns is imperfect, with precision/recall bounded by the performance of best-in-class detectors described earlier, so we expect both false positives and negatives. Static parsing misses dynamic runtime features and our rules rely on language/framework conventions, so generalization may degrade. Moreover, the diagram is schematic (text+arrows) and does not encode full UML semantics. And for pedagogical reasons, we used a hard-coded exclusion list to keep activities tractable. In this work, our focus was on analogical mapping across diagrams and other pattern schema components, but future work can improve the accuracy of pattern detection and the UI/UX of the diagrams.

Our system automatically generates activities as well. We automatically customize each pattern-specific activity to the professional example at hand. For example, in Figure 6.7, the activities reference example-specific mixins and methods. The activities also build upon previous answers provided by the learners, as in Activity 3B, which refers to 3A [25], [26]. The activities spotlight the inner workings of a pattern by directing the learner’s attention to diagrams and specific components of the pattern (e.g., the functionality of Mixin methods and where they are adopted). These automatically customized activities intend to teach the learner why the pattern works by prompting them to recall the application’s use cases, consider its domain, and map pattern theory to professional practice.

6.3 Controlled Think-Aloud Evaluation of RM to Gauge Learning

To evaluate RM and its instantiation of PM, we followed the same 5v5 setup as SE and KM, recruiting, screening, spot-checking (no red flags), and assigning conditions similarly. The nature of the study, timings, breaks, learning orientation, scoring, interventions⁸, and phases of pre-test → tool use → post-test (same as pre) → survey were also identical. The studies measured the effectiveness of the PM affordances, such as RM’s analogical mapping tables, in helping learners understand how and why design patterns (namely Mixins and MV*, i.e., variants of MVC) apply in professional codebases (namely Posthog). We note some system specifics:

Screening: RM participants were expected to be intermediate-level engineers with experience in Python, Django, and/or Flask. They were a cohort who could write applications but may lack practical experience applying design patterns and architectures in real-world codebases. They were asked if they had heard of, seen, or used 28 major patterns (e.g., Mixins, Lazy Loading, Model–view–viewmodel) and to explain the ones they could to ensure they were prepared. Participants were included if they indicated familiarity with ~ 10 or more patterns, frontend tools (e.g., CDT), development environments (e.g., VS Code), internet concepts (e.g., HTTP protocols), backend tools (e.g., command line), and/or source control (e.g., git). Moreover, they should have written between $\sim 1K$ and $5K$ lines of Python code. We also checked that they had experience reading documentation and setting up codebases similar to Posthog’s on localhost.

Control Details and Training: We aimed to isolate the effects of PM while still providing the control a fair state-of-the-art learning experience. RM’s control group completed the same CDT training as SE and had access to the same CDT cheatsheet. During tool use, control users received the Posthog documentation and codebase running live on localhost, accessible via Zoom remote

⁸Interventions occurred to ensure experimental condition users of RM proceeded linearly through activities. They were also expected to map step-by-step from theory to practice, following reference links as they are helpful.

control⁹. They could use VS Code, CDT, and the command line on the codebase and the Posthog website. They had access to the same content as the RALE Modules tool, including definitions/schemas, mechanisms, code snippets, data flow, goals, and rationales. However, this information was not organized in a 3-column table-like format for easy analogical mapping between theory, tutorials, and professional examples. All content was presented linearly and without checkboxes. Instead of active reflection questions with fillable textboxes, participants were instructed to consider how and why the example works over alternatives, referencing relevant code and data flow. In summary, there was no process-management-like ordered activities, reflection prompts, tradeoff tables, or autogenerated diagrams for the professional example. This control setup ensured we tested the efficacy of process management for analogical mapping.

On the other hand, the experimental tool, RM, was generally self-explanatory and did not require training.

Pre/Post-test: All RM participants (both conditions) watched a video demonstrating the key functionalities of Posthog (PH), such as filtering (1) actions, (2) insights, (3) dashboards, and (4) groups (functionality available to all PH users) and tagging the first three (functionality available to enterprise PH users). In the pre-test, they diagrammed and described *how* design patterns like Mixins and MV* work and *why* they were chosen over alternatives in PostHog. During the pre and post-tests, participants could revisit the video of PH features if they wished but not the code of the example.

Scoring and IRR: To evaluate whether participants¹⁰ could understand how and why design patterns¹¹ apply in the professional code context, we scored their pre and post-tests according to

⁹In fact, both RM conditions used the tools on the experimenter's computer via Zoom remote control.

¹⁰The tests of one RM participant were coded only by the experimenter due to time/technical constraints, using established rubric standards.

¹¹In RM, potential concerns regarding cross-influence between mixin and MV* sections were mitigated by rubric clarity and high IRR.

the qualitative coding rubric described in Table 6.1. We followed the same IRR process as SE (see Chapter 4’s System Evaluation - ‘Scoring and IRR’ section for more details), including the same style of scoring, summing points, establishing the codebook, handling sparse codes, and accounting for case-by-case discussion, etc. Like with SE and KM, we report (a) percentage agreement where possible (83.6% ($\sigma = 12.6\%$) over 27/31 codes), (b) Cohen’s weighted kappa on the subset of codes (0.76 ($\sigma = 0.18$) over 26/31 codes) from (a) where the weighted kappa metric could be applied, and (c) the remaining codes that required case-by-case discussion on all data (4 for RM).

Additional Interventions: Beyond relevant interventions described in the SE study, we asked participants to recall pattern definitions or core Posthog features from the demo video (can be added to and enforced in the interface in the future), to pace themselves if they were taking more than a few minutes on a particular activity (a rule can check if they spend more than ~ 4 minutes on a particular segment), to specify domain-specific points when attempting the rationales question if they were not tailoring to the professional example (rule-based checks or classifiers could help in the future), to cover each component of the mixin/MV* architecture if they were attempting to move on without doing so (enforceable through AST rule-based checks), to reference the diagrams/code/goals and content across theory/canonical/professional domains in their answers if they were not doing so for questions that rely on that content (enforceable through classifiers), to limit their time on deep code reading to optimize for architectural understanding (automatable through timers), to mention specific classes or methods if they were moving on without doing so (enforceable through classifiers), and to note (if they weren’t already doing so) the definitions of adopter classes and adopted methods and their presence in the diagrams to prepare them for identifying each in the context of Posthog (mitigable with interface fixes and better labeling of diagrams).

Learning Outcomes
4 Items on mixing and matching mixins, covering subtopics like Method Resolution Order and Dependency Inversion.
3 Items on enterprise tagging logic in the TaggedItemViewSetMixin, covering subtopics and rationales like collaboration.
5 Items on scoped access logic in the StructuredViewSetMixin, covering subtopics and rationales like permissions and compartmentalization.
4 Items on optional features and composition afforded by mixins, covering subtopics and rationales like code reuse and sharing.
2 Items on mixin-driven super() calls, covering subtopics like multiple inheritance.
2 Items on the model-related aspects of the MV* architecture, including persistence.
2 Items on the controller-related aspects of the MV* architecture.
2 Items on the view-related aspects of the MV* architecture, including API endpoints.
3 Items on the modularity of the MV* architecture, including separation of concerns, code reuse, and maintainability.
3 Items on how MV* handles complexity well, including multi-stakeholder collaborations, loose coupling, and readability.
1 Item on how Posthog uses MVT (model-view-template).

Table 6.1: Table describing the rubric items across two design patterns (Mixins, MV*) that RM and control users were tested on. Our rubric items had specific details for each point and also awarded fractional points (0.25, 0.5, 0.75) for situations where learners partly understood aspects of our learning outcomes.

6.4 RM Evaluation Results: Process Management’s Role in Facilitating Transfer from Theory to Practice

We found that RM users achieved greater knowledge gains than control participants. RM users had an average gain of 18.6 ($\sigma=2.2$) out of 31, while control users had an average gain of 5.9 ($\sigma=3.3$) out of 31. Experimental participant data points are presented in Table 6.2 and Figure 6.11, while control participant data points can be found in Table 6.3 and 6.12.

Score	P2	P3	P5	P7	P9
pre-test	5.5	7	11	13.75	9
post-test	27.5	26.5	28.75	30	26.5
gains	22	19.5	17.75	16.25	17.5

Table 6.2: Table showing knowledge gains with pre/post-test scores (out of 31) for RM users.

Score	P1	P4	P6	P8	P10
pre-test	5.75	6.25	7.25	12.25	10.75
post-test	11.5	9.75	11	23.75	15.5
gains	5.75	3.5	3.75	11.5	4.75

Table 6.3: Table showing knowledge gains with pre/post-test scores (out of 31) for control users.

Quantitative Analysis: To analyze RM results, we leveraged a two-tailed Mann-Whitney U analysis identical to the one performed on SE and KM data. The Mann-Whitney U statistic was 0 ($p = 0.01$), indicating a significant difference between the groups. The effect size ($r = .79$, $rb = 1.0$) also demonstrated a substantial difference, and the sensitivity/robustness check yielded a simulated power > 0.8 . Although these results are encouraging and pass the same simulated power check as SE/KM, they stem from an exploratory 5v5 design; replication with larger, “in-the-wild” samples

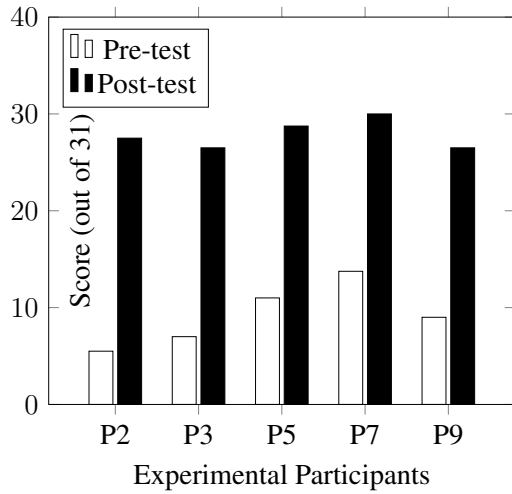


Figure 6.11: Comparison of pre-test and post-test scores for experimental (RM) users.

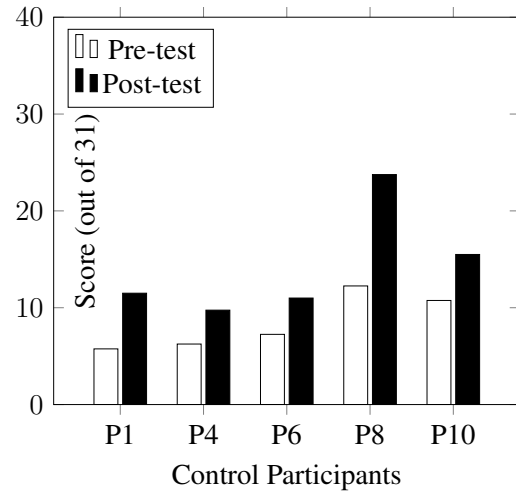


Figure 6.12: Comparison of pre-test and post-test scores for control users.

is still needed (as noted in the SE analysis).

Qualitative Mechanism Analysis: We found RM users cited the table of rationales as helpful for understanding why patterns apply in practice. All RM users went through all sections of the tool (definitions, goals, rationales, tradeoffs, etc.), checked checkboxes to confirm they understood concepts, and filled out the reflection questions.

Experimental users described RM's process management as helpful, with at least 4 out of 5 of them providing a feature rating of *extremely helpful* for each of the following features: **visual diagrams** (PM1), the **table structure for mapping** (PM7), data flow, **reflection questions** (PM8), and **comparable tradeoffs** (PM6). While there was no substantial difference between how the users rated the experimental ($\mu = 4.6, \sigma = 0.5$) and control ($\mu = 4.4, \sigma = 0.68$) tools, the participants in the experimental condition seemed to have engaged in sensemaking more effectively and learned much more about applying design patterns. The control participants focused mainly on reading the reference and finding code in VS Code/CDT. In contrast, the experimental participants actively checked off detailed rationales, mapping them from theory to professional practice. With

the data in mind, we can now answer our research questions on exploration and learning.

We now examine exploration (RQ1) and learning (RQ2) of RM and control participants.

