

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

GOBI DASU, ELEANOR O'ROURKE, HAOQI ZHANG, MAXINE WHITELY, ROXANNA WILCOX, LEV ROSENBERG, MIERAF MULAT, JONATHAN LIU, and NATALIE BREWSTER, Northwestern University, Evanston, Illinois, USA

Background and Context: 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 under-researched relative to its socioeconomic impact. Learner-focused inspection tools exist (e.g., Ply, Isopleth, Python Tutor), but these often address specific problems (e.g., implicit CSS dependencies, async JS, call stacks), without systematic guidance for developing deep conceptual models of architectures and techniques across large real-world codebases or interactions.

Objectives: We aim to support non-experts in learning holistically from professional code and to understand how they explore and learn in this context. Given that learners outnumber expert instructors who can teach these concepts, we propose Process Management (PM), a framework for designing tools that scaffold sensemaking on underused yet plentiful real-world code. PM addresses core obstacles learners face (code complexity, surface-level focus, and transfer challenges) through core design principles (intuitive sensemaking processes, comparison, reflection). After prototyping and piloting over 5 years, we developed 3 PM tools that surface JavaScript, CSS, and Python techniques. In this work, we seek to understand the effectiveness of PM tools, and the processes learners follow with and without PM support.

Study Methods: We report on think-aloud studies conducted over 1.5 years to evaluate our 3 PM tools. Data include ~300 hours of remote observation of intermediate-level developers using these tools or controls (Chrome DevTools, VS Code). There were five participants in each condition for each tool. Studies consisted of tool training and pre-test, tool-use, and post-test phases, followed by a post-study interview. We qualitatively coded pre/post- and tool-use data, and present learning gains and in-task behavioral metrics.

Findings: Our mixed-methods evaluations show PM tool users explored effectively (e.g., comparing, mapping) and achieved ~3× more learning outcomes (e.g., conceptual clarity, transferable patterns). Such outcomes were observed consistently across the three programming domains under study. Our qualitative data shows how PM tools can shape learners' exploration order and process, enhance their understanding of nuanced

Gobi Dasu is responsible for leading the work, system development, conducting studies, theoretical synthesis, and revisions. This work was supported in part by the National Science Foundation (Grant No. NSF1735977).

Authors' Contact Information: Gobi Dasu (corresponding author), Northwestern University, Evanston, Illinois, USA; e-mail: gobi@u.northwestern.edu; Maxine Whitely, Northwestern University, Evanston, Illinois, USA; e-mail: mp-whitely@gmail.com; Roxanna Wilcox, Northwestern University, Evanston, Illinois, USA; e-mail: roxygwilcox@gmail.com; Lev Rosenberg, Northwestern University, Evanston, Illinois, USA; e-mail: levrosenberg2024@u.northwestern.edu; Mieraf Mulat, Northwestern University, Evanston, Illinois, USA; e-mail: mierafulat2023@u.northwestern.edu; Jonathan Liu, Northwestern University, Evanston, Illinois, USA; e-mail: jonathanliu2014@gmail.com; Natalie Brewster, Northwestern University, Evanston, Illinois, USA; e-mail: ngbrewster@gmail.com; Eleanor O'Rourke, Northwestern University, Evanston, Illinois, USA; e-mail: eorourke@northwestern.edu; Haoqi Zhang, Northwestern University, Evanston, Illinois, USA; e-mail: hq@northwestern.edu.



This work is licensed under Creative Commons Attribution International 4.0.

© 2026 Copyright held by the owner/author(s).

ACM 1946-6226/2026/4-ART51

<https://doi.org/10.1145/3796525>

techniques through interactive comparison, and help them uncover the domain-specific rationales behind design pattern usage through analogical mapping and reflection activities.

Implications: Finally, we discuss PM’s potential impact on cognitive load, the role of experts today, and a sketch for how LLMs may scale PM. In particular, we envision a world in which educators and builders can develop PM systems that provide intermediate-level engineers a process for learning holistically from open source professional code to gain conceptual expertise, which they can then use to refine AI-generated code. We also provide examples of LLM prompts that can be used to potentially automate aspects of our PM tool-use sessions, laying some groundwork for AI-driven professional development.

In summary, our work contributes to computing education by introducing a scalable and forward-looking tool-building framework and design paradigm for helping intermediate-level learners extract expert-level concepts from complex, real-world code.

CCS Concepts: • **Human-centered computing** → **User studies**; • **Applied computing** → **Computer-assisted instruction**; **E-learning**; **Interactive learning environments**; • **Social and professional topics** → **Software engineering education**;

Additional Key Words and Phrases: program comprehension, professional development, self-directed learning, design patterns, scaffolding, knowledge maps, code instrumentation, analogical transfer, authentic learning, developer tools, sensemaking

ACM Reference format:

Gobi Dasu, Eleanor O’Rourke, Haoqi Zhang, Maxine Whitely, Roxanna Wilcox, Lev Rosenberg, Mieraf Mulat, Jonathan Liu, and Natalie Brewster. 2026. Process Management for Learning from Professional Source Code: Cultivating Experts in the Age of AI. *ACM Trans. Comput. Educ.* 26, 3, Article 51 (April 2026), 83 pages. <https://doi.org/10.1145/3796525>

1 Introduction

While the number of people with basic web development skills has grown [12, 74], novice and intermediate-level programmers still struggle to learn the more advanced concepts and skills required for professional practice. The rise of AI Copilot has only heightened the need for conceptual understanding, as engineers must now prompt **Large Language Models (LLMs)** with high-level architectural inputs and verify and refine the generated outputs. 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 [45, 93, 107, 139]. Unable to apply these, non-experts may create ineffective, unmaintainable, and unscalable applications. Books [107, 129], blogs [139], tutorials [30, 31], courses [83], 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 when building applications [64, 85].

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

¹When we use “intermediate-level developers,” “intermediates,” “non-experts,” “novices,” and “learners” in the context of the tools and studies of this work, we are referring to the same target demographic. We mean individuals who have taken in-person or online courses, but have little experience with doing professional-level work. For a more precise description of our participants and target demographic, please refer to the individual system evaluation sections’ screening criteria.

professional websites [64, 85]. *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 [101, 112, 140]. 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 [85]. In fact, experts do leverage such tools and make sense of complex professional code, make abstractions, and draw connections between code components [88, 89, 111, 143]. They read code strategically [94] and actively inspect visuals and follow dataflow [97, 102]. They organize their knowledge around deep rather than surface-level concepts [6] 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), learners (see Footnote 1) become overwhelmed by files, functions, variables, dataflow, and relationships, not knowing where to focus first. They lack a straightforward process for inspecting, breaking down, and connecting code-level details into high-level visual insights [64, 80, 114]. After observing novices use these web-inspection tools on professional code, we hypothesized that each tool's emphasis (e.g., debugging DOM or tracing events) inadvertently causes learners to *fixate* on certain architectures and ignore others. Such fixations may stem from novices' process deficiencies and a scarcity of process-focused tools. These deficiencies may include difficulty locating meaningful starting points in complex codebases, a narrow focus on surface-level details, and failure to transfer architectural relationships into practice.

To address these shortcomings, we propose **Process Management (PM)**, a framework for building professional development tools that scaffold learners to perform expert-level sensemaking on real-world 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. PM tools provide scaffolds [114, 147] that include intentional, step-by-step prompts and small, ordered activities to highlight key dataflows, comparative tradeoffs [120], 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 [72, 126] and searching for code [38]. Inspired, we investigate whether process management can help bridge the gap between theory and practice in software program sensemaking.

2 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 can 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 may have difficulty breaking down professional codebases, often becoming overwhelmed by multiple components at once. They may fail to recognize architectural elements, identify intuitive starting points, order their sensemaking steps strategically, and connect deeper logic (e.g., in JS, tracing value transformations of user inputs into visible DOM updates, without fixating on event binding; in CSS, investigating visual layout nuances rather than aesthetic styles; in Python, recognizing mixin pattern enactment across real-world code

components). Without a structured and holistic approach [114], they may not execute processes for incrementally building on their knowledge to understand dataflow and real-world architectures.

Surface-Level Focus. When sensemaking, learners may gravitate toward surface-level observations (e.g., two websites use CSS grid, mixins and inheritance afford code reuse, intermediary variables seem unrelated to end-user visuals) and miss subtler differences and tradeoffs [120] 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). Uncovering these conceptually driven fine-grained decisions is challenging when a learner's process consists of cursory observation.

No Transfer. 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 [49] and real-world applications, treating them as separate rather than interconnected. They may 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). Effective transfer requires de-siloing and recognizing shared representations across examples—linking general problems, solutions, and rationales to their specific real-world counterparts [47].

Lack of Auxiliary Awareness. Beyond the obstacles above, learners may neglect factors such as dataflow, 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 use cases that demand selective code reuse). Additionally, they might find it challenging to identify when and which alternatives 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 understanding 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.

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

PM tools aim to help learners overcome these obstacles, perform sensemaking, and achieve a conceptual understanding of key techniques and their structures. While existing learner-centered tools (e.g., Isopleth, Ply) offer localized code visualization or debugging support [64, 85], 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 user-facing components (e.g., DOM updates, comparable websites) or

diagrams (e.g., of design patterns) so they can begin sensemaking quickly about the function and supporting structure of code [111, 114, 143].

- (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 (ZPD)**, we can help them progress through unfamiliar content and professional techniques [138]. In this way, learners can gradually grow and organize their knowledge [6].
- (PM3) *Surface salient parts of code/visuals* so learners focus on the most conceptually meaningful segments rather than getting lost in extensive files [64, 97]. This helps build *awareness* of key components/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 [6, 114], 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 [112, 114].
- (PM6) *Enable comparison* so learners quickly spot crucial similarities and differences across examples, recognizing and analyzing meaningful variations in how techniques are applied [47, 120]. Relevant activities can include assessing alternative techniques and tradeoffs.

To overcome *transfer challenges*, PM recommends that tools prompt users to connect high-level concepts (e.g., architectural patterns, intricate CSS techniques) back to concrete code implementations. 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 [6, 47, 114]. 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 [6, 114, 145].

These process management approaches aim not only to ensure that learners inspect and understand professional code but also to equip them with conceptual knowledge that they can apply in real-world scenarios. Incorporating these approaches, each PM tool instantiates the PM framework in a manner tied to the structure of knowledge being taught (e.g., deep JS value transformations, granular CSS mappings, Python design pattern use). While existing tools and frameworks (e.g., W3Schools, Quintana [114]) may leverage these heuristics, our framework's contribution is to interleave them into a process specifically tailored to intermediate-level developers learning from real-world code.

2.2 Research Questions (RQs) around Exploration and Learning

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

- *RQ1 on Exploratory Interaction.* How do PM affordances scaffold learners' sensemaking? How do PM tool users build knowledge through professional source code exploration?

Specifically, how do the tools shape their navigation style, granular focus points, or articulation of mechanisms or rationale?

- *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?

2.3 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 a 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.

2.4 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 tools²:

- (1) **Scaffolded Exercises (SE)** provides aspiring JS programmers intuitive exercise orderings [114], e.g., starting with visual DOM updates but delving deep into the call stack. SE affords stepwise, interactive investigation of dataflow in JS applications, embeds reflection, and aims to minimize extraneous cognitive load by working with the learners' resurfaced prior knowledge and within their ZPD [138].
- (2) **Knowledge Maps (KM)** guides learners in iteratively grouping CSS-heavy examples by intuitive visual features, initially scaffolding away code details. The system helps learners 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 [120].
- (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.

3 Related Work and Gaps in Literature

We now assess the novelty of PM against related work.

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 [111, 130, 133]. Studies show that novices often struggle

²Our examples for these systems and studies are: (a) xkcd.com/1110/ for SE, (b) Flat Icons (flaticon.com), Hero Icons (heroicons.com), Italic (italic.com), Masterclass (grid-masterclass.webflow.io), Smashing Magazine (smashingmagazine.com), and CSS Tricks (css-tricks.com) for KM (linked from magenta-daffodil-7c0588.netlify.app), and (c) posthog.com for RM.

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

with PC because it involves inferring complex relationships between components and connecting code details with higher levels of abstraction [88, 89, 96, 143]. 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 [17, 35, 48]. Moreover, these learners may have design inertia and must engage in “hidden labor” to reorganize knowledge and change their mental models [33].

Interactive Tools. Well-studied sensemaking and learning tools include code and data structure visualization and tracing tools such as Python Tutor [53, 54] and notional machines [131]. Given complex code, WebCrystal and Whyline help novices explore and ask specific questions about program state or page visuals [26, 79], Tutorons let programmers quickly access inline definitions of complicated syntax within code [60], and code foraging tools, such as Blueprint and Meta, help professionals locate and reuse useful code snippets found on the web [20, 21, 41]. While these tools may clarify isolated code segments or provide specific insights, they may not necessarily prompt a global conceptual understanding of how changes propagate across a codebase or in new scenarios.

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 dataflow, rather than the horizontal ones needed for reading English text [94].

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. This gap indicates a need for structural guidance to proceed from surface-level observations toward deep insights and repeated code patterns.

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 [97, 102]. FireCrystal, Unravel, and Telescope help users specify UI interactions and locate the slices of code responsible for particular features or visual effects within large codebases [65, 66, 103]. Additionally, tools such as Scry [24] and Timelapse [23] 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 [64]. 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 [36, 81, 142] around what to understand, at what level of abstraction, and in what order.

The Need for Scaffolds Towards Professional Practice. Beyond tool use, research advocates for cognitive apprenticeship and authentic exposure [122] to professional codebases, encouraging work on large pre-existing projects to improve confidence [124]. However, while cognitive apprenticeship is valuable, such coaching is not often available to learners, and authoring professional examples into learning resources for novices is not 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 [123], pointing to the need for structure and scaffolding.

Recent approaches in scaffolding show promise. Work on UML-based peer feedback environments, iterative learning for design pattern mastery, and professional code quality checkers point to a future of AI-driven scaffolding [1, 18, 40]. Others propose industry-aligned scaffolding frameworks

to bridge the academia-to-industry skill gap among software engineering students [55]. Still, these works do not sufficiently address the messy, open-ended challenges learners face when exploring real-world code, and they place the onus of determining structured navigation pathways on the novice.

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 [1, 11, 37, 109]. In this context, open source engagement and progressive project complexity have been identified as effective approaches for improving students' ability to apply design patterns [1, 109, 124]. Researchers also stress organizing decision-making processes in structured tabular formats to reduce the cost of capturing and understanding tradeoffs [90]. 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. Research suggests that experts think beyond code syntax, as they also focus on behavioral and functional aspects [68]. To promote such expert practices, we should provide novices authentic experiences, tinkerable/modifiable examples, reflection opportunities, and engaging visuals that lower their cognitive load [132] 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 [39].

Additionally, 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 PC [95]. For novices, recent advances in AI-powered scaffolding systems and intelligent tutors (i.e., leveraging **Retrieval-Augmented Generation (RAG)** [106]) show that automated yet personalized questioning and feedback can strengthen existing scaffolding benefits to better support comprehension and CS education [13, 76, 92, 106]. Still, these semantic frame and AI scaffolding approaches do not seem to offer intermediate-level learners a process for walking through design decisions, mapping concepts to real-world code/visuals, and learning strategically within their ZPD.

Scaffolding 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 [71, 144, 151]. Studies on self-regulated learning recommend guided externalization and metacognitive scaffolding structures [86, 91, 110] and strategies. In AI/ML and data science, sensemaking requires adaptability. Studies stress continuous, tactical, and improvisational comprehension strategies that are both structured and flexible [146].

Research also underscores the role of integrated development environments and interactive representations, such as code diagrams, annotations, documentation, and *in situ* practice problems, in improving navigation and comprehension of unfamiliar codebases [2, 67, 69, 148]. Beyond PC and scaffolding, domain-specific debugging tools and autonomous code repair agents have been shown effective [19, 29]. 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.

Recent Developments That Redouble the Call for PM and Scaffolding. Even 30 years ago, software-realized scaffolding tools provided structured processes for coaching, reflection, goal-setting, and planning in programming and science education [56]. 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 [25]. They also indicate that code complexity metrics often misalign with ease of understanding [42]—an area where scaffolding can potentially help.

In AI-assisted programming, concerns over workflow integration, code validation, and privacy necessitate structured approaches to ensure AI-generated code aligns with domain-specific best practices [150]. In line with these trends, PM scaffolds non-experts towards the holistic, conceptual understanding needed to navigate today's complex and interdisciplinary software engineering challenges (e.g., fixing a class's method, but also reflecting on whether the method is appropriate for the business use case).

PC, Scaffolding, and Professional Development Precautions and Nuances. Adaptive learning systems in software engineering face challenges in engagement [10] and potentially scaffolding. Research on programming misconceptions [137], metacognition [91], and mental models [61] underscores the need for consistency and structure in modern tools and interventions. Structured decision-making reduces errors. Eye-tracking studies show that programmers employ structured scanning patterns aligned with their conceptual schemas [78]. Research recommends trajectories of activities, each custom to specific levels of abstraction and learning goals around program structure, behavior, and function [75]. Some recommend multimodal engagement, such as interactive visualizations and sketching [127], while others caution against excessive context switching that hinders deep focus [125].

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 [99]. Similarly, research on online resource usage strategies by novice programmers highlights the risks of code copying and the need for critical evaluation training [4]. Broader reviews on software engineering education and professional skill gaps stress communication, self-reflection, software architecture/design, and configuration management [46, 52]. While literature is rife with best practices, tools that operationalize them end-to-end for intermediate-level learners studying and effectively leveraging real-world code seem scarcer. This gap paves the way for structured PM tools.

Authentic Learning in Computing Education. Authenticity is increasingly recognized as a key factor in computing education, influencing students’ motivation, engagement, and perceptions of programming [136], 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 [9, 62]. For instance, while block-based environments can help novices, they can potentially impact later confidence in text-based programming [98]. Studies have shown that student perceptions of authenticity are nuanced and shaped by multiple aspects of the learning environment. DiSalvo [34] 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 [141] 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. [87] 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. [16] argue that authenticity is constructed through meaningful participation rather than being pre-defined by instructors, with individual characteristics influencing sensitivity to perceiving authenticity. Moreover, Schriegl et al. [119] provide a comprehensive model identifying key dimensions of authentic learning environments, including real-world connections, disciplinary practices, and personal contextualization.

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 underscores 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.

A Call for PM, Reflection, Comparing/Contrasting, and Analogical Transfer. The preceding sections reveal three themes: (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 and pattern implementations [77, 115], our process management framework builds on research showing that conceptual transfer requires active articulation and contrasting diverse examples [44, 59, 120], while drawing from cognitive scaffolding principles [114], knowledge organization [6], and analogical transfer theory [47]. These works together promote multiple views of artifacts, reflection, and scoping. However, unlike existing tools that focus on specific techniques (e.g., Ply for helping learners identify visually impactful CSS [85]) or static pattern definitions [45, 57], PM provides structured guidance for holistic sensemaking across entire codebases, enabling learners to construct [58] 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, dataflow, and domain-specific, multistakeholder design rationale. Next, we review needfinding cases that motivate precautions against fixation during sensemaking.

4 Needfinding through Pilot Cases of Novices’ Sensemaking of Professional Code

Existing tools for learning from professional codebases often focus on specific programming challenges rather than holistic sensemaking spanning entire codebases. To explore this gap, we conducted needfinding pilots where novices explored professional web applications using either CDT, a standard debugging tool, or Isopleth, a visualization tool for asynchronous JavaScript [64]. Together, this comprised ~70 hour of observation of 21 post-secondary students, followed by a detailed analysis of select cases. We examined how these tools shaped exploration strategies, learning outcomes, and conceptual understanding of complex code structures. Based on the observations, we reasoned that tool affordances strongly influence user strategies—CDT may promote a focus on DOM manipulation, while Isopleth may emphasize asynchrony, leading novices to fixate on one architectural feature (e.g., ones most visible at the *surface-level*) while overlooking others (e.g., ones requiring deeper investigation). For instance, users of CDT often traced visual features back to the DOM (e.g., image loading) without investigating underlying event flow, while Isopleth users followed asynchronous call graphs but missed visible page behavior and DOM updates. Additionally, these findings pushed us to consider the role that JS familiarity plays in novice sensemaking capabilities (of *complex* professional code).

These insights seeded our ideas around PM scaffolds that actively guide learners through a structured, holistic exploration of professional code. We identify two sensemaking challenges: (1) *Tool-architecture fit*, where tool affordances naturally emphasize certain architectural elements over others, and (2) *Tool-strategy fixation*, where novices unconsciously adopt exploration behaviors shaped by the tool rather than developing their own robust strategies. For example, Isopleth users focused on callbacks and relationships but often failed to read update functions responsible for DOM changes, while CDT users rarely examined event binding nuances, even when those were

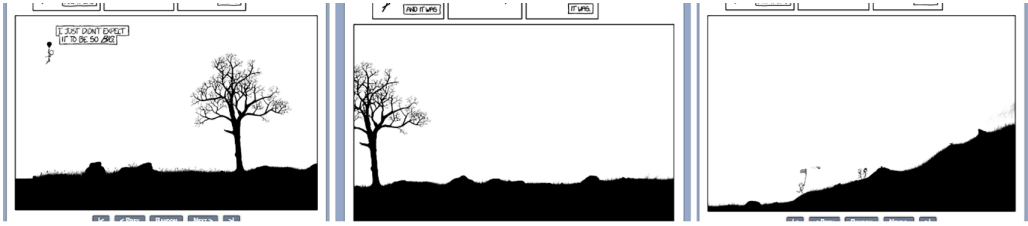


Fig. 1. The XKCD Click-and-Drag example presents a draggable map of a comic world. In its implementation, mousemove events trigger the system to calculate a new page position for the user, which in turn is used to update the CSS positioning of a larger map div. The mouse position changes are also used 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.

central to the architecture. PM aims to address such situations where learners seem less *aware* of aspects not highlighted by their tools. PM provides structure and guides learners in executing expert-level sensemaking strategies designed for knowledge retention or holistic concept transfer.

5 SE, a PM Tool for JS Value Transformations

5.1 Motivations and Initial SE Prototypes

The concepts of fit and fixation from our needfinding 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 of interconnections and dataflow, such as variable transformations that drive visual updates. In our pilots, when novices tried to inspect an interaction like dragging a map in the XKCD Comic (Figure 1) via CDT, they stumbled through scattered function calls (e.g., *X position*, *tile center coordinates*, *offsets*), set breakpoints without a clear strategy, fixated on local changes, and failed to connect mouse events to deeper dataflows or DOM updates, leaving them ill-equipped to reconstruct how the map reloads, or to extend/debug/enhance its functionality, and so on.

Problem. As we have seen, in the domain of JavaScript, existing tools like CDT (which surface everything at once) 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 and files impact the DOM. This leads them into a chain of confusing function calls where variable values, at the *surface-level*, seem loosely semantically related to the system’s functionality/domain. For example, in the XKCD Comic (Figure 1), learners struggle to structure their sensemaking, concretize dataflow, and connect a map drag gesture (async event trigger) with the deep variable transformations of mouse positions and tile centers that trigger image reloading (the final DOM update) [64].

Embedded in this thought space, we iteratively developed SEs—a process management system that incrementally scaffolds learning about variable value transformations (i.e., one variable affects the value of another, which affects another, etc.). Learners demonstrate an interaction on a website they want to learn about (e.g., dragging on a map loads new parts of the XKCD comic map as in Figure 1). Then, SE provides an auto-generated sequence of exercises that guide learners through exploring and learning about code that produces a visual effect on the front end. Through early designs, we explored how SE’s structure could help learners make observations, break down complex code, map code execution to visual changes, and incrementally build conceptual models. Early prototypes 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.

5.2 How SE Enacts PM: Ordered Exercises, Live Variable Values, Reflection, and Resurfaced Prior Notes

SE supports PM in two key ways: (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 and conceptually chunk functionality backwards through the call stack.

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 2) begins with a user (1a) observing a DOM-level effect, (1b) inspecting the live values of the driving 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 scaffolding heuristics because their suggestions around intuitive entry points, articulation, bound setting, diverse representations, and knowledge organization are known to improve inquiry and sensemaking [114].

PM Solution. SE (Figure 3) is a PM system that provides learners with an ordered set of SEs for sensemaking, where learners can progressively build on their intuitive, prior understanding to advance further understanding. It leverages *reverse call stack sequencing*, where learners start with exercises covering *visually intuitive DOM updates* (PM1) (Figures 2, Step 1a and 3(a)) and systematically trace backward from the surface-level to understand underlying value transformations deep in the call stack⁴ (i.e., how code changes internal variable values that eventually affect the DOM). By chunking each step into an exercise, SE aims to *reduce cognitive overload* (PM4) and supports learning within one’s ZPD [138]. Each step surfaces only the relevant variables and transformations (Figure 2, Steps 1b/2–4). The progression also 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 [114].

Unlike traditional debugging tools, SE provides ordered, goal-driven exercises that enable *value transformation exposure*, the study of changes in variable values across code components. SE surfaces *interactive live variable values* (PM5) (Figure 3(c)), *salient code snippets* (PM3) (Figure 3(f)), and hover-over syntax explanations (Figure 3(g)) that intend to *scaffold away complexity* (PM4). Each exercise includes structured prompts promoting *reflection* (PM8) and note-taking [114] (Figure 3(d)) to support incremental sensemaking of variables. These notes resurface across the ordered exercises (Figure 3(b)), aiming to help learners link *prior and current knowledge* (PM2), reinforcing their architectural understanding of dataflow and relationships (Figure 2, Step 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 at a higher level, from general conceptions of dataflow to specific variable value transformations. 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 2, Steps 1b/1c and 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

⁴The original (non-reversed) execution orders start from the bottom of the call stack, stacking on new function calls, and moves up to the top of the stack.