Experimental Participants' Exploration (RQ1): Experimental participants effectively utilized the tool's features to deepen their understanding of design patterns in a professional context. They went sequentially through the activities as designed, **interactively checking off sections** (PM5) as they understood them. Still, their reflections provide nuanced perspectives. P2 found the rationales section particularly beneficial, stating that it *"first defines the theory and gives an understanding, then we have the code sample just next to it which gives a visual representation on how the theory is being applied, it's great as soon after reading the concept we dive into implementation."* This structured approach seemed to help P2 *"grasp ... concept[s] e2e [end-to-end]."* Similarly, P3 highlighted the comprehensive support provided by the tool, noting that *"all the sections helped; there was no section which was not helpful,"* specifically touting the *"trade off[s] section [and] diagrams."* The visual aids were crucial for P5, who remarked, *"Diagrams gave me more understanding to visualize each design pattern architecture,"* and appreciated how the tool *"helped me to know the level of my previous knowledge compared to what I understand now."* This suggests RM helped P5 build on their **prior knowledge** (PM2). P7 echoed P5's sentiments around RM's **visual depictions of salient information** (PM3, PM4), explaining that *"the diagrams feature provided an overview of the relationship between classes, creating a mental note and a visual representation of the knowledge and relationship between Mixins, Models, Views and Templates/controllers."* P7 also found the table-like structure instrumental, as it *"outlined the information in an easy and intuitive manner, giving clear paths of knowledge across the rows, starting with a theoretical perspective...and finally mapping the key points into the professional example."* The reflection questions appeared to further reinforce their learning, with P7 stating that they *"pinpointed to how the design patterns were being used, prompting one to re-read and review the diagrams,*

theory, detailed explanation examples and the professional examples,” which helped the concept “*sink in with each reflective question.*” Overall, RM users effectively leveraged diagrams, tables, data flow, reflection questions, and tradeoffs with the goal of understanding how patterns transfer to a new, professional application context [135].

Control Participants’ Exploration (RQ1): Control participants engaged with the tools by thoroughly reading the provided content on goals, patterns, and rationales, and by exploring code using VS Code and CDT. Still, they struggled to fully **transfer** design patterns to professional contexts. P1 noted, “*Yeah in each reference we got to know about the goals, features, rationales, classes, data which help us to understand the pattern easily,*” indicating an appreciation of individual elements but lacking reflection on more in-depth example-specific architectural transfer. Similarly, P4 stated, “*I used the reference, VS Code, CDT and Command Line... to read and gain a better understanding of Posthog.*”. Yet, they did not effectively align specific code structures with Posthog example specifics, failing to specify key mixins and the fact that they do not stand alone. P6 found the documentation helpful for understanding Mixins but admitted, “*I had trouble understanding how the front end was implemented even when I tried using CDT since I could not locate where the files being executed were,*” highlighting how even state-of-the-art tools pose potential obstacles when it comes to mapping theoretical concepts to practical implementations in a **complex** full-stack context. P8 mentioned that their understanding of Mixins and MV* patterns improved, saying, “*My understanding of Mixins and MV* design patterns greatly improved with the use of this tool.*” Still, they had limited success applying the pattern nuances within the professional context, missing intricate concepts around Posthog’s mixin-related method resolution order and sub-frameworks within MV*, such as MVT. They noted, “*I used CDT to explore a real-world professional project... CDT helped me in exploring code base by checking the application layouts and also codebase file structure...*” and reflected on similar behaviors “*also through using*

VS Code.” Still, like the other control participants, they diligently engaged with the materials and tools but often remained at the **surface-level**, reading code and documentation without effectively synthesizing how and why underlying design patterns function within the professional context.

Experimental Participants’ Learning (RQ2): In the pre-test, the participants had a limited understanding of how design patterns like mixins and MV* architecture apply in professional contexts. P2 focused on general components, stating, “*Components of PostHog: 1. Backend System (Python) 2. Frontend System (Angular or React) 3. Events System (Event-based Architecture) - Python 4. Visualization (D3.js)*,” without mentioning the nuances of how mixins allow slices of functionality to be adopted as needed. P3 admitted not using mixins in practice, saying “*I have no clue ... [about] mixins ... but, CRUD for Insights, Events and Action, Dashboard, Access rights.*” P5 identified some patterns but lacked depth, noting, “*User access management pattern via MVC... It can use a restful API design pattern. Adapter pattern.*” In general, participants, even when primed about mixins and MV*, did not have a detailed conception of why such patterns are core to the professional context, spending most of their time overfitting to patterns they were more familiar with applying.

By the post-test (see Table 6.2 for participant-specific data), RM users showed marked gains, grasping pattern mechanisms and rationales in context (reference Figure 6.2 for pattern definitions/examples). P2 said “*StructuredViewSetMixin (SVSM) and TaggedItemViewSetMixin (TIVSM)...have methods that can be implemented or reused in... DashboardViewSet, InsightViewSet, etc...hence designing an inheritance class pattern is highly difficult as it will inherit all the methods directly, of which some might seem unnecessary*” highlighting how SVSM (for filtering) and TIVSM (for tagging) are mixed and matched as needed by adopter classes such as Insights, Actions, and Dashboards. They emphasized compartmentalization, saying mixins are “*independent and form separate modules,*”, providing Posthog “*flexibility*” and “*reusability*”. They also shared how the

model-touching layer persists Insight, Action, and Dashboard data to the database, while the view-touching layer passes the data to the front end. They also understood that controllers perform business logic for filtering and tagging. While P2 had a high-level idea of MV* before (Figure 6.15), this Posthog-specific description and MV*-mixin coordination is unique to their post-test (Figure 6.16). Similarly, P3 made tradeoffs stating that *“using mixins gives freedom to adopter classes to implement only methods needed by the class unlike in Inheritance which enforces parent-child relationship”*. They noted mixin intricacies impossible for inheritance to achieve, saying, *“One other unique feature of mixin is the adopter class can call one or both the parent methods in sequence without overriding. Like the get_queryset method in TIVSM and SVSM. Both the implementations of TIVSM and SVSM get called in sequence by its adopter class.”* P5 also emphasized domain specifics in that *“StructuredViewSetMixin filters objects data of logged-in PostHog and queries their access based on team and organization,”* while *“TaggedItemViewSetMixin adds tags based on licensed permission”* and demonstrated how these mixins facilitate modularity and code reuse as they are mixed and matched as needed, as shown in their pre (Figure 6.13) and post (Figure 6.14) tests. P7 echoed this understanding, stating that *“Mixins are used in PostHog to add additional functionalities such as structuring of data and tagging”* and that the adopter classes *“act as controllers to the data models provisioning data to the templates depending on the user input or the user request.”* This progression is noteworthy because they could explain how and why two design patterns interact in the professional code, seeing how mixins are mixed and matched to support viewset adopter classes, which in turn serve as controllers in the MV*. They concluded that employing mixins and MV* architecture promotes code testability, maintainability, and flexibility, saying that using MV* *“in a large complex codebase like Posthog allows for collaboration between frontend developers and backend developers”*. Overall, the participants moved from a limited understanding to a nuanced appreciation of how and why mixins like SVSM and TIVSM

are strategically employed alongside the MV* architecture to manage complexity and enhance collaboration on enterprise-grade or production-quality software.



Figure 6.13: Experimental P5's pre-test diagram of their understanding of Posthog's mixins.

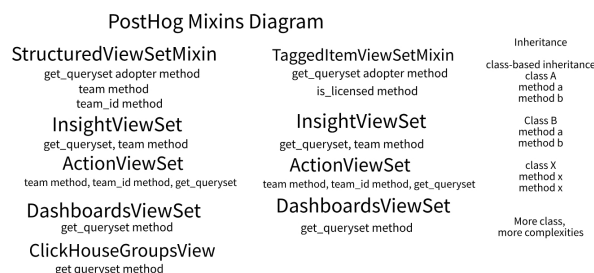


Figure 6.14: Experimental P5's post-test diagram of their understanding of Posthog's mixins.

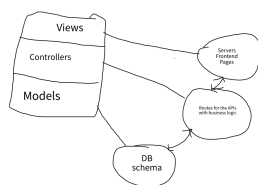


Figure 6.15: Experimental P2's pre-test diagram of their understanding of Posthog's MVT architecture.

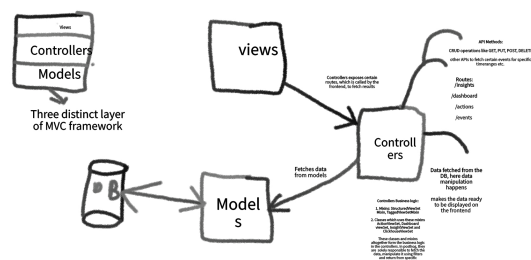


Figure 6.16: Experimental P2's post-test diagram of their understanding of Posthog's MVT architecture.

Control Participants' Learning (RQ2): Even by the post-test, the control participants' (see Table 6.3 for their data) understanding of how mixins and MV* patterns apply in a professional context remained limited. P1 acknowledged the use of mixins and MVC but provided a somewhat superficial explanation, stating, *"Mixins are the blueprint of objects; here things are optional. As we create methods and properties which can adopt the functionality according to our requirements,"* without naming the important mixins of Posthog and precisely defining their role (as in Figure 6.17 for mixins and Figure 6.19 for MV*). Even by the post-test, they failed to directly state

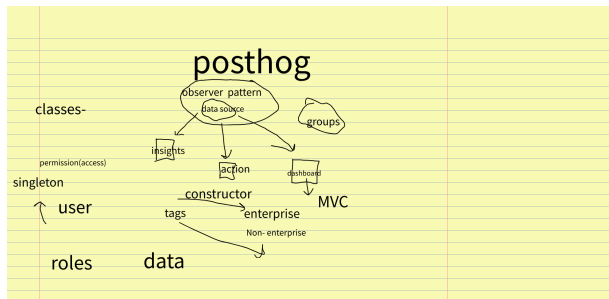


Figure 6.17: Control P1’s pre-test diagram of their understanding of Posthog’s mixins.

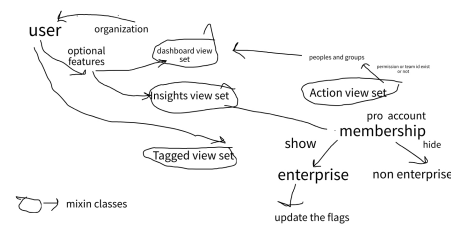


Figure 6.18: Control P1’s post-test diagram of their understanding of Posthog’s mixins.

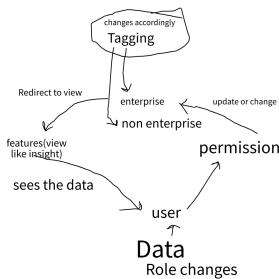


Figure 6.19: Control P1’s pre-test diagram of their understanding of Posthog’s MVT architecture.

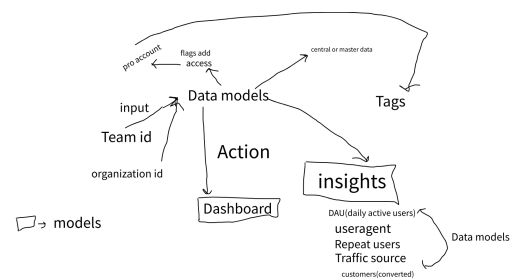


Figure 6.20: Control P1’s post-test diagram of their understanding of Posthog’s MVT architecture.

that the complexity of the Posthog codebase necessitates using MV*, which modularizes and structures code (Figure 6.20). Their level of understanding did not change much from the pre-test for mixins either; they failed to clearly delineate and link adopter and adopted classes (Figure 6.18). P4 mentioned various design patterns but misattributed functionalities, explaining that “*The Authentication Mixin was used to manage authentication-related functionality like session management of users, login and logout... The Routing mixin was employed to manage client-side routing within the application,*”, indicating a misunderstanding of the relevant mixins in PostHog for the functionality under investigation. P6 attempted to describe mixin use, even discussing tagging and filtering, but focused on general code reuse and did not demonstrate **awareness** of mixins’ unique competitive advantage over inheritance (i.e., selectively included functionality slices). Instead, they noted that

“a method was defined in a Mixin / Class that performs queries from the database... This method was then modified and used in two classes, which returned the same data but modified in different ways.” However, they did not grasp several of the specific mechanisms or rationales behind mixin use, mistakenly attributing tagging to `InsightViewSet` rather than `TaggedItemViewSetMixin` and possibly assuming `StructuredViewSetMixin` inherited another mixin. They also failed to describe how both mixins’ versions of the `get_queryset` method run sequentially to enable both filtering and tagging.

P8, our best-performing control user, provided a more detailed description but still occasionally lacked precision on mixins vs. adopter classes, explaining that *“Posthog has 3 mixin classes that contain methods for controlling the essential functionalities...”* and listing components without fully connecting them to the broader architectural sequence. They recognized that mixins provide reusable and selective inclusion logic for tagging and filtering, and even demonstrated an understanding of mixins’ modularity principles often associated with dependency inversion (DI)¹², but they possibly confused the filtering role of the `StructuredViewSetMixin` with the `InsightViewSet`, incorrectly treating the latter as a mixin in some parts of their answer. Moreover, they failed to see the strategic importance of filtering and tagging sequentially¹³, an insight that several RM users reached.

Overall, even after the post-test, the control participants struggled to fully comprehend how and why design patterns like mixins and MV* architectures apply in a professional setting.

¹²We acknowledge that while the TIVSM and SVSM Mixins promote code reuse and modularity, they do not strictly adhere to DI principles. For clarity, we use DI loosely here and in this study. In future work, we will apply the DI label more conservatively, restricting it to cases where dependencies are injected via interfaces—such as if a tagging or filtering service were passed into the constructor of adopter classes.

¹³This sequencing is achieved via `super()` calls and the method resolution order (MRO) in Python, which allows both mixins’ `get_queryset()` methods to run one after another.