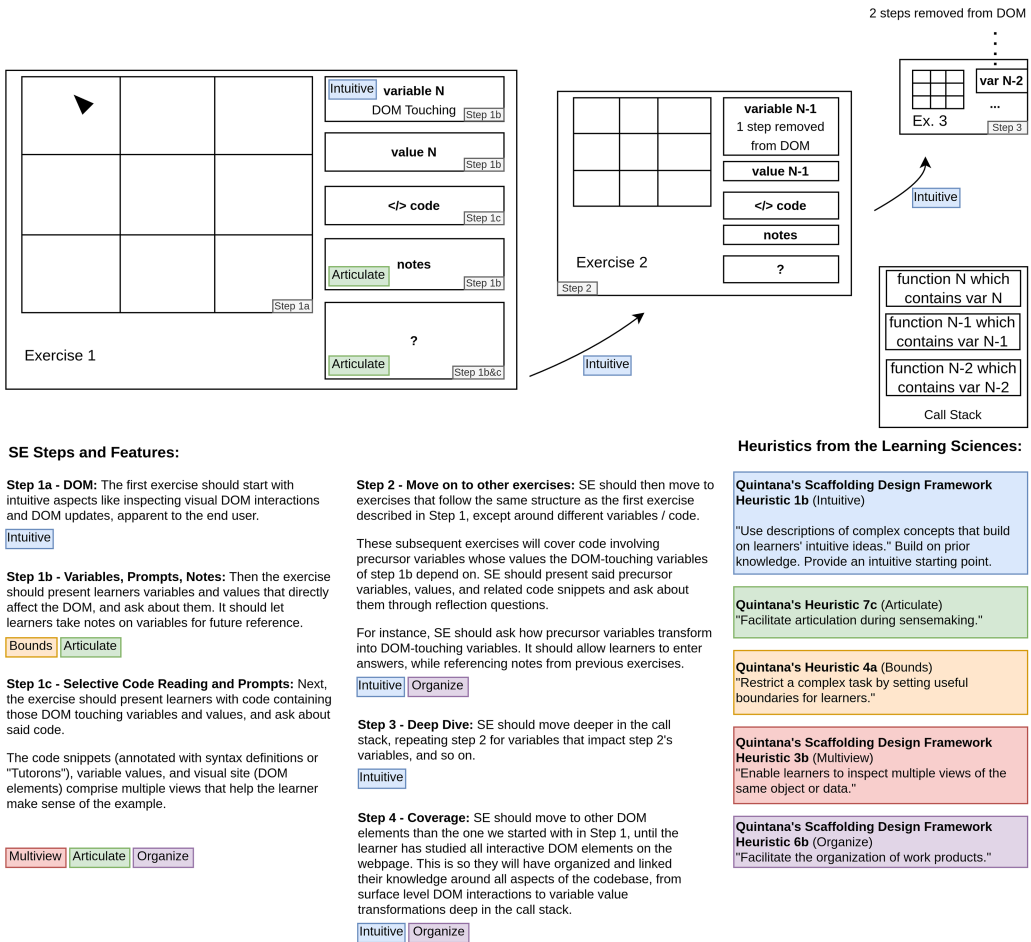


Fig. 2. Here we present a conceptual diagram of how SE works and enacts Quintana's heuristics, as well as several PM heuristics. The system presents exercises to the learner, starting with intuitive (PM1) DOM interactions (Step 1) and moving to variable value transformations deep within the call stack (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).

definitions (Figure 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 [114]. Instead, SE auto-tracks and surfaces live variable updates, both local and global (Figure 3(c)). This allows users to observe transformations dynamically as they interact, reinforcing their understanding of dataflow 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

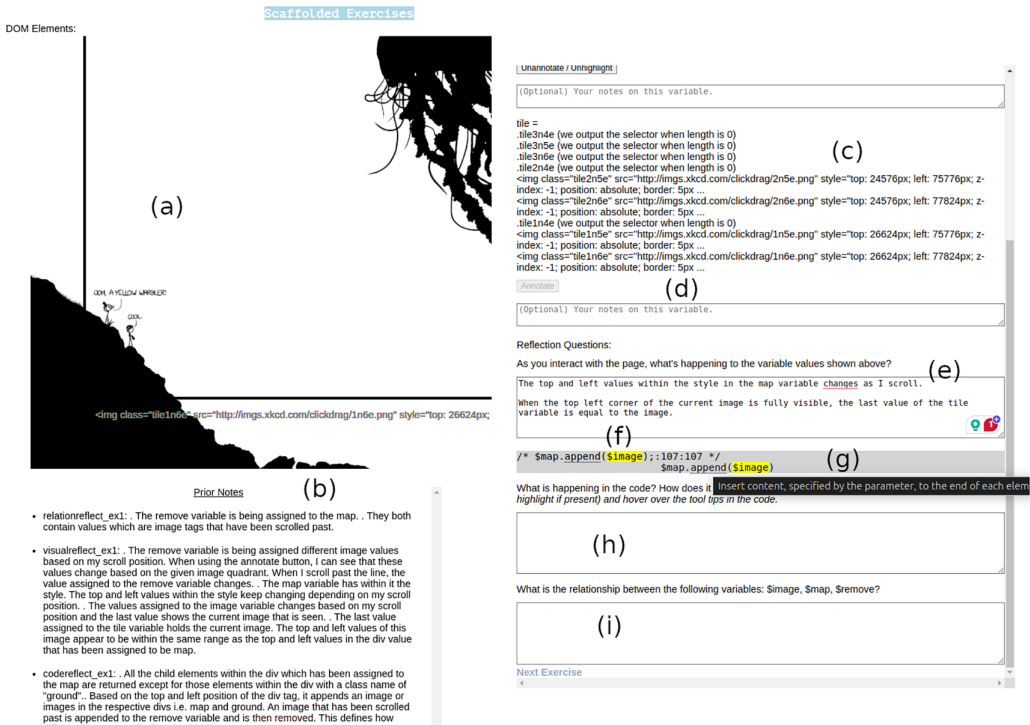


Fig. 3. SE prototype on XKCD: It features (a) annotatable elements on the page, (b) resurfaced prior notes, (c) live variable values and DOM element tracking, (d) variable-specific notes, (e) reflection questions on DOM elements and variable values, (f) code views where live variables can be highlighted, (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.

understanding of full-stack architectures (Figure 2, Steps 2–4). While we have described how SE enacts PM, Figure 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 [114].

5.3 SE System Implementation: Reverse Call Stack Sequencing

We now review techniques for automatically⁵ 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 stack.⁶ To enable live variable tracking and

⁵There are currently a few hard-coded 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.

⁶Equivalent to reverse function run log order, in non-async cases.

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, with details following.

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, modifies⁷ (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 ensuring 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.

⁷The first exercise's code/node already modified the reference, so this refers to yet another node that modifies the reference DOM object.

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 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, users.⁸

Interface Implementation. We also developed the UI for SE (Figure 3). We automated⁹ 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 3. An instructor can likely set up SE on a new example in ~2–4 hours, and details on customization and workload can be found in the Appendices D–F.

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.

5.4 Study Methods and 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 non-experts, 5 using our system and 5 using a control. We measured the effectiveness of the PM affordances, such as SE's incremental exposure through ordered exercises.

Participants were recent post-secondary graduates and students involved in Computer Science, IT, engineering, and so on. To recruit them, we sent public blasts in tech-focused Slack organizations and channels.¹⁰ Participants were clearly informed that the study was educational and voluntary, with the right to opt out at any time. Participant data were anonymized. 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.

Participant Screening. To ensure we were testing on non-experts 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

⁸We 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.

⁹Our exercise generator is general, but we passed in some XKCD-specific variables and elements to handle skips, unfurling, and aesthetics. These can potentially 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.

¹⁰Recruitment: 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.

conditions) needed to have (a) written 200–5,000 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 provide a coherent explanation—regardless of correctness—would have raised a red flag (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 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 goals to consider, such as inspecting page visuals, understanding variable values, linking code with visuals, and investigating relationships between variables. The goals cover what SE aims to highlight content-wise, but in a static rather than process-managed manner. We posit that providing this information to control users makes the comparison fairer and a true test of PM, as the SE condition participants had only a process advantage (what we are testing) rather than an information advantage. 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. Moreover, participants in both conditions could search online (though such content, even if structured, would not be example-specific).

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, dataflow, and intermediary variable values. It asked (a) *What user inputs change the DOM?*; (b) *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 1 for SE).

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) 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.

Table 1. SE Pre/Post-Test Learning Outcomes Grading Rubric

Num	Learning Outcomes Rubric Item
1	<i>\$map</i> has images (multiple HTML <i></i> instances) appended/loaded onto it.
2	Each image tile has a unique <i>name</i> — <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</i> /tile 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 3×3 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" ("mousemove" 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.

Each item corresponds to one point. Partial credit details omitted for brevity.

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 use the features available to them fully, especially when stuck, to strictly follow the exercise order as applicable,¹¹ 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"), reminding them of

¹¹Experimental condition users of SE and RM were expected to proceed linearly through activities. RM users were expected to map step-by-step from theory to practice, following reference links as they are helpful.

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 code components.

Interventions made during the pre/post-test to ensure accurate assessment included reminding participants to explore the examples' UX thoroughly, prompting them to share more about how/why the example works ~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 relations and dataflow).

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 (as applicable,¹² automatable using interface prompts), to use variable notes and refer to prior notes upon starting a new exercise (as applicable, automatable using interface prompts), and to use affordances they had not 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 (as applicable, rules can trigger learners to use these features when they are not doing so).

Future work can explore how LLMs can potentially automate interventions in production settings [104, 105].

Scoring and Inter-Rater Reliability (IRR). To evaluate whether SE can help learners understand JS variable value transformations, we coded the pre- and post-test according to a rubric (see Appendix H for more details on rubrics) of reachable learning outcomes around variable value transformations (e.g., how mouse positions transform into tile centers which influence image loading) (Table 1). Points were summed to obtain pre- and post-test scores, which were used to measure knowledge gains (post-test 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-test and post-test to ensure consistency. IRR was measured,¹³ and substantial agreement was reached for each system evaluation. We report (a) percentage agreement for most codes (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 (see Footnote 13), and (c) the remaining codes that required case-by-case discussion on all data (0 for SE).

5.5 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

¹²Depends on condition, N/A for control here.

¹³Our IRR process followed standard qualitative practices. 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. Circumstances requiring case-by-case discussion on every code (0 in SE, 3/80 in KM, 4/31 in RM) were excluded from IRR metrics, as they reflected collaborative resolution. The tests of one KM and one RM participant were coded only by the experimenter due to time/technical constraints, using established rubric standards. 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. In RM, potential concerns regarding cross-influence between mixin and MV* sections were mitigated by rubric clarity and high IRR.

Table 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

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 nine 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

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. We chose these learning goals based on our initial pilots.

of 12, while control users had an average gain of 2.9 ($\sigma = 2.5$) out of 12 (see Table 2¹⁴ and Figures 4 and 5 for participant-specific data points).

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 ($r=0.76$, $rb=0.96$).¹⁵

¹⁴For “discussed live variable values” and “fraction of vars in code discussed” (Table 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.

¹⁵To quantify effect size and uncertainty, we also computed Cliff’s delta ($\delta = 0.96$; 95% bootstrap CI: [0.760, 1.000]); a two-sided permutation test on $|\delta|$ yielded $p = 0.0159$. A Bayesian probability-of-superiority estimate gave $P(X > Y) = 0.962$ with a 95% HDI [0.886, 1.000] (equal-tailed 95% CrI [0.862, 0.999]).

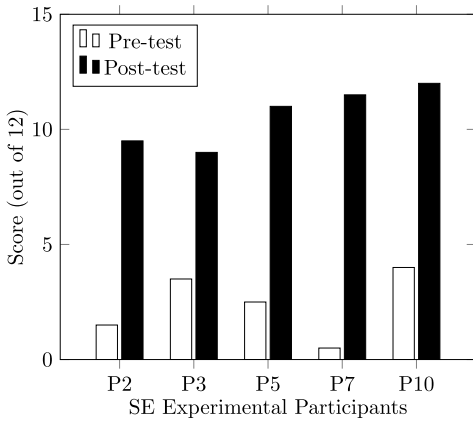


Fig. 4. Comparison of pre-test and post-test scores for SE experimental users.

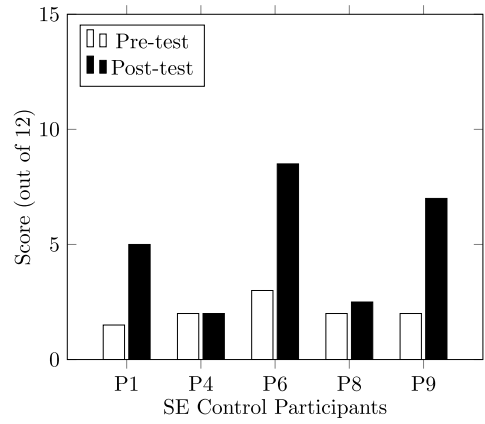


Fig. 5. Comparison of pre-test and post-test scores for SE control users.

Although this difference was not hypothesized *a priori*, we report effect size with uncertainty as exploratory evidence to motivate future confirmatory work. To assess sensitivity and robustness, we treated simulations as robustness checks: in 10,000 simulated 5×5 experiments at the observed effect size, estimated power ranged from 0.843 to 0.994 across several commonly used distributions for independent test score gains, including Normal (0.886 [0.880, 0.892]), Gamma (0.864 [0.857, 0.871]), Weibull (0.843 [0.835, 0.850]), Log-normal (0.855 [0.848, 0.862]), and bootstrap with replacement (0.994 [0.992, 0.995]) assumptions (ranges contain Clopper-Pearson's CIs).¹⁶ All assumptions yielded power > 0.80 . A small- n sensitivity check further indicates that, with $n = 5$ per group, $\delta \geq 0.9$ is typically required to exceed 80% power; our observed effect exceeds this threshold, but CIs are wide. Future evaluations with larger in-the-wild samples, production-grade PM systems, and *a priori* power analyses will tighten uncertainty and strengthen generalizability claims.

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 rated¹⁷ 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 dataflow between intermediary variable values. Given the data, we can now answer our RQs on exploration and learning with SE.

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

¹⁶Robustness check: To assess whether our results are robust to distributional assumptions, we simulated 10,000 5×5 experiments using distributions fitted to the observed data. Since the underlying distribution is unknown, we tested several commonly used distributions for independent test score gains: normal, gamma, Weibull, and log-normal. To meet the strictly positive support requirements of gamma, Weibull, and log-normal, we shifted all test score gains upward by a constant ($\text{number of codes} + 1$), a transformation that preserves rank order and does not affect Mann-Whitney U outcomes. We also performed bootstrapping with replacement. For each distributional assumption, we computed the proportion of simulations yielding $p < 0.05$ at the observed effect size, providing estimated power as a sensitivity metric.

¹⁷We 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.

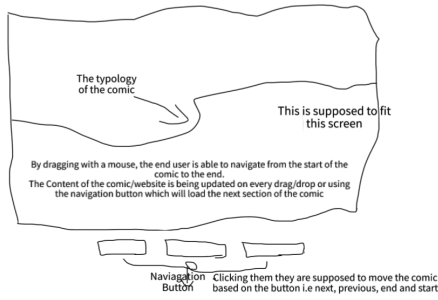


Fig. 6. This diagram shows P7's understanding of XKCD before SE use.

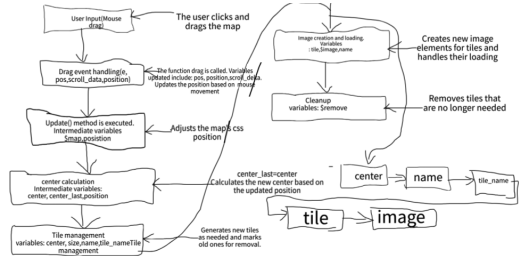


Fig. 7. This diagram shows P7's understanding of XKCD after SE use.