6.5 RM Discussion - Practical Learning and Technical Contributions

RM users outperformed control users in understanding how and why design patterns were used in professional code. By systematically mapping theory, tutorials, and professional examples, RALE Modules bridged the gap between the non-experts' prior or theoretical knowledge and expert practice. Our approach improved participants' reflection on domain-specific rationales, linking them with code-level details and tradeoffs.

By automatically detecting design pattern components in code and diagramming them for the user, RM alleviates a longstanding challenge in software engineering education: non-experts' difficulty in bridging conceptual knowledge with distributed, real-world code. In contrast to existing code exploration and debugging tools (e.g., CDT, Whyline, code foraging systems [23], [24], [49], [51], [52], [53]) that focus primarily on low-level or local inspection, RM systematically surfaces higher-level architectures, design goals, and tradeoffs across entire codebases, but in a practical rather than textbook manner [5], [7], [123]. Consequently, our approach addresses documented gaps in sensemaking support for intermediate learners (e.g., why Mixins are useful in a particular professional example as opposed to in general), illustrating how tailored scaffolding, guided reflection, and domain-specific rationale mapping can promote professional design pattern expertise.

Our key results demonstrate that experimental participants systematically leveraged RM's analogous tables, diagrams, reflection prompts, and tradeoff exercises to achieve a deeper understanding of patterns in professional code compared to controls. Study logs revealed that RM's structured approach helped participants overcome code complexity (e.g., the large Posthog codebase) by grounding each design pattern in intuitive "starter" diagrams and then guiding learners through analogous diagrams in the tutorial and professional example domains. In interviews, RM users identified how multiple classes coordinated around a single pattern (e.g., chaining mixins or

coordinating controllers), whereas the control group often became “stuck” in files without deeply linking their relationship to other modules (e.g., analyzing files and lines of code rather than latent cross-file patterns).

Participants in RM condition highlighted the value of the trade-offs exercises, which had them compare alternate implementations (e.g., Mixins vs. inheritance) spurring deeper reflection. These observations align with the broader theme of scaffolding sensemaking that emerged in SE and KM, suggesting that systematically mapping nuanced yet abstract ideas to real code and prompting reflection on rationale fosters more transferable, pattern-level understanding. Shared among all three PM tools is the deliberate orchestration of learning steps—lowering cognitive load, evoking connections between concepts and code, and encouraging domain-aware reflection—that appears critical for bridging novices’ theoretical knowledge with real-world source code complexity.

By systematically embedding these steps—outlined in the PM framework—RM ensures that learners move beyond local inspection and gradually integrate higher-level, design-conscious thinking into their exploration of professional code. In this way, RM aligns with SE and KM’s core idea of *managing the sensemaking process* to help learners overcome code complexity, surface-level focus, transfer challenges, and limited awareness, producing deeper understanding of professional-scale design patterns.

As participants sequentially checked off each section and mapped theory to real-world examples, they harnessed cognitive scaffolding (e.g., reducing extraneous load) and analogical transfer (e.g., bridging abstract schemas such as “mixins” to practical code in Posthog) in ways consistent with research on mental model development, structured reflection, and metacognition in program comprehension [26], [34], [36], [40], [74], [83], [99]. Such findings align with literature emphasizing how guided mapping and reflection foster a deeper conceptual grasp of software architectures and their domain-specific rationales.

Through RM, we see promise in using process management to guide developers as they build mental models of complex code, especially at production scale. RM shows how PM heuristics, such as mapping patterns across multiple contexts, prompting reflective reasoning, and comparing tradeoffs, effectively contribute to learners' conceptual understanding. Like with SE and KM, we see our approach helping reduce the perceived difficulty of applying professional patterns by systematically supporting sensemaking and transfer.

CHAPTER 7

DISCUSSION: CORE CONTRIBUTIONS AND PRINCIPLES FOR DESIGNING PM TOOLS

This chapter synthesizes the core contributions of our Process Management (PM) framework, highlighting key findings and principles from our studies. We reflect on how our conceptual and technical approaches systematically addressed common obstacles, discuss generalizability and transferability of our scaffolding techniques, and outline practical guidance for future researchers and tool designers.

Our controlled evaluations of SE, KM, and RM demonstrate that process management can systematically enhance learners' ability to explore professional codebases, organize knowledge, and transfer conceptual understanding to real-world settings. Our framework integrated the non-expert's intent to learn professional techniques and shepherded them along intuitive sensemaking pathways. Across varied learning outcome structures that non-experts easily miss—JavaScript variable transformations, granular CSS techniques, and Python design patterns—PM scaffolds supported learners in reaching learning outcomes and overcoming common obstacles, including code complexity, surface-level focus, lack of transfer, and awareness gaps. By structuring exploration [26], enabling active engagement, and promoting reflection [18], [36], PM tools provided learners with a systematic starting point and process for structured professional development.

7.1 Addressing Code Complexity through an Ordered Sensemaking Approach

Learners attempting to make sense of professional code often struggle with where to start, what to focus on, and how to organize their understanding. Research highlights that novices tend to

fixate on *bottom-up* exploration, *low-level* code components, or debugging steps (e.g., in CDT) rather than identifying deeper patterns across codebases [38], [39]. In reality, learners should also leverage *top-down* program comprehension strategies where *high-level* abstractions guide the understanding of *low-level* details [21], [37], [38]. While feature location tools like FireCrystal, Unravel, and Telescope help in linking UI elements to underlying code [54], [55], [56], our work goes further by providing learners a *high-level* process for uncovering deeper architectures (e.g., *data flow, mixins*) and inter-relationships (e.g., *one variable transforms into another, media queries qualify if flex or grid are used at different page widths*). Such process management mitigates the novice tendency towards the immediately visible, narrow, and specific, and scaffolds the “hidden labor” required to reorganize knowledge structures or approaches around high-level goals or concepts (e.g., *overall data flow governs specific value transformations, apparent finesse stems from visual nuances which stem from specific CSS intricacies, architectures dictate code-level implementations*) [43].

Our Studies Confirm the Challenges: Control participants in all three studies primarily focused on events, syntax, or reading line by line, often struggling to integrate their findings into a coherent picture. For instance, in SE, control participants fixated on event-based interactions rather than tracing intermediary variable transformations deeper in the call stack. They used debugging tools to navigate function calls but struggled to see broader data flow. Similarly, in KM, control users over-relied on syntaxes shown in CDT and engaged somewhat superficially, mechanically copying styles from inspected elements and fixating on CSS overriding. They rarely contrasted visuals and discovered fewer visual and code nuances. RM control users, too, read code and documentation diligently, but were less effective than their process-managed counterparts at mapping high-level theory to practice.

PM Structures Sensemaking to Overcome Complexity: In contrast to the controls, PM tools

scaffold exploration and guide learners through real-world examples. Their tendency to **reduce extraneous cognitive load** (PM4) [25], [26] is seen in SE’s chunked exercise progression, KM’s steps and cycles, and RM’s analogical mapping tables. The PM tools seem to break through complexity by leveraging visual intuition, linking prior knowledge, and surfacing important information.

Starting with Visual Intuition to Overcome Complexity (PM1): Each system aimed to ground users in intuitive visual representations before diving into code details. Users explicitly cited (a) the “visuals first” exercise ordering in SE, (b) visual groupings in KM, and (c) diagrams in RM, as key to their successful performance.

Linking Prior Knowledge to Overcome Complexity (PM2): After initial activities, PM tools structure exercises to connect new information with what users already know. For instance, SE users referenced their resurfaced notes, KM users linked CSS code to a map of differentiated visual features they developed earlier, and RM users built upon previously studied theory when investigating professional code.

Chunking, Surfacing Seminal Parts of Code or Visuals to Overcome Complexity (PM3): PM tools also guided learners to focus on the most critical sections of the codebase to prevent getting lost in extraneous details and cognitive overload. SE users highlighted and tracked live variable updates, KM users viewed and mapped only visually consequential CSS, and RM users only studied code relevant to high-level architecture diagrams and design pattern theory. Such scoping may have helped participants immediately see how and why key concepts manifest.

By harnessing such PM heuristics, our tools appeared to help users progressively build and integrate knowledge without becoming overwhelmed by large, interconnected codebases. Historically, tools like Unravel and Telescope [55], [56] focused on feature location but lacked systematic scaffolding for deeper comprehension. PM addresses this explicitly by providing structured sense-making scaffolds, reflecting a shift toward deeper and more systematic cognitive support.

7.2 Mitigating Surface-Level Exploration Biases through an Interactive Comparing and Linking Approach

Beyond order and structure, a major challenge for learners stems from over-reliance on superficial signals (e.g., shared syntax attributes) or familiar exploration pathways (e.g., following code in execution order straight into confusing function calls, line-by-line code reading). For instance, our control users gravitated toward JS event-based interactions in SE, overfitted to flexbox in KM, and defaulted to generic code reuse as the main reason for using mixins in RM—failing to explore deeper mechanisms, rationales, or tradeoffs. In this way, tools without process management can reinforce user exploration biases, leading to narrow and reactive sensemaking rather than goal-directed inquiry [23], [24]. While earlier tools such as Isopleth [14] and Ply [13] helped learners uncover relationships and code dependencies, fewer tools have directly addressed the class of problems centered around process deficiencies that keep novices at the “surface level”.

Interaction and Comparative Analysis to Overcome Surface-level Focus (PM5, PM6): PM tools appear to prevent such fixation on the superficial through **comparing** (PM6) code artifacts or cases, a known driver of nuanced awareness and knowledge transfer [29]. More so than control users, PM tool users **interacted** (PM5) through active comparison to advance their mental models around how code components coordinate or why specific techniques apply in given scenarios.

We saw how comparison drove conceptual learning outcomes in each of our three systems. SE participants compared variable value changes and prior notes to discover internal variable-to-variable relationships that eventually affect the DOM. Though not the same as comparing different examples, this may have helped them integrate their understanding of different but related variables present across multiple exercises. KM users systematically compared and contrasted examples to identify differences and tinkered and toggled properties to discover underlying CSS techniques.

Finally, RM users compared alternative patterns to realize the impact of design pattern choice in specific domains.

Interestingly, both conditions' behaviors align with Situated Action Theory [151], where the immediate situation—the tool's interface, feedback, and constraints—shapes the learners' actions in real time, including the form of their inquiry (e.g., code reading vs comparing or mapping). Without explicit process scaffolds, learners default to behaviors naturally afforded by their tools, missing deeper insights. To prevent ineffective sensemaking, PM tool design actively structures learners' interactions with the code and aims to shape program comprehension pathways.

Throughout, linking **prior knowledge** (PM2) played an instrumental role in mitigating surface-level exploration. By surfacing prior observations (e.g., intuitive visuals, layout differences, theoretical diagrams) and connecting them with deeper insights (e.g., deep variable value transformations, underlying CSS concepts, custom diagrams of pattern enactment in real-world code), PM tools helped learners connect new information with existing knowledge, preventing them from stopping at surface-level observations of disparate codebase artifacts.

7.3 Facilitating Transfer and Conceptual Awareness through a Mapping Approach

Even if learners understand examples deeply, transfer is still one of the most difficult challenges they face when learning complex concepts. While they may memorize syntactic constructs or canonical examples, they often struggle to generalize concepts across different contexts [34], [121]. They fail to see a concept's diverse manifestations or align lower and higher-order relational structures between base and target domains to facilitate analogical transfer. This limitation affects software education, where learners must grasp not only the mechanisms of specific implementations but also their rationale—understanding when and why to apply particular techniques based on domain-specific constraints [7], [25]. In reality, most learners grasp neither, e.g., getting lost in

CDT's Debugger or Network tabs or VS Code files.

Scaffolding Solutions for Transfer and Awareness: PM tools bridge this gap by scaffolding explicit transfer mechanisms, helping learners move from recognizing techniques to rationalizing their selection and application. While SE has PM affordances, such as **reflection** (PM8), that helped with architectural awareness and could help with transfer, the study focused on example-specific retention and learning but not transfer. On the other hand, KM's overall study and RM's tool use phase centered around transfer. While control users in the KM and RM studies struggled to apply learned techniques or patterns to new professional contexts, PM tool users leveraged strategic **mapping** (PM7) and reflection affordances to succeed (e.g., mapping visuals to code, linking theory with practice). KM users could then recognize use cases for granular CSS techniques in new settings. Similarly, RM users could connect multiple coordinating components and appeared to perform analogical transfer of design pattern theory into professional practice.

Among the three PM systems, KM provided the most substantial basis for transfer claims, utilizing three distinct pre/post-test examples, different from the six examples used during tool interactions. Reflections in KM contribute to a knowledge map, which functions as a cognitive artifact of an evolving user mental model, and a potential method of offloading cognition [152]. The map supports iterative externalization and organization—activities critical for transfer [25]. In summary, while each system varied in the extent of its support for participant knowledge transfer, together they provide evidence suggesting that process-managed learning can support transfer.