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-test (Figure 6) and post-test (Figure 7) diagrams, which demonstrate an improved grasp of variable value transformations (e.g., *centre* → *name* → *tile* → *image*). Participants also found chunked 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 2. The table also shows how nearly all exercises 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 dataflow between variables. P1 described leveraging “the debugging tool to navigate different functions” and variables, focusing on line-by-line reading but missing the bigger picture of data transformations. P4 emphasized how “isolation of variables” when inspecting makes what happens in the code clear, but still found it challenging to integrate this understanding fully. They noted that though they “did some intensive inspection [and] analyzed the code,” they found the experience a “bit hard,” suggesting they found the code complex.

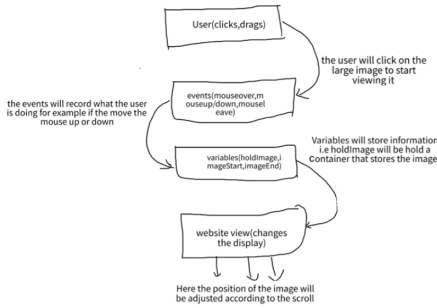


Fig. 8. This diagram shows P8's understanding of XKCD before control/CDT use.

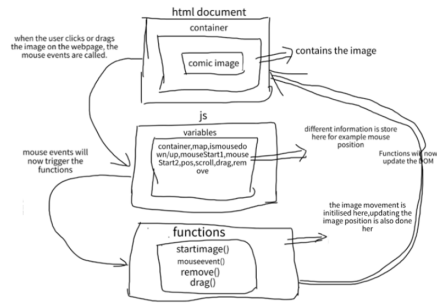


Fig. 9. This diagram shows P8's understanding of XKCD after control/CDT use.

P8 acknowledged that their understanding evolved with practice, saying that at first, they “*could not think of many variables*” that might be used. After looking at the code, they “*realized there were many items*” they had not considered, suggesting they could at least learn what it is they do not know. Still, their pre-test (Figure 8) and post-test (Figure 9) show limited progression in terms of understanding of variable value transformations (e.g., no clear delineation of the *centre* → *name* → *tile* → *image* transformation even by the post-test). In general, control participants described a more event-focused approach, *unaware* of deeper value transformations. While P9 reflected that “*dragging the map*” helped them understand the flow of different values, they noted that it was challenging to connect these insights directly to the code without annotation. Ultimately, they mainly focused on events and images, not intermediary variables. Collectively, participants’ efforts to comprehend the entire codebase were admirable but often led to fixating on specific variables (e.g., the image) or events rather than following the entire dataflow between 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.

Experimental Participants’ Learning (RQ2). The experimental participants appeared to demonstrate learning gains, especially in understanding the *position* → *centre* → *tile/name* → *\$image* dataflow. 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-test (Figure 10) and post-test (Figure 11) accordingly show improved understanding of the *position* → *image update* dataflow in the XKCD example. Generally, experimental condition participants connected interactions to the underlying code flow, with P7 (Figures 6 and 7) adding that the drag function updates the position, triggering the “*update()* function” to load and remove tiles as needed. These achievements demonstrate how SE users successfully uncovered the relationship between *position*, *centre*, and *tile* updates after hands-on tool use. The SE experience seems to have taught users the value of a gradual step-by-step progression when building understanding and seems to have brought them deeper conceptual and process clarity [27]. For instance, they did not just patch in new information (e.g., images are chosen

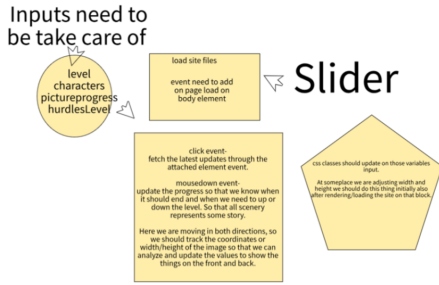


Fig. 10. This diagram shows P5's understanding of XKCD before SE use.

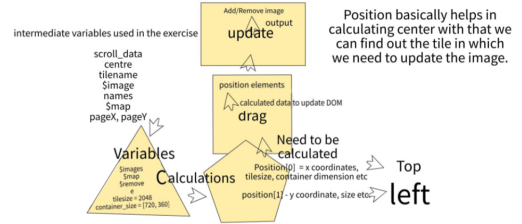


Fig. 11. This diagram shows P5's understanding of XKCD after SE use.

based on the tile name), but appeared to shift their mental models around coordinate systems and dataflow (e.g., it is not just mouse moves shifting content; rather events trigger dataflow that leads to image loads).

Control Participants' Learning (RQ2). Despite some improvements by the post-test, control participants generally struggled with grasping intermediary variable values and dataflow. 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 “*mouseX 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.

5.6 SE Discussion: Practical Learning and Technical Contributions

Our study indicates that SE can improve novices' grasp of how dynamic UI behaviors arise from variable value transformations often invisible to learners. SE users outperformed control users on post-tests and could clearly explain how the “position,” “centre,” and “tile” variables interact to load new map images. SE automates a reverse-engineering workflow by injecting lightweight instrumentation and capturing live variable values. It then generates focused and interactive exercises [23, 24] that step backward from visible DOM changes to their causal roots—building on insights from prior visualization and source-exploration tools (cf. CDT, WebCrystal, Whyline [26, 79, 97, 102]) while explicitly ordering activities [114]. We posit that anchoring each exercise in an observable effect kept learners oriented, reduced cognitive load, and helped them build mental models of variable-to-variable dataflow (*position* → *centre* → *tile* → *\$image*) rather than fixate on isolated line-by-line execution or get lost in daunting complexity [88, 89, 96, 143]. By guiding exploration within learners' zones of proximal development [138], linking their prior knowledge, and supporting reflection (RQ1 result), SE illustrates how process-managed, backward segmentation of the call stack can facilitate conceptual code-based learning of value transformations (RQ2 result). While larger-scale in-the-wild evaluations would strengthen our claims, our current findings may

still inform the design of scalable professional development tools that leverage chunked, intuitive, and outcome-anchored learning progressions [114].

6 KM, a PM Tool for Granular CSS Techniques

Like SE, KM is an interactive process management tool. It supports structured comparing and contrasting across multiple professional CSS examples to prime deep conceptual learning [120]. While SE focuses on dataflow exploration within a single example, KM aims to help learners identify granular visual layout differences across sites and map them to underlying CSS code, fostering transferable conceptual knowledge of production-grade UIs. In early KM pilots, we found that novices exploring real-world CSS examples (e.g., fixed vs. dynamic column layouts) tinkered with various sites without a clear process for transfer, fixated on superficial cues (e.g., font color), missed critical conceptual and sub-property distinctions, and struggled to articulate several tradeoffs or the design intent driving complex responsive layouts with multiple techniques and intricacies (e.g., changes under different viewport sizes). These pilots inspired our problem statement and PM-specific KM formulations.

Problem. Learners *struggle with knowledge transfer* of professional CSS techniques because they may view examples in isolation, lacking *awareness* of other examples, their nuances, and knowledge of alternative implementations against which to compare. Hence, they might miss subtle but architecturally significant distinctions. For instance, a fixed four-column layout (e.g., *grid-template-columns: 1fr 1fr 1fr 1fr* as in Italic from Figure 15) may seem indistinguishable from a dynamic layout (e.g., *grid: auto-fill* as in Hero Icons from Figure 14) 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 [6] of grouping, contrasting, and tinkering with the code of examples, learners, even when equipped with inspection tools, may not achieve significant coverage of the granular visual outcomes and their underlying CSS, or link the two.

6.1 Design-Based Research Iterations of KM

We iteratively designed 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, and so on. 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, highlighting, tinkering, feature-to-code matching, and structured reflection, challenges around transfer, particularly of granular CSS sub-properties (e.g., *grid-template-columns: 1fr 1fr*, not just *grid*) persisted. This pushed us to think more about designing a structured process for learning from diverse examples and heeding details.

6.2 How KM (Latest Iteration) 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 auto-fill*. 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 [50, 70]. To address this, we

reinvent the system through process management via prompts, activities, and an integrated map-process complex, in which learners iteratively group and contrast examples to better understand critical CSS details (Figure 12). We argue that KM can now enable learners to transfer nuanced CSS concepts from training examples (Figures 13–15) to new applications (presented later in Figure 26).

System Overview. KM (Figure 12) 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, with the goal of 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 in a manner custom to CSS, and prepares learners for transfer.

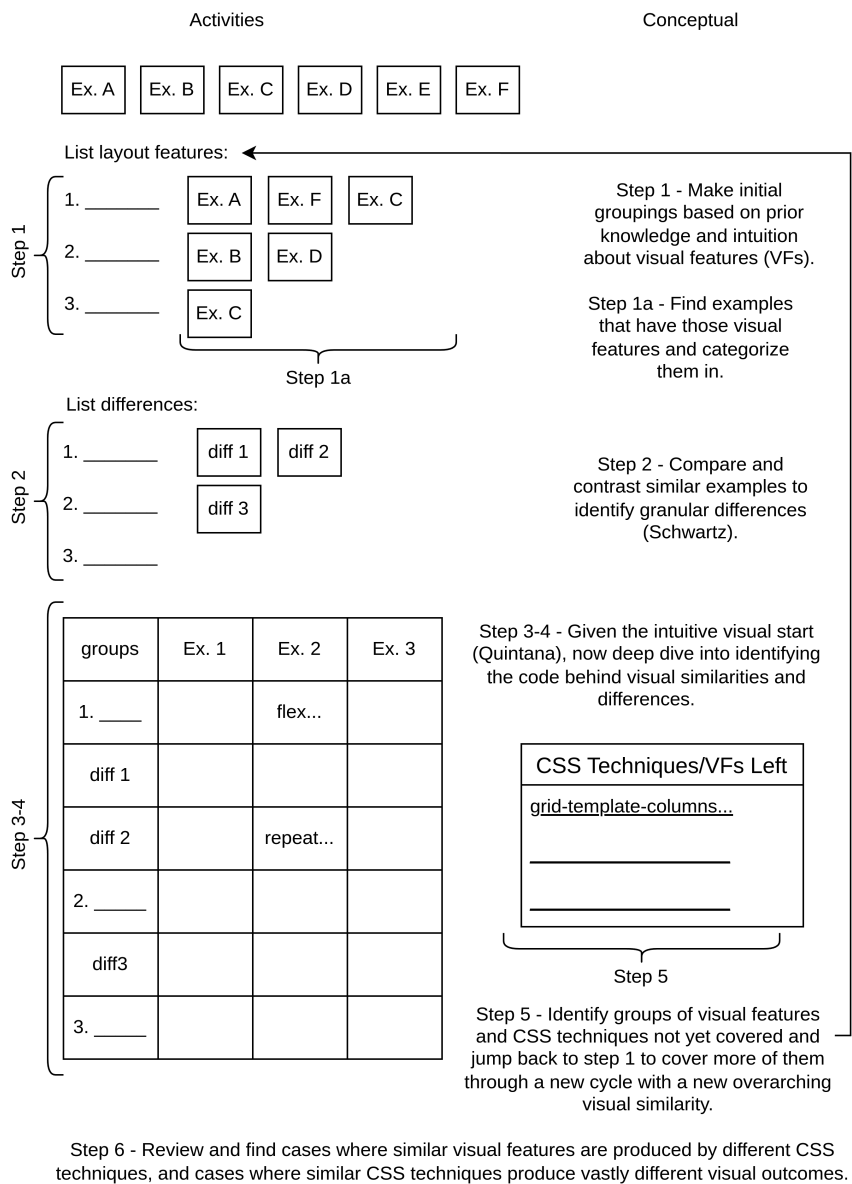
PM Solution. To begin this process, learners compare pairs of websites, focusing only on page visuals without looking at the code (Figure 16). This aims to reduce complexity and their *extraneous cognitive load* (PM4). They identify one *intuitive visual layout feature* (PM1) common to multiple websites (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). They then progress beyond the surface-level, *comparing and contrasting* (PM6) (Figure 17) examples within that visual layout group 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 18) serve as *prior knowledge* for future steps (PM2) and form the outline of their knowledge map.

Attuned and primed to learn from code, learners next examine the CSS for each example to find individual CSS properties and *map* (PM7) them to the visual features they yield, whether shared across examples (Figure 19) or distinct (Figure 20). They can refine their understanding using *interactive* (PM5) features such as property toggling and value tinkering (Figure 23(g) and (h)). At this stage, KM users may also speculate why said CSS properties were chosen. They repeat this process to ensure full code coverage (Figure 21).

At the end of these cycles, learners *reflect* (PM8) on their evolving “knowledge map,” consolidating insights from their comparisons (Figure 21). During this reflection, they analyze patterns such as how similar code can lead to different visual features or how different coding techniques can achieve the same visual outcome (Figure 22). 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. Learners who complete sufficient cycles of KM across diverse websites should understand a breadth of existing techniques, both broad and subtle, and why professionals use them in specific contexts.

Having introduced the KM process, we highlight specific affordances within it that facilitate PM:

Comparison. After the first step (Figure 23(a)) of finding a common visual feature across a group of websites, learners use KM’s side-by-side comparison tool (Figures 16 and 23(c)) to analyze pairs (Figure 23(e)) within their grouping. This can help them identify subtle differences between sites with similar visual features [120], better priming them to learn concepts, i.e., code in this case. This approach builds on their visual intuitions [114]. For example, when comparing the layouts of Flat Icons and Hero Icons (Figure 14), 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



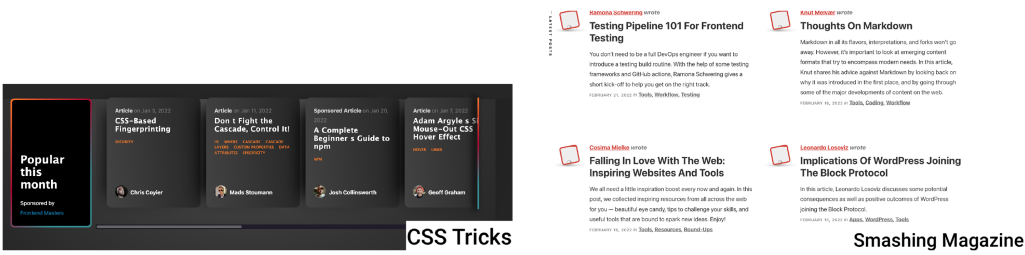


Fig. 13. KM Examples: Learners should recognize examples with (A) responsive layouts without clear table-like structures (e.g., CSS Tricks, Smashing Magazine); (B) layouts with specific overflow behavior (e.g., CSS Tricks).