7.4 Principles and Practical Guidance for Designing PM Tools to Facilitate Human-Driven Goals

Developing these tools deepened my appreciation for how explicitly structuring interactions can significantly influence the depth and quality of professional learning. Over the years, while inter-

acting with hundreds of developers, I faced the challenge of balancing detailed scaffolding with maintaining learner autonomy, an experience that potentially reflects real-world tensions computing educators and tool designers encounter when empowering novices to become experts.

To enable this transition, learning experience designers and researchers would benefit from empathizing with the plight of learners. As we have observed, learners want to understand the code they are working with, but they often struggle to do so. They want to explore, tinker, and learn, but they often get lost in the complexity of the code. They want to transfer their knowledge to new contexts, but they often become stuck in surface-level or unimportant details. We posit that this is a result of the absence of structured sensemaking pathways that can help them achieve their goals.

By shaping sensemaking, promoting interactive comparisons, and scaffolding transfer, PM strives to enhance learners' ability to construct and apply knowledge. Our process-focused framework is flexible in that different tools can instantiate PM across different domains and knowledge structures, and yet potentially yield similar learning outcomes. The framework helps intermediate-level learners navigate complexity, reflect on rationale, and transfer conceptual knowledge into production-grade environments—core needs in the professionalization of computing education. These findings help position our contribution to the research literature on program comprehension and scaffolding. While PM is grounded in known disciplinary best practices, unlike existing work, the PM tools and studies show the framework can support professional development use cases and real-world code exploration. Hence, researchers and designers can leverage the PM framework to create learning tools that utilize professional code to foster expertise development across diverse domains. In general, our process management insights could be uniquely valuable to software engineering educators given our analyses of related work, which suggest that:

- The internet is rife with **professional code that educational tools rarely harness** for peda-

gological purposes.

- While introductory CS education and expert program comprehension are rich literatures with diverse tooling, scaffolding tools that guide the intermediate-to-expert transition in software engineering are underresearched relative to the **economic and social impact they can have on diverse self-directed learners** [153].
- Although best practices in scaffolding abound, until now, we have had relatively little evidence of how those practices can help intermediate developers overcome unique struggles and become experts through the authentic study of the web’s immense supply of professional source code.

As I reflect, throughout developing these PM tools, I witnessed how structured scaffolding reshaped learner interactions—transforming initial confusion into clear conceptual articulation. Concurrently, the rapid evolution of AI coding assistants (e.g., GitHub Copilot, Cursor) underscores the timeliness of our work, as industry increasingly demands deeper architectural comprehension and adaptability that probabilistic tools alone may not fulfill. Design researchers in computing education may benefit from foregrounding conceptual learning by using our PM framework across diverse use cases, as this will help them equip the next generation of software engineers with the high-level architectural skills needed to oversee autonomous agents and complex codebases.

7.5 Ongoing Challenges, Limitations, Methodologies, Ethics, and Future Opportunities

Despite the promising results, we must note some limitations¹.

Sample Size and Generalizability: The three system evaluations involved sample sizes of 10 participants per tool (albeit ~ 300 hours over 30 sessions), limiting our generalizability claims for

¹N.B. Use of AI Assistance: The first author used AI-based writing tools (ChatGPT, Grammarly, and Claude) for drafting and editing across the whole manuscript, ensuring all content was reviewed, fact-checked, and aligned with research objectives, with intellectual contributions and conclusions remaining entirely their own.

both the individual PM systems and the broader PM framework. Still, the large effect sizes and statistical significance suggest promising learning gains, offering a foundation for future studies with larger sample sizes. Since deep conceptual learning requires extended engagement, scaling studies beyond controlled lab settings will be most feasible through in-the-wild deployments of these systems. Additionally, as SE, KM, and RM all have automated exercise generators, SE and RM can support broader example coverage (as KM did), and future work should explore their applicability across diverse real-world codebases. Future PM efforts can also leverage AI to generalize the exercise generators so they can be ported to new languages and frameworks easily. If PM can help novices learn the concepts across diverse programming languages and architectures, the challenge of far transfer—having developers apply techniques to their own applications—can then be approached ².

Tool-use Time: Another consideration is the time commitment required for PM tool use. Participants in both conditions could take as long as they wished on any study section, leading to total study durations ranging from ~5 to ~12h for SE (extremes: 4:10h and 13h), ~6 to ~19h for KM, and ~7 to ~18h for RM. Within the tool use phase, given the structured learning activities of the tools, process-managed users naturally engaged for longer average durations than control participants (~6 vs. ~1.5h for SE, ~8 vs ~3h for KM, ~9 vs. ~4h for RM). This reflects the systems' intentional scaffolding design—encouraging deep sensemaking—rather than an uncontrolled confound. Control participants were explicitly encouraged to explore their tools thoroughly (e.g., up to 3 reminders) but self-reported reaching saturation more quickly and requested to move forward. In contrast, PM tool users had a guided process that promoted deliberate engagement and continued

²Another minor issue that came up during the RM experiment is that the code snippet link of the tutorial example (ToDoMVC) for the MV* pattern broke so we had to inform the candidates of the commit hash to which to switch to get the correct code link. While we told the participants, some chose to view the code, and some skipped it or, unfortunately, did not replace the commit hash or ask about it. However, since most of the tutorial example content was in the experimental tool/control reference, we believe this is not a risk to our study.

discovery. Imposing a time limit on process-managed users would have disrupted the self-paced process, while forcing control users to explore beyond their natural stopping points would have introduced artificial engagement, reducing ecological validity. Allowing both groups to engage at their own pace facilitated a fair and authentic comparison. As both conditions were recruited identically and could take as much time as they needed, time-on-task alone is unlikely to explain the differential learning gains. Still, even with our PM supports, the significant time commitment required to learn professional techniques seems to be an ongoing challenge, and future work can explore optimizing engagement durations while preserving learning depth.

Beyond time management, maintaining learner motivation throughout extended PM sessions presents additional challenges. Future PM implementations could benefit from explicit progress-tracking mechanisms and scaffolding learners with simpler examples before introducing complex professional codebases, to build confidence and reduce cognitive overload.

5 vs. 5 Tool Evaluation Justification: Given the nature of our ~10-15+ hour study sessions, it was only feasible to conduct a 5 vs. 5 design across three systems while maintaining high-quality data collection, participant engagement, and methodological rigor. This study serves as an early-stage investigation into structured, process-managed learning tools, prioritizing depth of analysis over broad generalizability. The theoretical justification for this sample size aligns with case study and microgenetic-style methodologies, which emphasize detailed process analysis over large-scale generalization. While we acknowledge more extensive studies can achieve greater external validity, our mixed-methods approach—combining quantitative assessments with detailed qualitative insights—allows for rich, meaningful interpretations of learning processes. Future work can explore scalability and generalizability across larger participant pools.

Microgenetic-style Think-aloud Back-and-forth: One other potential limitation of these system evaluations is that talk/think-aloud interventions and back-and-forth exchanges with partici-

pants during tool use and the pre/post-tests may have influenced results. While these interactions may have shaped how participants engaged, they were necessary to elicit rich insights, capture authentic processes in real time, and measure their full potential. This decision aligns with the microgenetic method [154], which focuses on analyzing learning processes as they unfold and deep-diving into the knowledge in a participant’s head. Given our mixed-methods approach, which integrates qualitative process data with controlled evaluations, these interventions were an intentional and essential part of our study design rather than a confound. Moreover, we standardized each intervention type so that systems such as LLMs can potentially automate some of them.

Our mixed-methods approach explicitly leverages both quantitative pre/post-test analyses and qualitative microgenetic-inspired interactions, grounded in established scaffolding and situated-action theoretical frameworks, providing robust triangulation of learning outcomes.

Potential Biases and Methodological Transparency: Since the experimental condition post-test answers were much richer and more detailed, our follow-up prompts asking them to elaborate—though standard and intended to be applied similarly to control participants—may have elicited more extensive responses simply because there was more substantive content to expand upon. A similar dynamic applied in tool use—the experimental tools involved more structured, process-driven interactions, naturally leading to more involved follow-ups on whether participants had engaged fully with key features and steps. The effect of follow-ups on elicited information is a known limitation of qualitative interviews [155]. Still, we acknowledge potential positive feedback loops and prompting biases, as our familiarity with participants’ tool use could have subtly influenced the specificity of follow-up questions (e.g., bias in why/how/what-style follow-ups, follow-ups on things they said so far, etc.). While we aimed for fair, neutral scoring and transparent methodology, knowledge of participants’ tool exposure may have influenced terminology choices made in the context of the discussion at hand. However, across the board, we followed a

rigorous protocol with standardized templates for prompting both sides similarly, and ultimately, participants could only articulate the knowledge they had, meaning their responses likely reflected their actual understanding rather than the experimenter's influence.

Training Differences: In RM and SE, the control groups received CDT walkthroughs before rather than after the pre-test (unlike KM). This difference presents a minor limitation but is unlikely to threaten validity, as CDT training on simplified demo examples would be unlikely to affect the pre/post-test, where participants were not even permitted to view code or use any tools.

Talk/Think-aloud vs. Written Thinking: While participants were generally instructed to engage in verbal think-aloud, some wrote down their thoughts instead of speaking them aloud. This is understandable, given the active writing required by the pre/post-test forms and some tool interactions. We consider this a minor methodological variation, as both approaches externalize cognitive processes and provide insights into participant reasoning. Because our goal was to capture real-time learning and decision-making, we treated written externalizations as functionally equivalent to verbal think-aloud responses. Though acknowledged, this distinction is unlikely to impact the findings.

Perceived vs. Measured Learning: While experimental users demonstrated significantly greater learning outcomes, tool ratings were similar across conditions. One potential explanation for this discrepancy is that participants may evaluate tools based on UX fluidity rather than their impact on learning. It is also possible that learners may not always recognize the extent of their own knowledge gains, leading to ratings that do not fully reflect how much they learned. Further surveying may be required to confirm these conjectures.

Diversity: Beyond expertise cultivation, though not our primary contribution, our participant diversity (63% Black, $\geq 89\%$ POC) is notable and resonates with shifting global population trends and potential socioeconomic impact. Future work may engage in further analyses of demographic

factors and the extent to which automated process management can scale expertise development to those without access to expert mentors.

7.6 Technical Achievements and Gaps

Through our work, we systematically transform professional codebases into structured processes and learning experiences, yet several technical challenges remain open research problems that we do not yet know how to solve. While we provide suggested directions for exploration, we emphasize that these represent areas for future investigation rather than solved contributions.

7.6.1 Design Pattern Detection as an Open Research Problem

Our RM system implementation employed AST analysis and rule-based approaches for design pattern detection, but this represents only one approach to a fundamentally challenging research problem. Even state-of-the-art benchmarks from the literature, including recent LLM-based approaches [150], do not achieve 100% accuracy. The problem’s complexity stems from several factors: patterns depend on both static (code-level) and dynamic (runtime) context [141]; they require semantic interpretation of class, method, and variable roles that intelligent systems model probabilistically [148], [149]; and they operate across codebase components, increasing their complexity. Practical considerations compound these challenges, particularly for dynamic detection around setting up environments and containers from READMEs and wikis, authenticating users programmatically, and deploying AI testing agents to reproduce runtimes [156], [157]. Even with complete data, conceptual challenges remain around whether patterns can be unambiguously determined, potentially requiring best-of-N experimentation [158] to test for convergence. While determining what constitutes an interesting and relevant pattern for today presents another challenge, LLMs and trend crawling may help address this partially. However, once patterns are detected,

since there are a finite number of possible patterns, it becomes a matter of filling diagram and activity templates—a process our system automates.

7.6.2 Automation Boundaries: What Can and Cannot Be Automated

Future work can explore how some aspects of our systems can be automated. Experimenter interventions, which our studies did not use AI for but could, represent a potential automation target. Similarly, participant screening and selection processes could be automated to identify learners at appropriate stages. However, good example selection remains a separate, unsolved research problem. For example, we selected XKCD for SE because of its semantically understandable code and intricate data flow, while we chose Posthog for RM because its mixin use was evident in frontend interactions for filtering and tagging; moreover, Posthog is a widely used and enterprise-grade library. To identify such examples, we could explore approaches using GitHub stars, forks, domain authority, or LLM-based assessment using rubrics [128], but we do not yet know whether this will work for automatic identification of high-quality examples. The challenge involves determining what makes an example suitable—whether it has clearly demonstrable features that can be traced to pattern mechanisms, requires crowdsourced insight since AI agents for UI manipulation may not perfectly map interactions to code (or even be able to reliably sign in to applications) [156], [157], [159], [160], [161], or needs semantic analysis for matching learner needs with appropriate examples [79]. While design pattern detection and crawling can be automated, albeit imperfectly, examples may still need manual discovery and filtering by domain goals or learner objectives (e.g., *I want to build a scalable app* or *I want to build one that supports high bandwidth media.*). Future work could also build curricula by sequencing examples, but this too remains unexplored. That being said, once examples are found or selected by professionals, our system can automate them into learning resources.