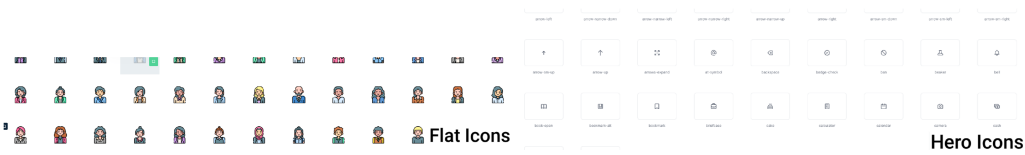


Fig. 14. KM Examples: Learners should also recognize examples with (C) layouts where equally sized content overflows into new columns dynamically (e.g., Flat Icons, Hero Icons).

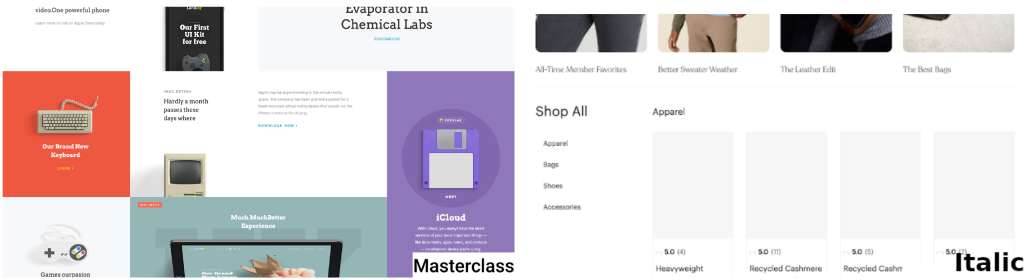


Fig. 15. KM Examples: Learners should also recognize examples with (D) layouts following a fixed table-like structure with sidebars and assorted rectangular sections (e.g., Italic, Masterclass).

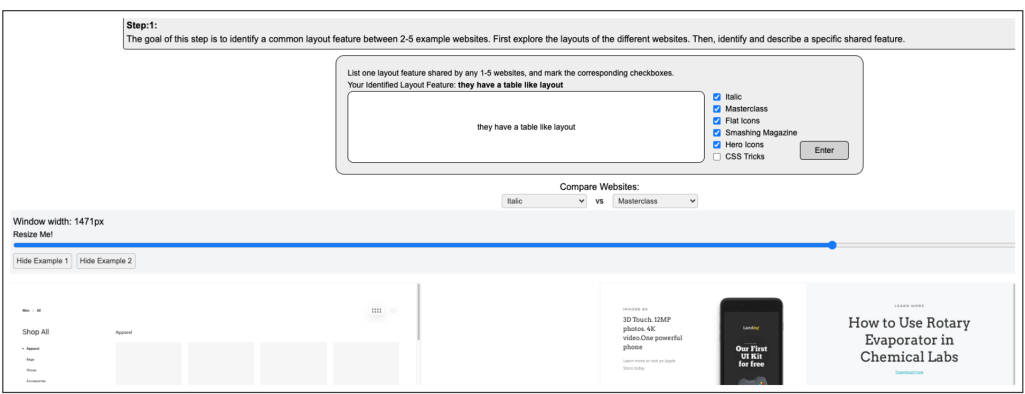


Fig. 16. Step 1—Identify common visual layout features (sample data).

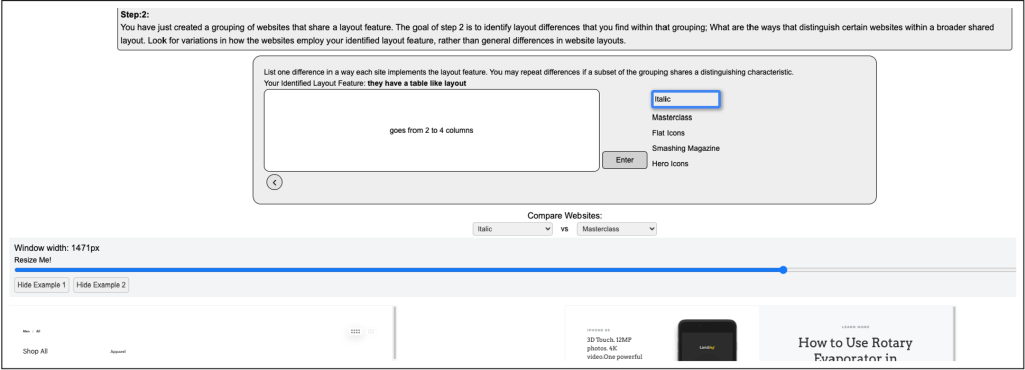


Fig. 17. Step 2—Find differences within group (sample data).

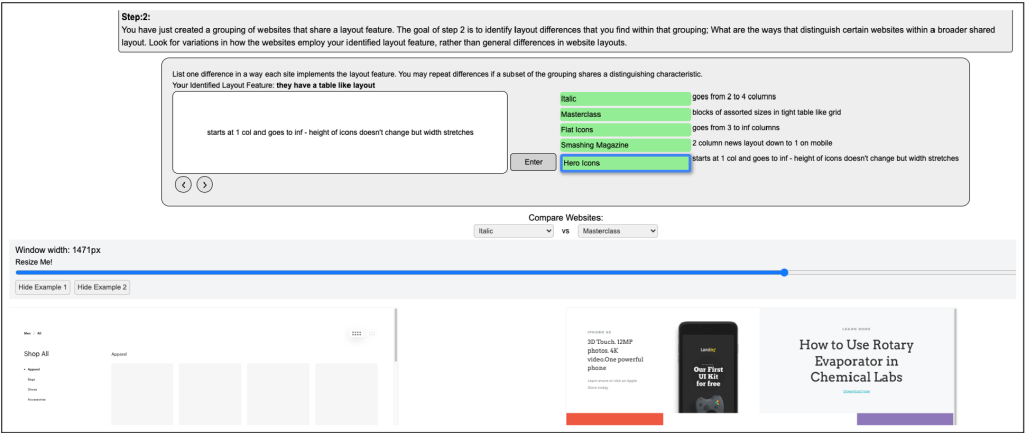


Fig. 18. Step 2—Collect all differences within group (sample data).

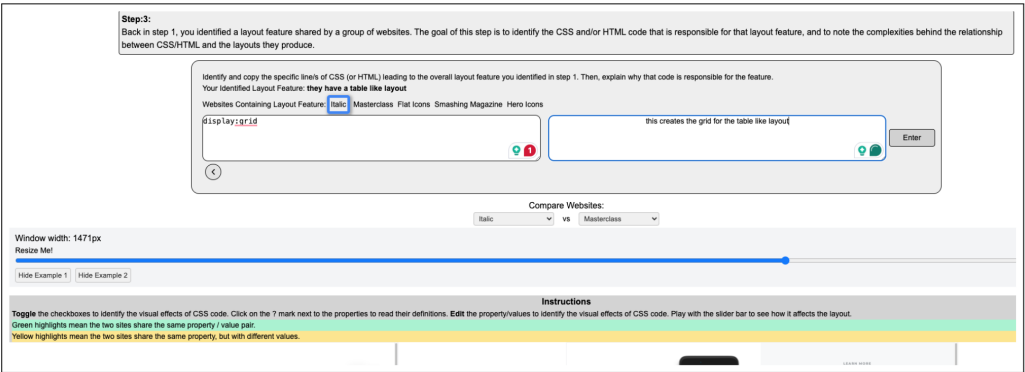


Fig. 19. Step 3—Find code behind common visual layout features, and the rationale behind that choice of code (sample data). More is described in prose.

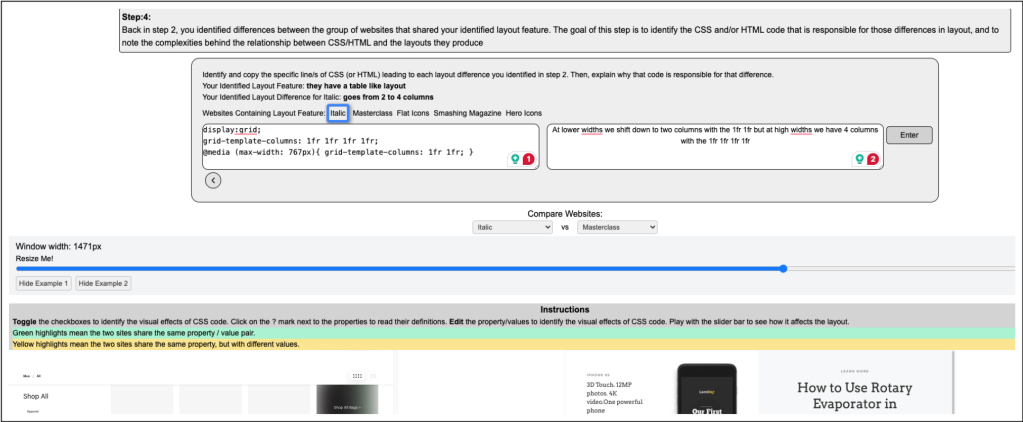


Fig. 20. Step 4—Find code behind granular differences, and the rationale behind that choice of code (sample data). More is described in prose.

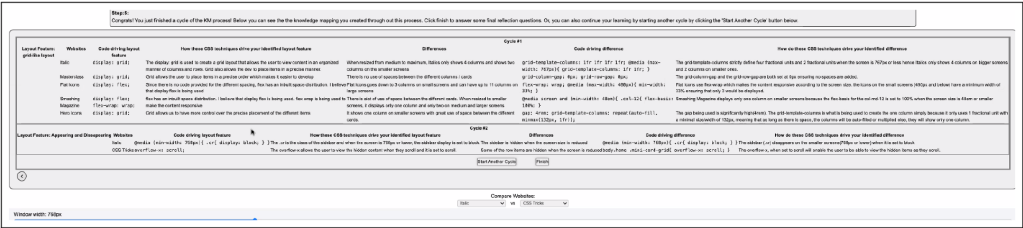


Fig. 21. Step 5—Review cycles of granular features and code to spot missing details (P4 data). More is described in prose.

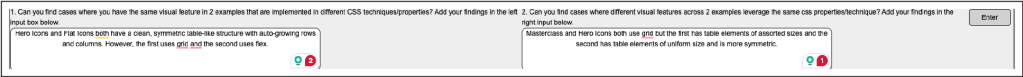


Fig. 22. Step 6—Find cases where different CSS techniques produce similar visual features and cases where similar CSS techniques produce vastly different visual outcomes (Sample Data). More is described in prose.

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 clustering visual outcomes across examples, learners implicitly predict which may share similar code, but through comparison and mapping, they may improve the transfer of their understanding of said visuals and underlying CSS [6].

Mechanistic Details for Overcoming Surface-Level Focus When Learning from Code. To overcome fixation on superficial visual features or syntax, KM’s PM scaffolds guide learners to compare examples and update their knowledge map with insights into the CSS driving granular visual layout outcomes. We argue that chunking the process [114], by starting with intuitive visual comparisons to tease out differences [120], and then prompting learners to tinker, reflect, and map code and rationales [145], enhances learning by providing a structured pathway for understanding relevant CSS details. By actively modifying CSS properties—in line with Quintana’s “multiple views” and

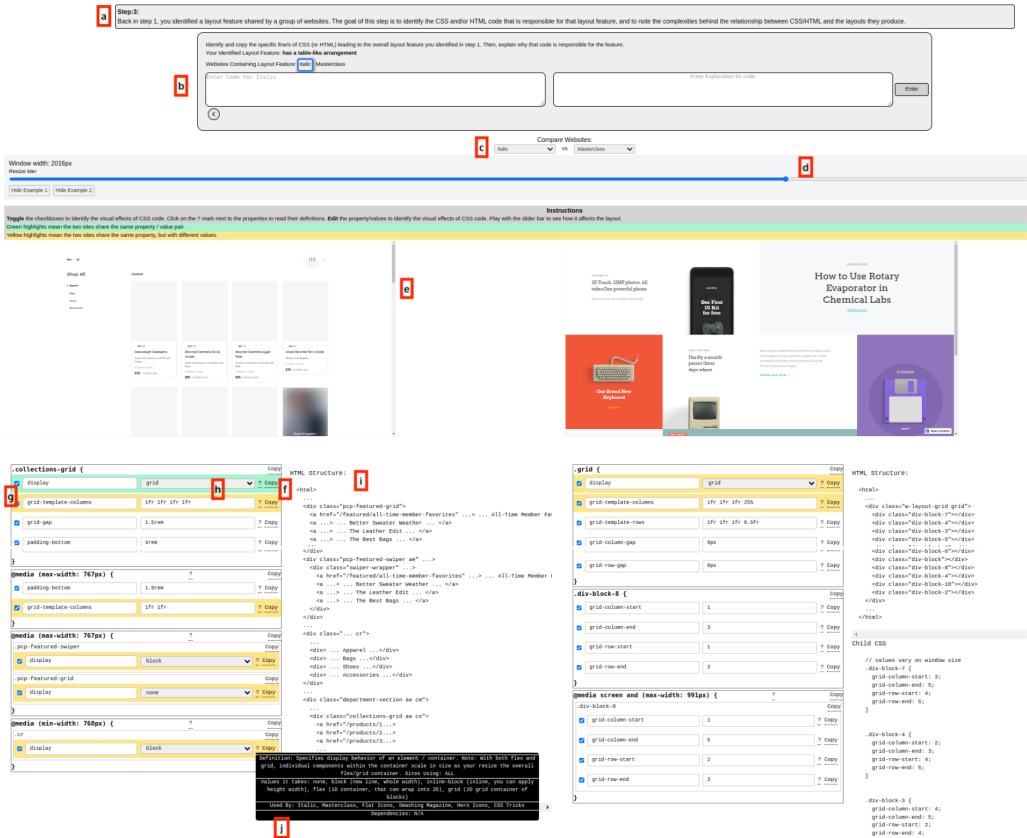


Fig. 23. KM walks users through (a) a sequence of steps (Step 3 shown here as an example), each with (b) reflection inputs shown at the top of the webpage. Users can also select (c) a pair of websites to compare. Below the inputs, KM provides users the ability to (d) interactively resize examples' widths (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.