Code preprocessing and setup also present automation challenges. While AI agents following setup instructions could potentially handle downloading code, setting up environments, running AST parsing scripts, analyzing dynamic run logs, configuring websites for iframe embedding, moving files, and changing configuration variables, they may not achieve this deterministically and perfectly [162]. Setting file and variable exclusion lists for tractability may also require best-of-N approaches and/or using LLMs as judges [163], but even these may not be perfect. Overall, end-to-end automation and code preprocessing remain unsolved problems.

7.6.3 Example Scope and Limitations

Our systems can work with any JavaScript, CSS, or Python codebase, but practical limitations exist. Minified JavaScript may be incomprehensible, making open-source examples preferable to production code retrieved through use of “Inspect Element” in browser developer tools. For RM, inherent limitations of design pattern detection mean that even LLMs with full static and dynamic context are not perfect, though patterns themselves are finite and can be encoded into templates. For SE, some sites with complex libraries (e.g., Google Maps) may be challenging to instrument and parse both statically and dynamically through run logs. Moreover, not all examples yield clean exercise splits with elegant and incremental addition of new variables within learners’ zones of proximal development; some may require manual pruning, LLM-based filtering, or exclusion of examples that produce messy or redundant exercises with skewed variable distributions (e.g., one exercise introduces 10 variables and another introduces 1). For KM, tools like Ply [13] using visual regression testing, along with LLM judges, crowdsourcing, or computer vision algorithms, could help isolate visually impactful CSS, though this may not be perfect. In general, while our PM tools are designed to work with arbitrary examples, they may be imperfect, and some examples may yield more useful exercises than others. However, once we decide on examples and detect

their components, variables, classes, and methods, we can generate exercises reliably since the exercises themselves cover finite concepts (data flow sequencing, CSS techniques, design patterns) in templated ways.

7.6.4 Technical Contributions and Limitations

We have contributed an automated way to transform source code into visual, activity-driven learning processes. However, we have not contributed the most optimal pattern detection methods, a fully automated end-to-end solution for process management on arbitrary examples, or a fully automated way to find good examples. Our contribution lies in the conceptual framework and implementation of the process management approach, not in solving these broader automation challenges.

7.6.5 Empowering Educators with New Capabilities

Our tools enable educators to work with professional examples in fundamentally new ways. Instructors may previously have been limited to simpler examples, or may not immediately have been able to transform multiple complex examples into effective exercises. Our systems help them support many more professional examples and thereby better target intermediate-level learners, who are somewhat underrepresented in the CSEd literature. Educators can select examples, perform a few manual steps, and generate tutorials using the process management structure for their students. Even if exercise generation is not fully automated—though AI agents, crowdsourcing, and hybrid approaches could be explored—instructors can follow manual tweaks and run scripts to generate exercises. They can also check in periodically on learner progress through exercises, even if interventions are not fully automated.

Given that AI can easily generate syntax and code snippets with best practices in mind, ed-

educators should focus on testing concepts and project-based learning outcomes while learners use the PM tools. This will empower learners to gain the conceptual knowledge required to check and refine AI-generated code, a key responsibility of future software engineers. Educators training students on such conceptual outcomes and skills (going beyond syntax or basic program flow) may find our PM tools useful.

7.6.6 System Intelligence and Learner Modeling

Our current system does not know what learners understand; it is not an intelligent system in that sense. The experimenter used standardized rules for follow-ups and asked questions that may have been influenced by response semantics but were still largely rule-based. Reflections during tool use for SE and KM were evaluated after the fact against rubrics to determine concept understanding, but future versions of these systems could use AI to model learners' evolving knowledge semantically. Moreover, real-time assessment occurred in all systems according to standardized criteria to determine follow-up questions. Future work can explore making these interventions more robust by replacing the experimenter with an LLM agent—for example, determining when to have learners delve deeper versus move on in exercise sets.

7.6.7 Motivation and Commitment Considerations

We selected already motivated and committed participants, though some motivation may have stemmed from the presence of a hands-on instructor (i.e., the experimenter, myself). Future systems may benefit from being self-paced over longer periods, which our technology, when automated end-to-end should afford. Coupling the program with incentives for completion, such as unlocking other learning or earning opportunities on freelancing or MOOC platforms [164], could be an area of exploration. While we do not know how committed people will be in general, we

surmise that even committed learners may struggle without scaffolds like the PM tools. Real-time feedback from AI agents instead of experimenters may also be an option for sustaining motivation in future settings. Though less human, these agents could afford immediate feedback at any time. In summary, learner motivation is a research problem in and of itself, but once learners are motivated, the framework and tools we have contributed can help them gain conceptual knowledge.

7.6.8 Control Group Justification

We do not claim our system is better than human tutors. For scalable tools, we must compare against the best available tools rather than human tutors, and that thought inspired our choice of controls (CDT, VS Code, etc.). Most people cannot hire human tutors, so our focus is on comparing readily available solutions. Despite this constraint, our participants scored well on target learning outcomes, indicating some relative positioning toward optimal learning, but we acknowledge the limitations of our control group design.

7.6.9 Technical Learnings and Future Directions

Our work demonstrates several key technical learnings: it is possible to automate exercise generation through scripts given examples; these activities can provide learners with processes that target conceptual rather than just practical knowledge; and it is possible to elicit deep reflection, nuanced comparison, and chunked focus through careful scaffolding. However, achieving this in open environments may be harder due to challenges in automated example selection, cleaning, setup, exercise generation, and pruning. Recent advancements in AI agents, best-of-N techniques [158], and crowdsourcing to qualified individuals for quality assessment could potentially help address these challenges, but the research spaces remain open.

While our approach may apply to application programming contexts (we tested across JavaScript,

CSS, and Python), more research is needed to determine whether this works in other domains, such as different forms of engineering, writing, or media. We have not explored these possibilities.

7.6.10 Conceptual Contributions

Our primary conceptual contributions include the ability to autogenerate process-driven learning exercises from examples, identify relevant structures in examples and map them into learning exercises, work with professional examples from the wild (albeit modified), target intermediate-level learners, focus on conceptual knowledge (crucial for refining AI output), intentionally design for scaling and automation, and test CSEd best practices in the niche but high-impact space of upskilling intermediate-level developers. These contributions position our work as addressing a critical gap in the transition from introductory programming to professional practice.

7.7 Future Work and Generalization Opportunities

Our findings invite several opportunities for future research. First, integrating automated interventions with large language models (LLMs) is a path future work can explore for scaling PM tools beyond lab settings, enabling widespread use in diverse educational contexts. Second, adapting PM frameworks to different domains—such as CI/CD or UX Design—can demonstrate broader applicability. Finally, longitudinal studies examining PM scaffolding’s long-term impact on professional development could provide deeper insights into sustaining learner expertise over time.

Envisioning a Culture of Expert Development with AI Assistance: The impact of process management extends to both education and industry. As probabilistic machines such as LLMs handle simple coding tasks, human engineers may shift focus towards developing precise and compliant architectures [165] that satisfy stakeholders and maintain key relationships [59], [166], [167]. Hence, it is plausible that only true experts will remain essential amidst an AI transforma-

tion. By democratizing access to expert practices, our PM tools can potentially enhance workforce readiness, diversify the talent pool, and support coding boot camps and self-taught learners. Our findings and automatable scaffolding structures may pave the way for AI-driven platforms that personalize professional development. This vision aims to scalably bridge the gap between introductory programs and professional practice. At the same time, the human-AI collaboration we envision can conceivably initiate a future where both parties learn from each other and work together, each doing what they do best. In other words, we strive to set the stage for *AI to help train non-experts to become the experts that the AI does not replace* [59], [166], [167].

For stakeholders—including educational institutions, boot camps, self-taught learners, and industry leaders—PM tools offer explicit, tangible strategies for elevating professional expertise at scale. This democratization (i.e., inclusion of individuals without access to expert mentors) not only diversifies the tech workforce but strengthens its readiness amidst ongoing technological disruption (e.g., the rise of AI agents and copilots).

Scaling Process Management with LLMs: Looking forward, large language models (LLMs) offer a potential avenue for scaling PM tools [128]. Future work can explore whether fine-tuned LLMs with retrieval capabilities can automate some scaffolding interventions, such as prompting learners to articulate insights fully or encouraging feature exploration [129]. If debiased and vetted for fairness, such automation could enhance accessibility and enable deployment in diverse educational settings. This can potentially reduce the need for human facilitation while maintaining the benefits of PM.

In this vein, we aim to automate some of the wizarded interventions performed by the experimenter (myself) during the study sessions and work towards deploying the PM systems in the wild for learners worldwide. While making UX fixes and hosting the tools are routine, we present (a) a catalog of interventions we made along with an assessment of their automatability in Table 7.1,

and (b) LLM prompts in Table 7.2 to provide insight into how future work might proceed on the AI front.

Table 7.1 summarizes the specific experimenter interventions we layered onto the PM sessions to make sure participants could articulate their reasoning, resolve internal inconsistencies, and fully leverage the tool before moving on. We wanted the participants to reach their full potential, thinking through steps, processes, and reflections carefully according to best practices taken from the learning sciences. These nudges were in a sense process management interventions, and did not reveal the answer but prompted learners to clarify their understanding, surface missing details, or revisit diagrams/code so we could verify how well they understood the material after genuine exploration with the tool. While this scaffolding likely reduced the chance that confusion, not ability, limited performance, it also carried costs: every prompt interrupted the learner, demanded real-time judgment from the facilitator, and risked advantaging our experimental condition due to more chances for articulation and self-correction. We therefore view these interventions as methodological guardrails rather than required product features, and we treat broad implementation and deployment as future work. Many prompts (e.g., reminding participants to “say more,” revisit the UI, or log their reasoning) could be automated with rules or classifiers, but higher-touch moves may need LLMs that have the context of the codebase and rubrics. The systems such as LLMs must be able to tell whether an answer is incomplete versus incorrect and, in the case of the former, ask for custom just-in-time clarification without revealing the answer. This is an open research problem, and we do not know if LLMs can achieve this today. Still, we outline in Table 7.2 what such systems would have to detect, while noting that implementing all of these checks simultaneously may not be desirable; determining which interventions truly merit automation, and how to balance interruption costs for learners and implementers, remains an open research question.

Aside from the interventions in Table 7.1 and those described in the individual study sections,

there were some additional, easily automatable interventions the experimenter made, such as acquainting participants with UI interface (replaceable using a UI tour), telling them to say more a fixed number of times ($\sim 2-3X$) per session/question³ upon attempts to move forward (replaceable with a rule-based system), telling them to explore more a fixed number of times ($\sim 2-3X$) per session/question⁴ before moving on, encouraging participants to write (or occasionally draw) everything they are saying down if they were not doing so (live transcription features), asking them to explore the example's UX thoroughly if they were not covering all webpage components (a rule-based system can check for this using interaction data), using encouraging language such as “good work” or “good progress” if they answered all parts of the question before moving forward (a classifier or context-free zero-shot prompting could do this), reminding participants of definitions when they need them (a classifier or context-free zero-shot prompting could time and provide these based on participant responses), and instructing the participants to look at the site from an end-user perspective from the start (can be added to the pre/post-test interface).

³Sometimes, if we missed reminding them to say more on a specific question, we would remind them at the end of a study segment to consider if there is anything more to say for all the questions in that segment.

⁴Sometimes, if we missed reminding them to explore more on a specific question, we would remind them at the end of a study segment to consider if there is anything more to explore for all the questions/activities in that segment.

Intervention Type	Level of Automation Complexity
REREAD We prompted participants to reread the question or activity contents carefully, <i>if</i> they attempted to move on before covering all parts. E.g., in SE, we ensured they didn't ignore subparts of the suggested activities at the bottom of the interface.	Future versions of our interface could enforce that users highlight over each part of the question or activity to indicate they have read it. Future work can also investigate a simple NLP classifier or context-free zero-shot prompting to check for coverage.
RESTATE We restated content that the participant already said or wrote to follow up on meaning, <i>if</i> the meaning of a participant's answers was unclear to the experimenter and required clarification. This is a result of our microgenetic-style interviewing methods.	Future work can explore if a system such as a context-aware LLM can replace the experimenter who manually analyzed each response to determine whether to follow up. The LLM would need to know the difference between an incomplete or unclear answer (for which a follow-up is merited) and a wrong answer (no follow-up). It may require context of the codebase, relevant rubrics, and past study transcripts. This is an open research problem.
W QUESTIONS We asked who/what/where/when/why/next/how-style follow-ups to elicit more detail <i>when</i> we needed more information about a participant's answer in order to gauge it. e.g., In SE, when a participant referenced a set of "matched elements", the experimenter followed up asking, "What are the set of matched elements?"	Such probing questions are a result of our microgenetic-style interviewing methods. Future work can explore whether a system such as a context-aware LLM , refined with RAG, memory, SFT (supervised fine tuning), and RL (reinforcement learning), can replace the experimenter who decided on, phrased, and performed these custom follow-ups. The context-aware LLM may be equipped with study transcripts and a rubric (itself generated by a tool aware of the full codebase, like Cursor, but checked by a human-expert-in-the-loop during example-specific setup). Still, it will need to simulate nuanced decisions (e.g., we need more info on the learner's concept of "matched elements"), and hence, this is an open research problem.
FEATURES We told participants to use the features available <i>if</i> they were completing questions without using all the features, when they appeared stuck, or when they could not immediately resolve an internal inconsistency the experimenter pointed out. E.g., Reminding participants how to hide/show code, use the slider, or tinker with properties in KM	The first part can be considered rule-based : <i>if</i> analytics suggest a participant hasn't used particular features, our system can prompt them to do so. Future work can also develop a simple classifier to identify words or phrases (e.g., <i>if</i> they ask a question or ask for the answer) or periods <i>where</i> they make no progress (e.g., silence or minimal computer use), that together suggest the participant may be stuck.
INTERNAL CONSISTENCY We asked about anything internally inconsistent across their responses to activities <i>before</i> letting them move on to the next question or segment. E.g., In SE, one participant made a statement that contradicted their previous statement that the \$remove variable contains the images that you remove.	Future work can explore if a context-free zero-shot prompt can ingest the transcript of a learner's answer and flag internal inconsistencies or contradictions so our system can ask "what do you mean by ..." questions <i>before</i> letting them move to the next activity.