“malleable representations” [114]—learners observe how small code adjustments impact core layout outcomes.

PM affordances that further drive learning include (1) KM (Figure 21) for organizing insights, capturing granular differences, and prompting learners for further detail in each successive cycle; (2) reflection prompts (Figure 23(b)) during comparison that remind learners to *interactively resize the page* (PM5) (Figure 23(d)) and discover easily overlooked visual and CSS differences; and (3) example-relevant CSS displays with tinkering, toggling, highlighting, and a side-by-side CSS-HTML view (Figure 23(i)). Through the CSS displays, KM *highlights salient property and property/value pairs* (PM3) (Figure 23(f)) shared by the two examples under comparison. The side-by-side HTML and hover-over syntax definitions (Figure 23(j)) intend to reduce *extraneous cognitive load* (PM4) during mapping. Together, these repeated tool use interactions aim to scaffold expert-like processes, integrating exploration and map construction into one continuous workflow. Figures 16–22 show how KM creates a seamless interplay between noticing, testing, and formalizing CSS insights.

6.3 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 KM system implementation is simpler than SE/RM. Aside from the wizarded HTML snippets, the KM system is automatically customizable: it can embed new websites, run Ply [85] to obtain¹⁸ their relevant, visually impactful CSS property-value pairs, and automatically update the page based on user tinkering, toggling, etc. Under the hood, KM structures a user's CSS inputs and leverages JS that uses the browser's *postMessage API* to update the websites' visuals. An instructor can likely set up KM on a new example in ~1–2 hours, and details on customization and workload can be found in the Appendices D–F.

6.4 Study Methods and 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, 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:

Participant 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 and reported having at least two CSS development experiences from coursework, projects, internships, or jobs. Screener questions were multiple choice. We 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, and so on. It excludes styling of text, color, and so on. By CSS technique, we mean the actual code, or in other words, the properties and values they can take on, e.g., 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, and so on. (Figure 26). 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.

¹⁸Note, we did not need to rerun Ply since that was already done in an earlier iteration for each of these sites. Our prototype from the earlier iteration 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).

Italic

<pre> collections-grid { display: grid; grid-template-columns: 1fr 1fr 1fr 1fr; grid-gap: 1.5rem; padding-bottom: 4rem; } @media (max-width: 767px) { grid-template-columns: 1fr 1fr; } @media (max-width: 767px) { .pcp-featured-swiper { display: block; } .pcp-featured-grid { display: none; } } @media (min-width: 768px) { ... </pre>	<pre> HTML Structure: <html> ... <div class="pcp-featured-grid"> ... All-Time Member Favorites ... <a ...> ... Better Sweater Weather ... <a ...> ... The Leather Edit ... <a ...> ... The Best Bags ... </div> <div class="pcp-featured-swiper ae" ...> <div class="swiper-wrapper" ...> ... All-Time Member Favorites ... <a ...> ... Better Sweater Weather ... <a ...> ... The Leather Edit ... <a ...> ... The Best Bags ... </div> </div> ... <div class="... cr"> ... </div> Apparel ...</div> </div> Rans </div> </pre>
---	---

Fig. 24. A sample of the code snippets available to all KM users.

- (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*.

Fig. 25. A sample of the glossary terms available to both KM conditions (delivered via Tutorons for the experimental condition).

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 on professional code examples. 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 tutorial¹⁹ to familiarize themselves with CDT features, such as inspecting elements and manipulating CSS properties. During CDT tool use, they filled out a spreadsheet (Figure G1 in Appendix G) 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 24) including a glossary of syntax definitions (Figure 25). This setup ensured that control users focused on learning CSS techniques rather than finding relevant code in individual examples, which is handled by existing tools [85] (not doing this may overwhelm the learner and unnecessarily limit their learning). However, the control did not experience KM's interactive PM 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.

Scoring and IRR. To evaluate if participants could understand granular CSS techniques, we scored the tests according to a rubric of outcomes, or codes, as described in Table 3, which, inspired by Ply/Isopleth evaluations, collates criteria for success [64, 85]. 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

¹⁹<https://developer.chrome.com/docs/devtools/css/>.

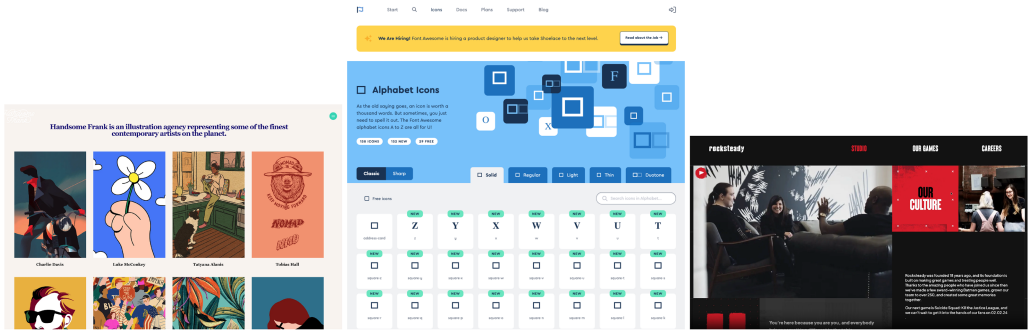


Fig. 26. Three websites rich in granular CSS techniques used in the pre-test and post-test. Handsome Frank (handsomefrank.com) is rich in grid techniques, media queries, transitions, and scrolling nuances. Alphabet Icons ([by fontawesome.com](http://fontawesome.com)) adds stretchable flex implementations 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.

Table 3. 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)

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			

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. we chose these learning goals based on our initial pilots.

(see Footnote 13), and (c) the remaining codes that required case-by-case discussion on all data (3 for KM).

Additional Interventions. Beyond relevant interventions described in the SE study, we asked participants to (a) keep track of examples they already explored, (b) resize the page, and (c) note definitions (e.g., visual layout feature/CSS technique). We proceed to describe more interventions with plausible methods of automation future work can explore.

We also encouraged all participants to strive for coverage of granular layout features and underlying CSS at the start of each new cycle (as applicable, auto-detectable in future with rules

Table 4. 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

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

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.

or zero-shot prompts since Ply [85] 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 attempt this), to ensure the code behind visual differences and similarities are distinct (as applicable, a rule-based check could attempt this), to cover all sections of the web pages in their answers if they were proceeding without doing so (a computer vision-based classifier could attempt breaking 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 attempt referencing 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).

6.5 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 4 and Figure 27, while control participant data points can be found in Table 5 and Figure 28.

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

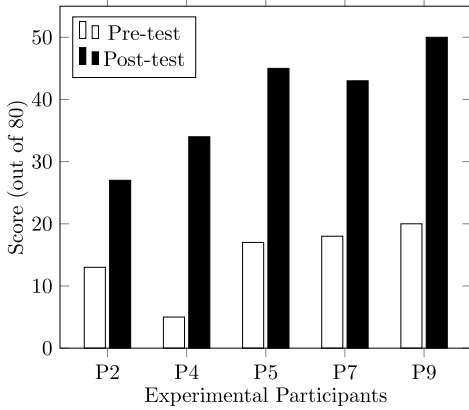


Fig. 27. Comparison of pre-test and post-test scores for experimental (KM) users.

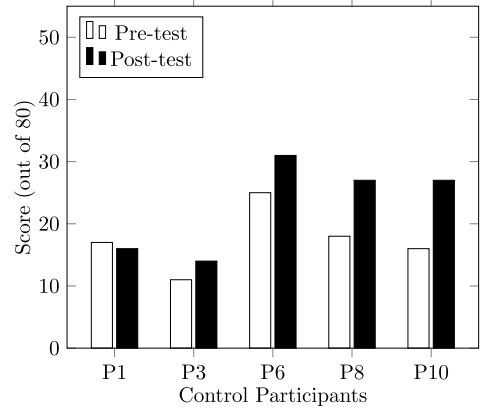


Fig. 28. Comparison of pre-test and post-test scores for control users.

Table 5. 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)

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

a significant difference between groups ($r=0.8$, $r_b = 1.0$).²⁰ As with SE, our study was exploratory, and this statistical difference was not hypothesized *a priori*; we report effect size and uncertainty as exploratory evidence for sufficient sensitivity. Robustness checks across 10,000 simulations yielded estimated power ranging from 0.991 to 1.000: Normal (0.993 [0.992, 0.995]), Gamma (0.995 [0.994, 0.997]), Weibull (0.991 [0.989, 0.993]), Log-normal (0.996 [0.994, 0.997]), and bootstrap (1.000 [1.000, 1.000]). Under all assumptions, power exceeded 0.8, and like with SE, the min effect size detectable with power > 0.8 in this 5v5 setting is $\delta = 0.9$. Future work with larger in-the-wild samples and *a priori* power analyses will strengthen generalizability.

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

²⁰ Cliff's delta was $\delta = 1.000$ (95% bootstrap CI: [1.000, 1.000]), with a two-sided permutation test yielding $p = 0.0079$. The Bayesian probability-of-superiority was: $P(X > Y) = 0.963$, with a 95% HDI of [0.891, 1.000] (equal-tailed CrI [0.868, 0.999]).

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 RQs.

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 *previously 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 first-hand. 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 4²¹ 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

²¹To ensure robustness in our tool use metrics, we developed a tool use rubric similar to Table 3. 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 [22, 84]. 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.

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 may have 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 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 [7]. 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 collectively show participants effectively reaching outcomes after tool use. 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 1fr6* 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*.

6.6 KM Discussion: Practical Learning and Technical Contributions

Unlike static CSS tutorials or web debuggers [97, 102] that leave users to probe on their own, KM orchestrates a structured learning process. While larger evaluations would strengthen our claims, we found that KM users outperformed control users engaged in open-ended exploration. They were spotting subtle layout differences between multiple examples (e.g., responsiveness, spacing, or overflow) (RQ1 result), pinpointing differentiated CSS sub-properties behind them (e.g., *flex-basis*, *grid-template-columns* nuances) (RQ1 result), and re-using them in new designs (RQ2 result). KM is designed to support such behavior with an iterative process that (a) has learners cluster visually similar sites, (b) prompts active and systematic observation, comparison, and reflection, and (c) lets learners toggle highlighted CSS rules to see immediate effects—concretely linking/mapping code to appearance. These comparative example-driven cycles embody “time-for-telling” principles [120] (i.e., nuanced contrast primes conceptual learning from code). We posit they may have helped KM users transfer opaque and intricate CSS concepts to new applications, exemplifying how process-managed tooling can ease knowledge reorganization, prevent surface-level exploration, make advanced CSS concepts more accessible, and improve frontend development education [33, 120].

7 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 RM (Figure 29), 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

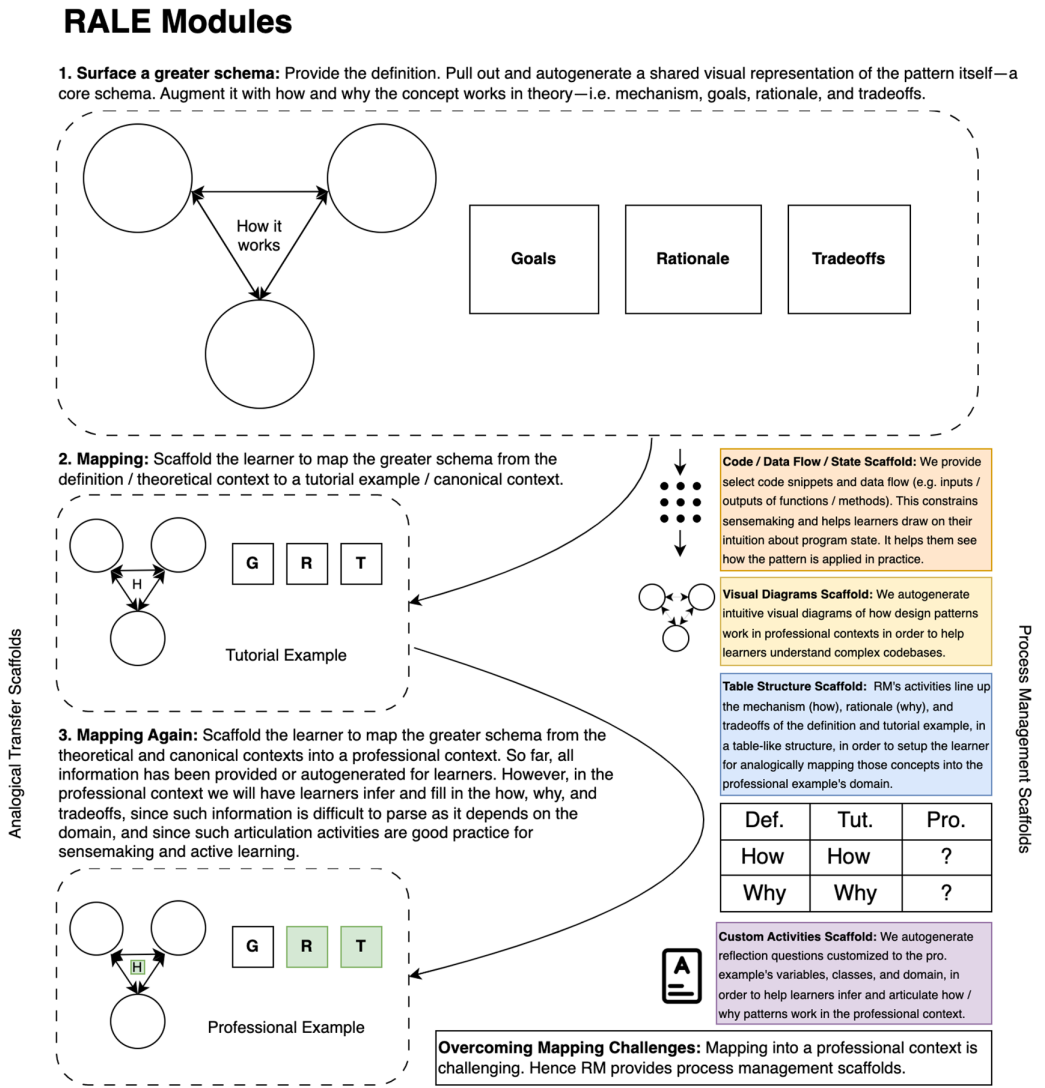


Fig. 29. Process Management for Transfer in RM. Learners progress through three stages: (1) conceptual understanding by defining the core schema with its mechanism, goals, rationale, and tradeoffs, (2) mapping (PM7) the concept to a structured tutorial example, and (3) applying it in professional contexts 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 dataflow (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.

surfacing design patterns sourced from the entire professional codebase, yet remains specific by presenting only the information necessary to understand these patterns and architectures.