Table 7.1: Catalog of PM tool use intervention types and assessment of their automatability. Here, we define each intervention, including details around signals/heuristics that trigger it and the timing of when and if to intervene in each exercise/session.

Intervention Type	LLM Prompt to Detect Situation	LLM Prompt to Guide User
Elicit more detail, e.g., who, what, where, when, why, how, etc. (SE, KM, RM)	Analyze the response for completeness with respect to the AI-generated but expert-reviewed rubric for the example. If key learning outcome-related points around cause-effect, function interactions, or technical reasoning are missing or unclear from the user's externalizations, return true.	Return a custom version of: "Why do you think this behavior occurs? How does it relate to [OTHER THINGS THEY SAID OR READ]?"
Prompt to reread the question carefully	Check whether the user's response aligns with the core intent of the question and has made full use of the data and example-specific information available to them. If it does not directly and thoroughly answer the prompt, return true.	Return a more custom version of: "It looks like your answer may not fully address the question. Try rereading it carefully and checking for missing details."

Table 7.2: Future Work Sketch for exploring automation of some PM interventions via a state-of-the-art LLM fine-tuned on our existing study transcripts and capable of retrieval (RAG) over a vector database of domain-specific content. We present 2 sample prompts out of several developed.

With respect to Table 7.2, one can extrapolate the prompt to pre/post-test settings, where a bot can attempt to nudge participants to: (1) Explore the UX of the sites under study. (2) Share more

about mechanisms and rationales, given what they already said. (3) Consider all that the question is asking, including details about variables and values, granular techniques, classes/methods, and patterns, etc. (4) Recall their tool use.

Far Transfer of Process Management to other Domains: In this work, we presented process management as a way of helping novices move past suboptimal practices (e.g., getting lost in code complexity, stopping at surface-level observations, failing to transfer, etc.). However, this idea of machines “process managing” a human can be extended to other domains. For instance, it is commonly known that industries and sectors such as finance, healthcare, real estate, media, and even government profit considerably off of flawed or risky human practices (e.g., borrowing excessively rather than saving, eating unhealthily instead of exercising, chasing greener pastures rather than adopting a stable lifestyle, engaging in media production politics rather than honing one’s art, missing rules only to incur fines and taxes). In these domains, we imagine process management interventions can also help humans align their behaviors with their intents (e.g., meditative interventions to quell materialistic desires beyond one’s means, HIIT workouts between 25-minute work sessions and healthy curated meals, cost comparisons of settling down in an affordable town vs. renting month-to-month in expensive cities, presenting success vs. failure rates of status-seekers in one’s industry, automated compliance checking tools to help people and businesses get through government processes quickly). In domains as diverse as these, researchers may still benefit from reducing complexity, scaffolding a process, highlighting deeper long-term insights, and promoting reflection and transfer of knowledge from the theoretical (e.g., in theory I’d like to eat healthier) to the practical (e.g., cooking alone is difficult but I can try joining this cooking collective in the neighborhood). In this way, we see process management not just as a learning framework, but as a way of helping people make better decisions in their lives. We hope to see more research and AI “copilots” addressing this class of problems in the future.

CHAPTER 8

CONCLUSION: DEMOCRATIZING EXPERT PRACTICES THROUGH PROCESS MANAGEMENT

In this dissertation, we presented Process Management (PM), a scaffolding framework that systematically enhances learners' sensemaking of real-world professional code. Our evaluations demonstrate that PM tools effectively address obstacles around code complexity, superficial exploration, and conceptual transfer, in order to advance professional software education. Though sensemaking and scaffolding tools abound, few tools have leveraged the internet's abundant supply of professional code examples to address classes of problems centered around conceptual learning in software engineering. Our fundamental idea is that by scaffolding the process by which learners make sense of professional code examples, we can help them deeply understand and potentially transfer architectural knowledge. In the most general form, this idea of machines managing a human's process may apply beyond software engineering into other domains where novices adopt suboptimal strategies by default.

8.1 Practical Problems and Technical Solutions

In this work, we built and tested the following PM solutions:

Scaffolded Exercises (Ch. 4): SE addresses the challenge of superficial exploration of JavaScript UIs without understanding underlying variable value transformations. The tool provides structured exercises that guide learners backwards through the call stack, beginning with the intuitive visualization of the UI and then prompting them to explore the code that generates it. This approach helps learners develop a deeper understanding of the code's structure and functionality.

Knowledge Maps (Ch. 5): KM tackles the problem of learners’ difficulty in identifying nuanced visual layout features and the granular CSS techniques used to implement them. By shaping the process of exploration around comparing visuals to identify nuances before tracing them back to the code, KM helps learners develop a more comprehensive understanding of CSS techniques and their applications.

RALE Modules (Ch. 6): RM addresses the challenge of learners’ difficulty in understanding the rationale behind expert techniques in Python. By having learners map design pattern goals, mechanisms, and rationales from theory to practice, RM helps them develop a deeper understanding of the underlying principles and concepts that drive expert-level programming practices. This approach also encourages learners to reflect on theory in general and link it to practice, enhancing their overall understanding of the subject matter.

8.2 Conceptual Contributions

In this work, we also evaluated these core PM concepts:

Intuitive Sensemaking Order to Overcome Code Complexity: We found that learners often struggle to understand complex codebases, leading to superficial exploration. By providing a structured order of sensemaking, we can help learners navigate the complexity of real-world code and develop a deeper understanding of its structure and functionality. For example, in SE, we found that learners who followed the structured exercises were able to identify the underlying variable value transformations and understand how they contributed to the overall functionality of the code. This approach not only enhances learners’ understanding of the code but also helps them ground their conceptions of data flow in concrete example-specifics. While scaffolding is a rich literature space, ordering the process of sensemaking professional source code for conceptual learning seems more novel.

Linking Prior Knowledge to Overcome Surface-Level Focus: We found that learners often fixate on immediately surface-level aspects of applications or code without considering the underlying principles, concepts, architectures, or data flow. By linking prior knowledge (e.g., intuitive visuals, architectural theory) to deeper knowledge (e.g., internal data flow, nuanced differences or granular underlying CSS, practical instantiations of theory), we were able to help learners develop a more comprehensive understanding of the code and its functionality and rationale. For example, in KM, we found that learners who were prompted to compare visuals and identify nuances before tracing them back to the code were able to develop a more detailed understanding of CSS techniques and their applications. This approach not only enhances learners' understanding of the code but also helps prepare them better for understanding its rationale.

Actively Mapping Concepts to Example Specifics to Aid in Conceptual Transfer: Traditionally, conceptual transfer is very challenging for learners. Learners often struggle to apply their knowledge to new contexts or situations, and fail to obtain a generalizable understanding of underlying principles and concepts. By actively mapping concepts (e.g., data value transforms, visual nuances, design pattern theory) to professional example-specific code, we were able to help learners develop a deeper understanding of the underlying principles and concepts that drive expert-level programming practices. For example, in RM, we found that learners who were prompted to map design pattern goals, mechanisms, and rationales from theory to practice were able to develop a more comprehensive understanding of the architectural concepts that drive expert-level programming practices. This approach not only enhances learners' understanding of the code but also helps them apply their knowledge to new scenarios.

8.3 Evaluation Summary, PM Values, and Future Directions

In this work, we have shown that guiding exploration with structured prompts, intuitive visuals, ordered exercises, resurfaced prior knowledge, careful comparison, pointed reflection, tinkerable interfaces, and analogical mapping can potentially boost architectural understanding, conceptual knowledge transfer, and awareness of the rationales behind expert techniques. Fundamentally, our thesis advocates for a future where structured scaffolding via Process Management tools complements AI-driven code generation, empowering learners to become experts whose deep conceptual understanding and adaptable sensemaking capabilities remain indispensable in professional software practice. We challenge educators, designers, and researchers to join in fostering this synergistic human-AI future. By doing so, we as a society can value the unique strengths of both humans and AI systems, ensuring that the next generation of software engineers is equipped with the skills and knowledge necessary to thrive in an increasingly complex and automated world.

We envision a future of application development in which probabilistic machines such as LLMs can offer guidance and automation during both training and professional practice (e.g., GitHub Copilot) [3]. Embedded in the context of their emotions and relations [166], [167], human experts with conceptual understanding can perform compliance [165] checks and guarantee domain-level precision [59], [91], [92]. The active, symbiotic, and pedagogical relationship between automated tools and human developers will prevail from the first computing lesson to the most advanced stages of enterprise product development.

REFERENCES

- [1] A. Begel and A. J. Ko, “Learning outside the classroom,” 2019.
- [2] S. E. Inc. “Stack overflow insights - developer hiring, marketing, and user research,” Accessed: Jul. 16, 2021.
- [3] B. Zhang, P. Liang, X. Zhou, A. Ahmad, and M. Waseem, “Practices and challenges of using github copilot: An empirical study,” *arXiv preprint arXiv:2303.08733*, 2023.
- [4] A. Shah, A. Chernova, E. Tomson, L. Porter, W. G. Griswold, and A. G. Soosai Raj, “Students’ use of github copilot for working with large code bases,” in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, 2025, pp. 1050–1056.
- [5] A. Osmani, *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer’s Guide*. “O’Reilly Media, Inc.”, 2012.
- [6] G. Web. “Web fundamentals,” Google Developers, Accessed: Jul. 16, 2021.
- [7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, “Design patterns: Elements of reusable object-oriented software 1st edition addison-wesley professional,” 1994.
- [8] S. Magazine, *Understanding Advanced JavaScript*. Smashing Media GmbH, 2013, ISBN: 9783943075601.
- [9] K. Simpson, *You Don’t Know JS: Async & Performance*. “O’Reilly Media, Inc.”, 2015.
- [10] Code.org. “Learn today, build a brighter tomorrow.,” Code.org, Accessed: Jul. 16, 2021.
- [11] Codecademy. “Learn to code - for free,” Codecademy, Accessed: Jul. 15, 2021.
- [12] LeetCode, *System design for interviews and beyond*, <https://leetcode.com/explore/learn/card/system-design/>, Accessed: 2025-03-13.
- [13] S. Lim, “Ply: Visual regression pruning for web design source inspection,” in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2017, pp. 130–135.

- [14] J. Hibschan, D. Gergle, E. O'Rourke, and H. Zhang, "Isopleth: Supporting sensemaking of professional web applications to create readily available learning experiences," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 26, no. 3, pp. 1–42, 2019.
- [15] K. E. Weick, *Sensemaking in organizations*. Sage, 1995, vol. 3.
- [16] C. Naumer, K. Fisher, and B. Dervin, "Sense-making: A methodological perspective," in *Sensemaking Workshop, CHI*, vol. 8, 2008.
- [17] P. Pirolli and S. Card, "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis," in *Proceedings of international conference on intelligence analysis*, McLean, VA, USA, vol. 5, 2005, pp. 2–4.
- [18] M. C. Linn and M. J. Clancy, "The case for case studies of programming problems," *Communications of the ACM*, vol. 35, no. 3, pp. 121–132, 1992.
- [19] M. Weiser and J. Shertz, "Programming problem representation in novice and expert programmers," *International Journal of Man-Machine Studies*, vol. 19, no. 4, pp. 391–398, 1983.
- [20] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, "Not seeing the forest for the trees: Novice programmers and the solo taxonomy," *ACM SIGCSE Bulletin*, vol. 38, no. 3, pp. 118–122, 2006.
- [21] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive psychology*, vol. 19, no. 3, pp. 295–341, 1987.
- [22] K. Mangaroska, K. Sharma, M. Giannakos, H. Tr etteberg, and P. Dillenbourg, "Gaze insights into debugging behavior using learner-centred analysis," in *Proceedings of the 8th international conference on learning analytics and knowledge*, 2018, pp. 350–359.
- [23] D. Odell, *Pro JavaScript Development: Coding, Capabilities, and Tooling*. Apress, 2014.
- [24] M. Monroe and C. Dugan, "Disperse: Enabling web-based visualization in multi-screen and multi-user environments," in *IFIP Conference on Human-Computer Interaction*, Springer, 2015, pp. 418–435.
- [25] S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman, *How learning works: Seven research-based principles for smart teaching*, 2010.