In the context of design patterns, transfer may be difficult because learners: (1) may not “spontaneously” activate relevant prior knowledge [6, 114]; (2) may possess procedural knowledge of

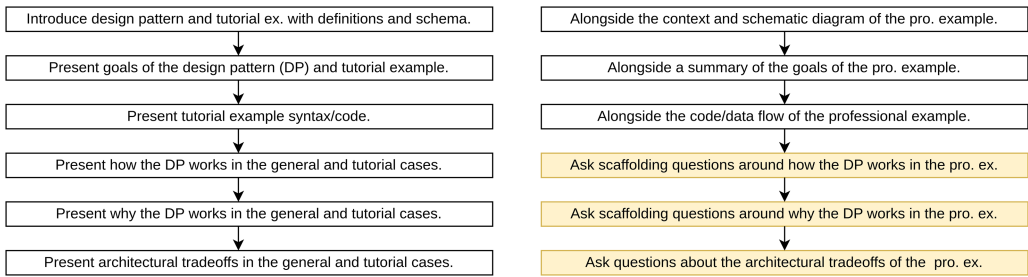


Fig. 30. Steps of RM. RM provides the learner with a table of activities for each design pattern in a professional example (as in Figures 32, 36, and 37). The table's columns correspond to the theoretical and tutorial/canonical (shown on the left), and professional (shown on the right) domains. The table's rows correspond to pattern definitions/schemas, goals, code/data, mechanisms, rationales, and tradeoffs. Most table cells are reading activities, where learners check a box after reading how each row-wise consideration manifests in the column's domain (e.g., the tutorial example's code, the theoretical goals). However, the learners are not provided the professional example's mechanisms (*how*), rationale (*why*), and architectural tradeoffs. Instead, they are provided with auto-generated scaffolding prompts that challenge them to map from the theoretical and tutorial domains into the professional context as an exercise of transfer.

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 [47, 59].

Even when learners develop a mental model of a pattern, they may struggle to extend it beyond isolated examples and fail to connect definitions, tutorials, and professional applications. In RM's process, the learner may initially have a shallow understanding of a pattern, but by the end, they can express how and why the pattern applies in practice.

System Overview. RM²² supports the sensemaking process of an intermediate developer learning about design patterns 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 and goals (Figure 32), mechanisms, rationales (Figure 36), and tradeoffs (Figure 37) across domains (Figure 30).²³ RM users proceed through three high-level steps as shown in Figure 29. The steps reflect the PM framework and the structure of the analogies being taught.

- *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 31 (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,

²²Some mixins-related content from the prototype was adopted from <https://en.wikipedia.org/wiki/Mixin>.

²³Our 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 auto-generated, the context, goals, code/data, and some Django-related footnotes were wizarded.

uses mixins to make some HTTP requests accept headers but make other requests accept both headers and etags, Figure 31 (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 tags but not others, Figure 31 (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.

Since the most effective analogies emphasize shared relationships, particularly higher-order ones that govern simpler ones [47], the table introduces rows of (i) explanations and goals (Figure 32), (ii) UML-inspired diagrams (Figure 33), (iii) code snippets (with key components like mixins and their methods highlighted) and dynamic dataflow (as applicable, Figures 34 and 35), (iv) mechanisms and rationales (Figure 36), and (v) tradeoffs (Figure 37) 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 around key pattern components and their rationales.

7.1 How RM Enacts PM: Intuitive Diagrams, Analogical Mapping, Comparing Tradeoffs, Reflection

We want learners to link theory to practice and see how and why professional examples use patterns and architectures. This includes understanding tradeoffs between alternatives and reasoning about them across real-world domains.

Problem. Learners often struggle to recognize structures, patterns, and their rationale in *complex* professional codebases. They might have limited working memory and schemas during PC [111, 117, 118], and hence may suffer from high cognitive demand when attempting to keep track of multiple coordinating components while reading code. Even if learners have some sensemaking processes and consider important example specifics, they may *struggle to transfer* design pattern theory into professional practice. They may 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 dataflow). 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 may be *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.

Furthermore, learners may struggle to integrate theoretical, tutorial, and professional contexts because they might 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

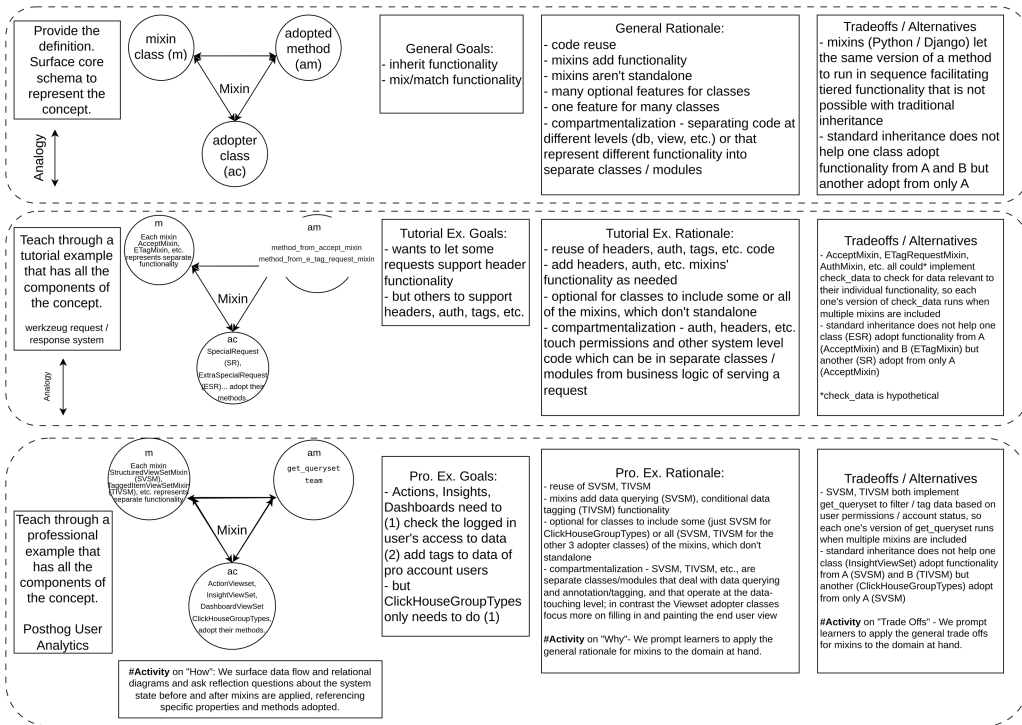


Fig. 31. Knowledge Taught by RM. This figure demonstrates the application of RM across three stages, using the pattern of mixins in Python/Django: (1) *Core Schema*: Introduces the concept of mixins (e.g., Mixin, adopter classes that inherit it, and associated adopted methods) to show how functionality 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 pattern components, evaluate tradeoffs, and understand when to adopt each mixin based on their specific goals (e.g., filtering data or conditional tagging), first on canonical examples and then on professional ones. Each stage includes scaffolds or prompts to explore this knowledge, aiming to foster deep understanding and transfer.

codebase, they may not 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 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 and validated these obstacles while designing and piloting 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.

PM Solution. RM auto-detects patterns, generates architectural diagrams, and guides learners through structured activities to connect patterns with their instantiations in professional applications (Figure 29). For each pattern, the activities pull in relevant code structures from disparate

Mixins

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

Theory	Canonical / Tutorial Example	Professional Example (posthog)
Definitions ☐ 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.)	NAME TAGS Cumulative DAUs ☐ Shows the total cumulative number of unique users that have been using your app. official tag goal essential another one tag grammar agent scope yes Repeat users over time ☐ Shows you how many users visited your app for a specific number of days (e.g. a user that visited your app twice in the time period will be shown under "2 days"). goal Daily Active Users (DAUs) ☐ Shows the number of unique users that use your app every day. official Sample - Purchase conversion funnel ☐ This is a sample of how a user funnel could look like. It represents the number of users that performed a specific action at each step. — ☐ 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.
Goals ☐ 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.
Diagrams Mixin Adopter Class method1 method1 Mixin 2 method2	Mixins of werkzeug: AcceptMixin $\rightarrow$ SpecialRequest method_from_accept_mxn method_from_accept_mxn BaseRequest $\rightarrow$ prop_of_basereq_mxn prop_of_basereq_mxn prop_of_basereq_mxn AcceptMixin $\rightarrow$ ExtraSpecialRequest method_from_accept_mxn method_from_accept_mxn	Mixins of posthog: StructuredViewSetMixin $\rightarrow$ ClickhouseGroupsView get_queryset get_queryset team team StructuredViewSetMixin $\rightarrow$ ActionViewSet get_queryset get_queryset

Fig. 32. The design pattern definition and goals.

parts of the codebase (Figure 34) and use them to build intuitive, just-in-time *visual diagrams* (PM1) (Figure 33) and *customized reflection questions* (PM8) to help the learner manage complexity based on their progress so far and hence their ZPD [116, 138]. Learners using RM should consider the domain and stakeholders of the example and answer questions that help them apply the general rationale to the specific example at hand. Throughout the process, RM guides the learner to connect the schema (definition/theory) (Figure 32), tutorial, and professional examples, breaking down silos between them. 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) [114], 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) by providing clear visual diagrams and guiding them on what to do and in what order for effective sensemaking. By linking otherwise disconnected components within the codebase, offering *relevant code snippets* (PM3), and sharing disciplinary insights on the mechanisms and rationales [114], the diagram-based activities simplify the complex learning process. The UML-inspired diagrams enhance PC [149], helping developers avoid the "Turing tarpit" [32], where everything is technically possible, but

Mixins of posthog:

StructuredViewSetMixin --
get_queryset

team

ClickhouseGroupsView
get_queryset

team

StructuredViewSetMixin --
get_queryset

team

team_id

ActionViewSet
get_queryset

team

team_id

TaggedItemViewSetMixin --
get_queryset

StructuredViewSetMixin --
get_queryset

team

InsightViewSet
get_queryset

team

TaggedItemViewSetMixin --
get_queryset

StructuredViewSetMixin --
get_queryset

team

DashboardsViewSet
get_queryset

team

Fig. 33. 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.")
```

Fig. 34. A sample of the professional example's code snippets.

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 auto-generated²⁴ 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 33) highlight how some classes in PostHog (Figure 38) 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 [32] and helping prevent learners from fixating on unimportant aspects. Starting with high-level diagrams helps learners structure their knowledge [111, 117, 118] and build a schema for how the design pattern works. Later, the system's process management seeks to scaffold analogical transfer [47] 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 prompts learners to evaluate and articulate tradeoffs, helping them understand why the chosen pattern best suits the professional example.

Tradeoffs. The tradeoffs table (Figure 37) 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 38). 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

²⁴N.B. One exception is that we wizarded a list of classes to exclude to limit the length of the activities.