- [26] C. Quintana et al., “A scaffolding design framework for software to support science inquiry,” *The journal of the learning sciences*, vol. 13, no. 3, pp. 337–386, 2004.
- [27] A. J. Ko, B. A. Myers, and H. H. Aung, “Six learning barriers in end-user programming systems,” in *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, IEEE, 2004, pp. 199–206.
- [28] D. Wood, J. Bruner, and G. Ross, “The role of tutoring in problem solving,” *Journal of child psychology and psychiatry*, vol. 17, no. 2, 1976.
- [29] D. L. Schwartz and J. D. Bransford, “A time for telling,” *Cognition and instruction*, vol. 16, no. 4, pp. 475–5223, 1998.
- [30] J. S. Hui, D. Gergle, and E. M. Gerber, “Introassist: A tool to support writing introductory help requests,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–13.
- [31] J. Shi, A. Shah, G. Hedman, and E. O’Rourke, “Pyrus: Designing a collaborative programming game to promote problem solving behaviors,” in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.
- [32] B. Dorn, A. Stankiewicz, and C. Roggi, “Lost while searching: Difficulties in information seeking among end-user programmers,” *Proceedings of the American Society for Information Science and Technology*, vol. 50, no. 1, pp. 1–10, 2013.
- [33] M. L. Gick and K. J. Holyoak, “Schema induction and analogical transfer,” *Cognitive psychology*, vol. 15, no. 1, pp. 1–38, 1983.
- [34] D. Gentner, “Structure-mapping: A theoretical framework for analogy,” *Cognitive science*, vol. 7, no. 2, pp. 155–170, 1983.
- [35] L. S. Vygotsky and M. Cole, *Mind in society: Development of higher psychological processes*. Harvard university press, 1978.
- [36] B. Y. White and J. R. Frederiksen, “Inquiry, modeling, and metacognition: Making science accessible to all students,” *Cognition and instruction*, vol. 16, no. 1, pp. 3–118, 1998.
- [37] E. Soloway and K. Ehrlich, “Empirical studies of programming knowledge,” *IEEE Transactions on software engineering*, no. 5, pp. 595–609, 1984.

- [38] M.-A. Storey, “Theories, tools and research methods in program comprehension: Past, present and future,” *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.
- [39] T. J. McGill and S. E. Volet, “A conceptual framework for analyzing students’ knowledge of programming,” *Journal of research on Computing in Education*, vol. 29, no. 3, pp. 276–297, 1997.
- [40] D. Gentner and A. L. Stevens, *Mental models*. Psychology Press, 2014.
- [41] S. Bhatta and A. Goel, “Use of mental models for constraining index learning in experience-based design,” in *Proceedings of the AAAI workshop on Constraining Learning with Prior Knowledge*, Citeseer, 1992, pp. 1–10.
- [42] A. A. DiSessa, “Toward an epistemology of physics,” *Cognition and instruction*, vol. 10, no. 2-3, pp. 105–225, 1993.
- [43] B. Danielak, “How code takes shape: Studying a student’s program evolution,” *Cognition and Instruction*, pp. 1–38, 2022.
- [44] P. J. Guo, “Online python tutor: Embeddable web-based program visualization for cs education,” in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 579–584.
- [45] P. Guo, “Ten million users and ten years later: Python tutor’s design guidelines for building scalable and sustainable research software in academia,” in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 1235–1251.
- [46] J. Sorva, “Notional machines and introductory programming education,” *Trans. Comput. Educ*, vol. 13, no. 2, pp. 1–31, 2007.
- [47] D. Feltey et al., “Languages the racket way,” *Language Workbench Challenge*, vol. 65, 2016.
- [48] K. S.-P. Chang and B. A. Myers, “Webcrystal: Understanding and reusing examples in web authoring,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012, pp. 3205–3214.
- [49] A. J. Ko and B. A. Myers, “Designing the whyline: A debugging interface for asking questions about program behavior,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 151–158.

- [50] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, “Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code,” in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2015, pp. 3–12.
- [51] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, “Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.
- [52] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, “Example-centric programming: Integrating web search into the development environment,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 513–522.
- [53] E. Fast and M. S. Bernstein, “Meta: Enabling programming languages to learn from the crowd,” in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 2016, pp. 259–270.
- [54] S. Oney and B. Myers, “Firecrystal: Understanding interactive behaviors in dynamic web pages,” in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2009, pp. 105–108.
- [55] J. Hibsichman and H. Zhang, “Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection,” in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 270–279.
- [56] J. Hibsichman and H. Zhang, “Telescope: Fine-tuned discovery of interactive web ui feature implementation,” in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 2016, pp. 233–245.
- [57] B. Burg, A. J. Ko, and M. D. Ernst, “Explaining visual changes in web interfaces,” in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 259–268.
- [58] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, “Interactive record/replay for web application debugging,” in *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 2013, pp. 473–484.

- [59] A. J. Ko, B. A. Myers, M. Coblenz, and H. H. Aung, “A retrospective on how developers seek, relate, and collect information about code,” *IEEE Transactions on Software Engineering*, 2025.
- [60] M. Weiser, “Programmers use slices when debugging,” *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.
- [61] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: A taxonomy and survey,” *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.
- [62] D. W. Shaffer and M. Resnick, “■ thick■ authenticity: New media and authentic learning,” *Journal of interactive learning research*, vol. 10, no. 2, pp. 195–216, 1999.
- [63] A. Shah, J. Yu, T. Tong, and A. G. Soosai Raj, “Working with large code bases: A cognitive apprenticeship approach to teaching software engineering,” in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, 2024, pp. 1209–1215.
- [64] R. Engineer, V. Yaremchuk, E. Suner, O. Khamis, A. Apostolu, and A. Ng, “Uml mentor: A tool for interactive and collaborative software design education,” in *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 2*, 2024, pp. 309–309.
- [65] G. Agerwala and L. Bass, “Teaching software architecture design-building intuition,” in *Proceedings of the 1st International Workshop on Designing Software*, 2024, pp. 27–33.
- [66] A. Birillo, “Bringing industry-grade code quality and practices into software engineering education (doctoral consortium),” in *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, 2024, pp. 1–2.
- [67] C. Gupta and V. Gupta, “C4 skills in the engineering graduate: A study to align software engineering education with market-driven software industry needs,” *IEEE Transactions on Education*, vol. 67, no. 1, pp. 31–43, 2023.
- [68] W. L. Pantoja Yopez, J. A. Hurtado, L. M. Bibbo, A. Fernandez, and A. Bandi, “Towards a software architecture training pattern language,” in *Proceedings of the 30th Conference on Pattern Languages of Programs*, 2023, pp. 1–33.
- [69] M. Barr, O. Andrei, A. Morrison, and S. W. Nabi, “The development of students’ professional competencies on a work-based software engineering program,” in *Proceedings of*

the 55th ACM Technical Symposium on Computer Science Education V. 1, 2024, pp. 81–87.

- [70] B. Dorn and M. Guzdial, “Learning on the job: Characterizing the programming knowledge and learning strategies of web designers,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 703–712.
- [71] M. X. Liu et al., “Unakite,” in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, 2019, pp. 67–80.
- [72] C. E. Hmelo-Silver and M. G. Pfeffer, “Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions,” *Cognitive science*, vol. 28, no. 1, pp. 127–138, 2004.
- [73] J. Sorva, V. Karavirta, and L. Malmi, “A review of generic program visualization systems for introductory programming education,” *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–64, 2013.
- [74] R. Duran, A. Zavgorodniaia, and J. Sorva, “Cognitive load theory in computing education research: A review,” *ACM Transactions on Computing Education (TOCE)*, vol. 22, no. 4, pp. 1–27, 2022.
- [75] A. Marques, G. Viviani, and G. C. Murphy, “Assessing semantic frames to support program comprehension activities,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, IEEE, 2021, pp. 13–24.
- [76] I. Ma, A. Krone-Martins, and C. Videira Lopes, “Integrating ai tutors in a programming course,” in *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*, 2024, pp. 130–136.
- [77] L. Kaldaras et al., “Employing technology-enhanced feedback and scaffolding to support the development of deep science understanding using computer simulations,” *International Journal of STEM Education*, vol. 11, no. 1, p. 30, 2024.
- [78] B. R. Belland, A. E. Walker, N. J. Kim, and M. Lefler, “Synthesizing results from empirical research on computer-based scaffolding in stem education: A meta-analysis,” *Review of Educational Research*, vol. 87, no. 2, pp. 309–344, 2017.
- [79] OpenAI, *Rag*, 2024.

- [80] L. Zheng, Y. Zhen, J. Niu, and L. Zhong, “An exploratory study on fade-in versus fade-out scaffolding for novice programmers in online collaborative programming settings,” *Journal of Computing in Higher Education*, vol. 34, no. 2, pp. 489–516, 2022.
- [81] X. Hou, B. J. Ericson, and X. Wang, “Understanding the effects of using parsons problems to scaffold code writing for students with varying cs self-efficacy levels,” in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 2023, pp. 1–12.
- [82] J. Whalley, A. Settle, and A. Luxton-Reilly, “A think-aloud study of novice debugging,” *ACM TOCE*, vol. 23, no. 2, pp. 1–38, 2023.
- [83] D. Loksa et al., “Metacognition and self-regulation in programming education: Theories and exemplars of use,” *ACM Transactions on Computing Education (TOCE)*, vol. 22, no. 4, pp. 1–31, 2022.
- [84] Y. Pechorina, K. Anderson, and P. Denny, “Metacodenition: Scaffolding the problem-solving process for novice programmers,” in *Proceedings of the 25th Australasian Computing Education Conference*, 2023, pp. 59–68.
- [85] S. Lin, N. Meng, D. Kafura, and W. Li, “Pdl: Scaffolding problem solving in programming courses,” in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 2021, pp. 185–191.
- [86] C. T. Wolf and D. Paine, “Sensemaking practices in the everyday work of ai/ml software engineering,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 86–92.
- [87] E. Agrawal et al., “Code compass: A study on the challenges of navigating unfamiliar codebases,” *arXiv preprint arXiv:2405.06271*, 2024.
- [88] A. Horvath, B. Myers, A. Macvean, and I. Rahman, “Using annotations for sensemaking about code,” in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, 2022, pp. 1–16.
- [89] L. Himbeault and C. Latulipe, “An initial exploration of code diagram query effectiveness,” in *Proceedings of the 26th Western Canadian Conference on Computing Education*, 2024, pp. 1–7.

- [90] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, “Measuring program comprehension: A large-scale field study with professionals,” *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.
- [91] A. Chiş, T. Gırba, and O. Nierstrasz, “The moldable debugger,” in *International Conference on Software Language Engineering*, Springer, 2014, pp. 102–121.
- [92] I. Bouzenia, P. Devanbu, and M. Pradel, “Repairagent,” *arXiv preprint arXiv:2403.17134*, 2024.
- [93] M. Guzdial, “Software-realized scaffolding to facilitate programming for science learning,” *Interactive learning environments*, vol. 4, no. 1, 1994.
- [94] F. Cammaerts and M. Snoeck, “Assessing the testing skills transfer of model-based testing on testing skill acquisition,” *Software and Systems Modeling*, vol. 23, no. 4, pp. 953–971, 2024.
- [95] D. G. Feitelson, “From code complexity metrics to program comprehension,” *Communications of the ACM*, vol. 66, no. 5, pp. 52–61, 2023.
- [96] D. Brown, E. Mulder, and S. Mulder, “Toward improving binary program comprehension via embodied immersion: A survey,” *arXiv:2404.17051*, 2024.
- [97] C. Tikva and E. Tambouris, “The effect of scaffolding programming games and attitudes towards programming on the development of computational thinking,” *Education and Information Technologies*, vol. 28, no. 6, pp. 6845–6867, 2023.
- [98] A. K. Veerasamy, D. D’Souza, and M.-J. Laakso, “Identifying novice student programming misconceptions and errors from summative assessments,” *Journal of Educational Technology Systems*, vol. 45, no. 1, pp. 50–73, 2016.
- [99] A. Heinonen, B. Lehtelä, A. Hellas, and F. Fagerholm, “Synthesizing research on programmers’ mental models of programs, tasks and concepts—a systematic literature review,” *Information and Software Technology*, vol. 164, p. 107 300, 2023.
- [100] P. Kather, R. Duran, and J. Vahrenhold, “Through (tracking) their eyes: Abstraction and complexity in program comprehension,” *ACM Transactions on Computing Education (TOCE)*, vol. 22, no. 2, pp. 1–33, 2021.

- [101] P. L. S. Barbosa, R. A. F. d. Carmo, J. P. Gomes, and W. Viana, “Adaptive learning in computer science education: A scoping review,” *Education and Information Technologies*, vol. 29, no. 8, pp. 9139–9188, 2024.
- [102] C. Izu et al., “Fostering program comprehension in novice programmers-learning activities and learning trajectories,” in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 2019, pp. 27–52.
- [103] N. Sibia, “Enhancing novice programmers’ understanding through interactive visualizations and multiple representations,” in *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, 2024, pp. 1–2.
- [104] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, “Eyes on code: A study on developers’ code navigation strategies,” *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1692–1704, 2020.
- [105] L. Müller, M. S. Silveira, and C. S. de Souza, “Source code comprehension and appropriation by novice programmers: Understanding novice programmers’ perception about source code reuse,” *Journal on Interactive Systems*, vol. 10, no. 2, pp. 96–109, 2019.
- [106] O. Alghamdi, S. Clinch, M. Alhamadi, and C. Jay, “Novice programmers strategies for online resource use and their impact on source code,” in *2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering*, IEEE, 2023, pp. 92–104.
- [107] W. Groeneveld, J. Vennekens, and K. Aerts, “Software engineering education beyond the technical,” *arXiv:1910.09865*, 2019.
- [108] V. Garousi, G. Giray, E. Tuzun, C. Catal, and M. Felderer, “Closing the gap between software engineering education and industrial needs,” *IEEE software*, vol. 37, no. 2, pp. 68–77, 2019.
- [109] C. T. N. Tran, M. Kanwal, K. Fasiang, and E. O’Rourke, “Exploring student-perceived dimensions of authenticity in high school computer science,” in *Proceedings of the 2025 ACM Conference on International Computing Education Research V. 1*, 2025, pp. 93–108.
- [110] J. Herrington and R. Oliver, “An instructional design framework for authentic learning environments,” *Educational technology research and development*, vol. 48, no. 3, pp. 23–48, 2000.

- [111] S. A. Barab, K. D. Squire, and W. Dueber, “A co-evolutionary model for supporting the emergence of authenticity,” *Educational technology research and development*, vol. 48, no. 2, pp. 37–62, 2000.
- [112] L. Moors, A. Luxton-Reilly, and P. Denny, “Transitioning from block-based to text-based programming languages,” in *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, IEEE, 2018, pp. 57–64.
- [113] B. DiSalvo, “Graphical qualities of educational technology: Using drag-and-drop and text-based programs for introductory computer science,” *IEEE computer graphics and applications*, vol. 34, no. 6, pp. 12–15, 2014.
- [114] D. Weintrop and U. Wilensky, “To block or not to block, that is the question: Students’ perceptions of blocks-based programming,” in *Proceedings of the 14th international conference on interaction design and children*, 2015, pp. 199–208.
- [115] Y. Lin, D. Weintrop, and J. McKenna, “Is block-based programming ■real programming■?” In *Proceedings of the 18th International Conference of the Learning Sciences-ICLS 2024*, pp. 2165-2166, International Society of the Learning Sciences, 2024.
- [116] A. Betz, S. Flake, M. Mierwald, and M. Vanderbeke, “Modelling authenticity in teaching and learning contexts: A contribution to theory development and empirical investigation of the construct,” in Singapore: International Society of the Learning Sciences, 2016.
- [117] D. Schriebl, A. Müller, and N. Robin, “Modelling authenticity in science education,” *Science & Education*, vol. 32, no. 4, pp. 1021–1048, 2023.
- [118] M. Guzdial and A. E. Tew, “Imagineering inauthentic legitimate peripheral participation: An instructional design approach for motivating computing education,” in *Proceedings of the second international workshop on Computing education research*, 2006, pp. 51–58.
- [119] P. Kamthan, “On the prospects and concerns of integrating open source software environment in software engineering education,” *Journal of Information Technology Education: Research*, vol. 6, no. 1, pp. 45–64, 2007.
- [120] E. Raymond, “The cathedral and the bazaar,” *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.
- [121] R. Haskell, “Transfer of learning: Cognition,” *Instruction, and Reasoning*, San Diego, CA: Academic, 2001.

- [122] N. Furman and J. Sibthorp, “Leveraging experiential learning techniques for transfer,” *New directions for adult and continuing education*, vol. 2013, no. 137, pp. 17–26, 2013.
- [123] A. Hallie Lydia Osmani, *Learning Patterns*. 2021.
- [124] I. E. Harel and S. E. Papert, *Constructionism*. Ablex Publishing, 1991.
- [125] K. Doherty and G. Doherty, “The construal of experience in hci: Understanding self-reports,” *International Journal of Human-Computer Studies*, vol. 110, pp. 63–74, 2018.
- [126] C. M. Lewis, “The importance of students’ attention to program state: A case study of debugging behavior,” in *Proceedings of the ninth annual international conference on International computing education research*, 2012, pp. 127–134.
- [127] S. Dziallas and S. Fincher, “Accountable disciplinary knowledge in computing education: A case-comparative approach,” in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, 2019, pp. 1–9.
- [128] OpenAI, *Openai prompt engineering guide*, Accessed: March 2024, 2024.
- [129] OpenAI, *Openai fine-tuning guide*, Accessed: March 2024, 2024.
- [130] M. T. Chi, “Three types of conceptual change: Belief revision, mental model transformation, and categorical shift,” in *International handbook of research on conceptual change*, Routledge, 2009, pp. 89–110.
- [131] Google, *Google analytics*, <http://analytics.google.com/>, Accessed: 2025-02-24.
- [132] Hotjar, *Website heatmaps & behavior analytics tools*, <https://hotjar.com/>, Accessed: 2025-02-24.
- [133] M. Li et al., “Coannotating: Uncertainty-guided work allocation between human and large language models for data annotation,” *arXiv preprint arXiv:2310.15638*, 2023.
- [134] Z. Buçinca, M. B. Malaya, and K. Z. Gajos, “To trust or to think: Cognitive forcing functions can reduce overreliance on ai in ai-assisted decision-making,” *Proceedings of the ACM on Human-computer Interaction*, vol. 5, no. CSCW1, pp. 1–21, 2021.

- [135] L. W. Anderson and D. R. Krathwohl, *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives: complete edition*. Addison Wesley Longman, Inc., 2001.
- [136] R. C. Schank and R. P. Abelson, *Scripts, plans, goals, and understanding: An inquiry into human knowledge structures*. Psychology Press, 2013.
- [137] D. E. Rumelhart, "Schemata: The building blocks of cognition," in *Theoretical issues in reading comprehension*, Routledge, 2017, pp. 33–58.
- [138] M. Riel, "Education in the 21st century: Just-in-time learning or learning communities," in *fourth annual conference of the emirates center for strategic studies and research, Abu Dhabi*, 1998.
- [139] J. Yang, Y. Lee, and K. H. Chang, "Evaluations of jaguarcode: A web-based object-oriented programming environment with static and dynamic visualization," *Journal of Systems and Software*, vol. 145, pp. 147–163, 2018.
- [140] K. Cunningham, B. J. Ericson, R. Agrawal Bejarano, and M. Guzdial, "Avoiding the turing tarpit: Learning conversational programming by starting from code's purpose," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–15.
- [141] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating design pattern detectors from pattern specifications," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, IEEE, 2003, pp. 245–248.
- [142] F. Bergenti and A. Poggi, "Improving uml designs using automatic design pattern detection," in *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*, World Scientific, 2002, pp. 771–784.
- [143] M. L. Bernardi and G. A. Di Lucca, "Model-driven detection of design patterns," in *2010 IEEE International Conference on Software Maintenance*, IEEE, 2010, pp. 1–5.
- [144] A. Alnusair, T. Zhao, and G. Yan, "Rule-based detection of design patterns in program code," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 3, pp. 315–334, 2014.

- [145] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, “A survey on design pattern detection approaches,” *International Journal of Software Engineering (IJSE)*, vol. 7, no. 3, pp. 41–59, 2016.
- [146] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, “Source code and design conformance, design pattern detection from source code by classification approach,” *Applied Soft Computing*, vol. 26, pp. 357–367, 2015.
- [147] F. A. Fontana and M. Zanoni, “A tool for design pattern detection and software architecture reconstruction,” *Information sciences*, vol. 181, no. 7, pp. 1306–1324, 2011.
- [148] R. Mzid, I. Rezgui, and T. Ziadi, “Attention-based method for design pattern detection,” in *European Conference on Software Architecture*, Springer, 2024, pp. 86–101.
- [149] J. Asaad and E. Avksentieva, “A review of approaches to detecting software design patterns,” in *2024 35th Conference of Open Innovations Association (FRUCT)*, IEEE, 2024, pp. 142–148.
- [150] S. K. Pandey, S. Chand, J. Horkoff, M. Staron, M. Ochodek, and D. Durisic, “Design pattern recognition: A study of large language models,” *Empirical Software Engineering*, vol. 30, no. 3, p. 69, 2025.
- [151] L. A. Suchman, *Plans and situated actions: The problem of human-machine communication*. Cambridge university press, 1987.
- [152] E. Hutchins, “How a cockpit remembers its speeds,” *Cognitive science*, vol. 19, no. 3, pp. 265–288, 1995.
- [153] H. Zhang and E. O’Rourke, *Readily available learning experiences: Turning the entire web into scaffolded examples to bridge conceptual knowledge gaps for novice web developers*, NSF Grant, 2017.
- [154] R. S. Siegler and K. Crowley, “The microgenetic method: A direct means for studying cognitive development.,” *American psychologist*, vol. 46, no. 6, p. 606, 1991.
- [155] F. Kontogianni, L. Hope, P. J. Taylor, A. Vrij, and F. Gabbert, ““tell me more about this. . .”: An examination of the efficacy of follow-up open questions following an initial account,” *Applied Cognitive Psychology*, vol. 34, no. 5, pp. 972–983, 2020.

- [156] Model Context Protocol, *Getting started: Introduction*, <https://modelcontextprotocol.io/docs/getting-started/intro>, Accessed: 2025-09-17, 2025.
- [157] Reddit Users, *All ai coding agents you know?* https://www.reddit.com/r/OpenAI/comments/1m54yjsx/all_ai_coding_agents_you_know/, Discussion thread on r/OpenAI. Accessed: 2025-09-17, 2025.
- [158] Y. Ichihara et al., “Evaluation of best-of-n sampling strategies for language model alignment,” *arXiv preprint arXiv:2502.12668*, 2025.
- [159] Browserbase Inc., *Browserbase*, <https://www.browserbase.com/>, Accessed: 2025-09-17, 2025.
- [160] OpenAI, *Openai agents python*, <https://openai.github.io/openai-agents-python/>, Accessed: 2025-09-17, 2025.
- [161] LangChain, *Langgraph*, <https://www.langchain.com/langgraph>, Accessed: 2025-09-17, 2025.
- [162] A. Vaswani et al., “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [163] J. Gu et al., “A survey on llm-as-a-judge,” *arXiv preprint arXiv:2411.15594*, 2024.
- [164] G. Chen, D. Davis, M. Krause, E. Aivaloglou, C. Hauff, and G.-J. Houben, “From learners to earners: Enabling mooc learners to apply their skills and earn money in an online market place,” *IEEE Transactions on Learning Technologies*, vol. 11, no. 2, pp. 264–274, 2016.
- [165] Secureframe, *Compliance frameworks*, <https://secureframe.com/hub/grc/compliance-frameworks>, Accessed: 2025-03-12.
- [166] K. Quach, “Google ai chatbot more empathetic than real doctors in tests,” *The Register*, Jan. 2024, Archived at: <https://web.archive.org/web/20241212173224/>.
- [167] M. Tomprou and M. K. Lee, “Employment relationships in algorithmic management: A psychological contract perspective,” *Computers in Human Behavior*, vol. 126, p. 106997, 2022.

VITA

Gobi Dasu is a Ph.D. candidate in Computer Science at Northwestern University, advised by Prof. Haoqi Zhang and Prof. Nell O'Rourke. His research interests lie in software engineering professional development, with a specialization in remote future-of-work settings and ICT4D. His current work, the RALE-Modules project, aims to help novice programmers develop an architectural understanding of professional websites so they can build their own enterprise-grade applications.

Gobi holds a Bachelor of Science (B.S.) and Master of Science (M.S.) in Computer Science from Stanford University, where he specialized in Human-Computer Interaction (HCI) and Artificial Intelligence (AI). He accelerated through Stanford's undergraduate program in two years, drawn back to academia after traveling the world and initiating a global talent network.

Outside academia, Gobi is a co-founder and board member of LD Talent, a future-of-work platform that financially incentivizes tech workers to engage in lifelong learning. The platform has connected hundreds of developers, data scientists, and designers from diverse, often overlooked geographies with work opportunities while ensuring continuous upskilling. Since its inception, the LD network has helped close the STEM skill gap through its "learn-to-earn" model, promoting education and economic uplift.

As a digital nomad, Gobi has traveled to 47 countries and 41 U.S. states, experiences that have shaped his passion for building scalable solutions to global challenges. His startup work and academic research aim to advance the future of work by aligning industry demand with accessible, high-quality education and professional growth opportunities. Inspired by university research experiences and meetups across 9 African and South Asian nations, Gobi and his partner Anisha hope to leverage AI to train and manage the world's overlooked knowledge workers.