```

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'

```

Fig. 35. A sample of the professional example's dataflow.

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

Fig. 36. A sample of the auto-generated activities around the professional example and its design pattern rationales.

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 esr_method) 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.

Fig. 37. The tradeoffs exercise of the professional example.

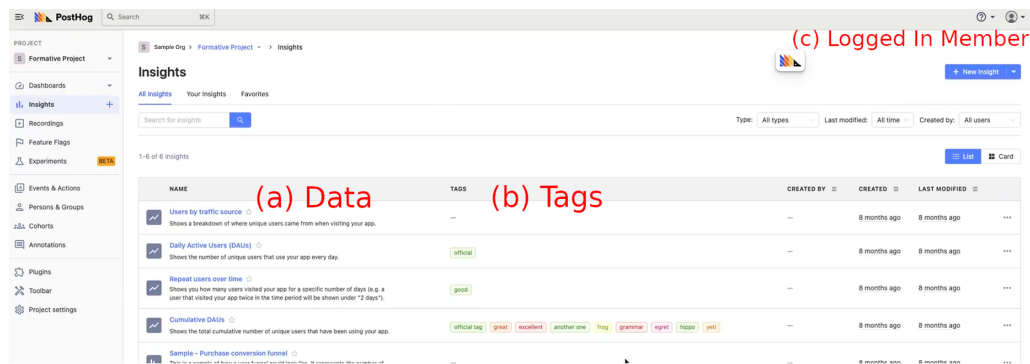


Fig. 38. PostHog is an open source product analytics dashboard that helps software development teams analyze their users' behavior. It uses mixins to determine which logged-in team members (c) can access which users' data (a). It also uses mixins to determine team members' PostHog license status, affecting their ability to tag data (b) with labels. Finally, PostHog adopts an MVC/MVT structure in Django, with views or templates for different data, controllers or viewsets that determine the data surfaced for each view, and models to keep track of different types of analytics data.

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 [120] similarly applicable but competing techniques, such as mixins and inheritance, to discover

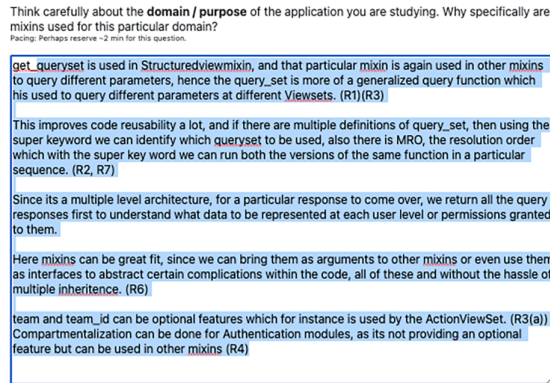


Fig. 39. Here, we can see P2 mapping rationales from the theoretical and canonical contexts (see Figure 36) into the professional context.

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, dataflow (e.g., function/method input/output as in Figure 35), activities, and tradeoffs in a side-by-side layout (as in Figures 32, 36, and 37). This unified interface is designed to equip learners to quickly grasp high-level architectures before diving into real-world implementations, and to easily map abstract rationales into concrete practice, as shown in Figure 39. The activities begin with questions about how patterns function (Figure 36) and gradually progress to questions addressing the rationale behind their use in the specific example. These guided prompts and opportunities for self-explanation [88] serve to discourage aimless exploration and encourage meaningful inquiry.

7.2 RM System Implementation

To implement RM, we begin by parsing a professional Python codebase²⁵ (i.e., Django or Flask²⁶), using 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 the “view”; and (c) Django ViewSets,²⁷ 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 Mixins²⁸ and also note which Mixin methods are adopted by other classes.

We note 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 [63], UML-based approaches [14], meta-modeling patterns as compositions of simpler components [15],

²⁵We 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.

²⁶djangoproject.com, flask.palletsprojects.com.

²⁷django-rest-framework.org/api-guide/viewsets/.

²⁸It is the convention to have the suffix `Mixin` after `Mixin` classes in Python as per python-tutorial.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).

ontology-based semantic rules [5], structural/behavioral/semantic signals [3], machine learning featurization [28], code-related metrics [43], attention-based methods [100], similarity scoring, probabilistic approaches [8], and methods that leverage LLMs to produce embeddings which are then classified [108].

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-generated²⁹ 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 33), by filling in placeholders in generic pattern templates. This being said, it should be noted that heuristic, static AST pattern detection has inherent precision/recall limits, misses dynamic behavior, and depends on language conventions; while it serves our focus on analogical mapping, future work is needed to improve robustness and reliability.

Finally, our system automatically generates activities as well. We automatically customize each pattern-specific activity to the professional example at hand. For example, in Figure 36, 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 [6, 114]. 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. An instructor can likely set up RM on a new example in ~2–3 hours, and details on customization and workload can be found in the Appendices D–F.

7.3 Study Methods and 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, interventions,³⁰ 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:

Participant 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

²⁹N.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.

³⁰Interventions 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.

written between ~1K and 5K lines of Python code and reported having at least two Python development experiences from coursework, projects, internships, or jobs. 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 control.³¹ 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 RM tool, including definitions/schemas, mechanisms, code snippets, dataflow, goals, and rationales. However, this information was not organized in a three-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 dataflow. In summary, there was no process-management-like ordered activities, reflection prompts, tradeoff tables, or auto-generated 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 PH. During the pre-test and post-test, participants could revisit the video of PH features if they wished, but not the code of the example.

Scoring and IRR. To evaluate whether participants could understand how and why design patterns apply in the professional code context, we scored their pre and post-tests according to the qualitative coding rubric described in Table 6. 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 (see Footnote 13), 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 all 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 (as applicable, rule-based checks or classifiers may help in the future), to cover each component of the mixin/MV* architecture if they were attempting to move on without doing so (as applicable, 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 (as applicable, may be enforceable through classifiers or a one-shot LLM prompt), 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 (as applicable, enforceable through classifiers), and to note while answering questions (if they were not already doing so) the

³¹In fact, both RM conditions used the tools on the experimenter's computer via Zoom remote control.

Table 6. Table Describing the Rubric Items across Two Design Patterns (Mixins, MV*) That RM and Control Users Were Tested On

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).

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. We chose these learning goals based on initial discussions with professional developers and the study of popular design patterns.

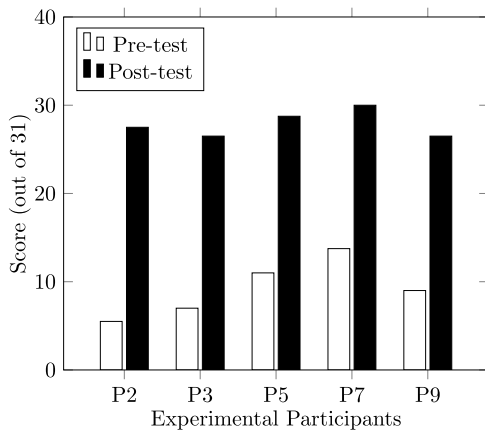


Fig. 40. Comparison of pre-test and post-test scores for experimental (RM) users.

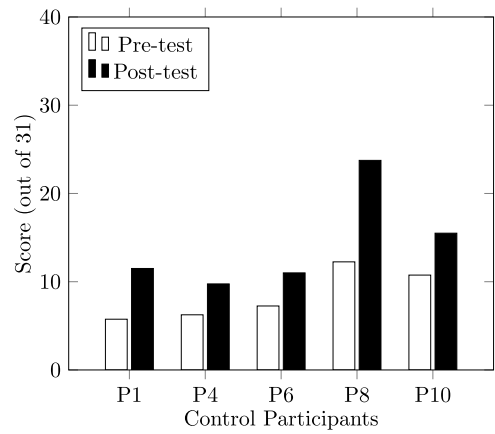


Fig. 41. Comparison of pre-test and post-test scores for control users.

definitions of adopter classes and adopted methods and their presence in the diagrams to prepare them for identifying each in the context of PostHog (as applicable, mitigable with interface fixes and better labeling of diagrams).

7.4 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 7 and Figure 40, while control participant data points can be found in Table 8 and Figure 41.

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 groups ($r = .79$, $rb = 1.0$).³² As with SE, our study was

³²Cliff's delta was $\delta = 1.000$ (95% bootstrap CI: [1.000, 1.000]), with a two-sided permutation test yielding $p = 0.0079$. The Bayesian probability-of-superiority was: $P(X > Y) = 0.963$, with a 95% HDI of [0.891, 1.000] (equal-tailed CrI [0.868, 0.999]).

exploratory, and this statistical difference was not hypothesized *a priori*; we report effect size and uncertainty as exploratory evidence for sufficient sensitivity. Robustness checks across 10,000 simulations yielded estimated power ranging from 0.996 to 1.000: Normal (1.000 [1.000, 1.000]), Gamma (1.000 [0.999, 1.000]), Weibull (0.996 [0.994, 0.997]), Log-normal (0.999 [0.999, 1.000]), and bootstrap (1.000 [1.000, 1.000]). Under all assumptions, power exceeded 0.8, and like with SE, the min effect size detectable with power > 0.8 in this 5v5 setting is $\delta = 0.9$. Future work with larger in-the-wild samples and *a priori* power analyses will strengthen generalizability (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), *dataflow*, *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 RQs on exploration and learning.

We now examine the 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 "*tradeoff[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, dataflow, reflection questions, and tradeoffs with the goal of understanding how patterns transfer to a new, professional application context [7].

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* 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 7 for participant-specific data), RM users showed marked gains, grasping pattern mechanisms, and rationales in context (reference Figure 31 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 the viewsets of 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 44), this PostHog-specific description and MV*-mixin coordination is unique to their post-test (Figure 45). Similarly, P3 made tradeoffs stating that “*using mixins gives freedom to adopter classes*



Fig. 42. Experimental P5's pre-test diagram of their understanding of PostHog's mixins.

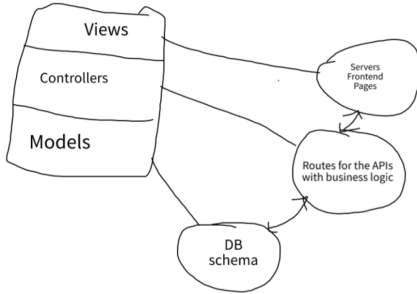


Fig. 44. Experimental P2's pre-test diagram of their understanding of PostHog's MVT architecture.



Fig. 43. Experimental P5's post-test diagram of their understanding of PostHog's mixins; some are adopted more than others.

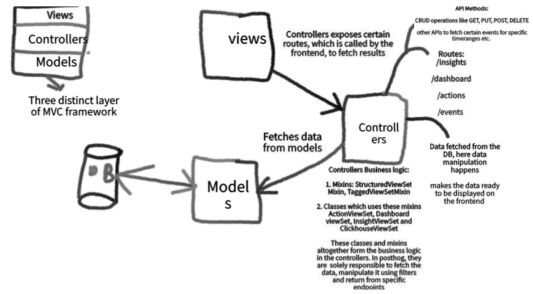


Fig. 45. Experimental P2's post-test diagram of their understanding of MVT in PostHog and MVT-mixin coordination.

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, saying “`StructuredViewSetMixin` filters objects data of logged-in PostHog [users] and queries their access based on team and organization,” while “`TaggedItemViewSetMixin` adds tags based on licensed permission.” P5 demonstrated how these mixins facilitate modularity and code reuse as they are mixed and matched as needed (see P5’s pre-test and post-test in Figures 42 and 43). 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 it 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.

Control Participants’ Learning (RQ2). Even by the post-test, the control participants’ (see Table 8 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

Table 7. Table Showing Knowledge Gains with Pre/Post-Test Scores (Out of 31) for RM Users

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 8. Table Showing Knowledge Gains with Pre/Post-Test Scores (Out of 31) for Control 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

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 46 for mixins and Figure 48 for MV*). Even by the post-test, they failed to directly state that the complexity of the PostHog codebase necessitates using MV*, which modularizes and structures code (Figure 49). 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 47). 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),³³ 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 sequentially,³⁴ 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.

³³We 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.

³⁴This sequencing is achieved via `super()` calls and the method resolution order in Python, which allows both mixins’ `get_queryset()` methods to run one after another.

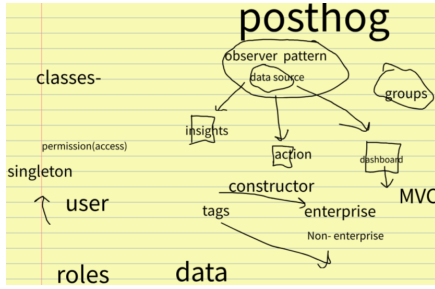


Fig. 46. Control P1's pre-test diagram of their understanding of PostHog's mixins.

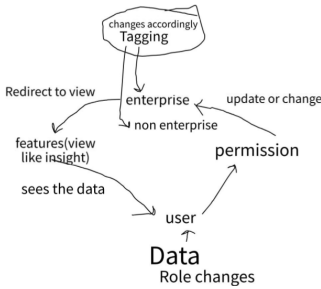


Fig. 48. Control P1's pre-test diagram of their understanding of PostHog's MVT architecture.

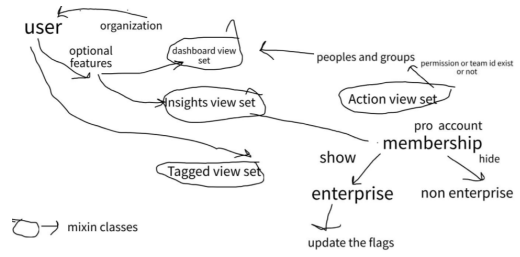


Fig. 47. Control P1's post-test diagram of PostHog's mixins. Clear evidence of relevant knowledge gains does not seem present.

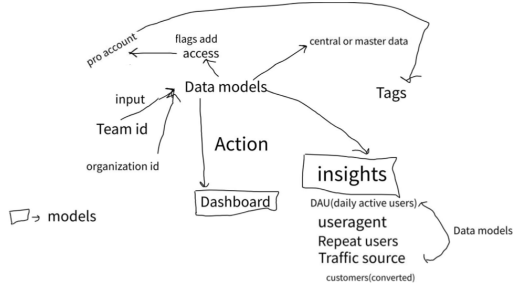


Fig. 49. Control P1's post-test diagram of MV*/MVT in PostHog. Clear evidence of relevant knowledge gains does not seem present.

7.5 RM Discussion: Practical Learning and Technical Contributions

RM automatically detects design pattern structures in production code, diagrams them, and walks learners through analogical tables, reflection prompts, and tradeoff exercises. In a controlled study, this PM workflow appeared to help RM users outperform a non-process-managed control at identifying, explaining, and transferring patterns. While larger-scale evaluations would strengthen our claims, we found that RM users could bridge theoretical lessons with real-world architectures and their domain-specific rationales. To aid them, RM goes beyond low-level or local inspection [97, 102], code foraging affordances [20, 21, 41], or textbook definitions [45, 57, 107]. It surfaces higher-level, codebase-wide architectures and tradeoffs. Via ordered, reflection-anchored tables, RM may have reduced learners' extraneous load [39], supported analogical transfer (i.e., *mapping abstract theory to practice*) [47] (RQ1 result), and fostered metacognitive [91] (e.g., *start with the high-level diagram*) and sensemaking (e.g., *reflect on how theoretical rationales apply in the professional ex.*) support for incremental mental model development [48, 61] and pattern application (RQ1 result). Through our RM study, we saw how such process-managed exploration can guide non-experts as they approach the otherwise difficult endeavor of building and retaining a domain-specific conceptual understanding of production-scale architectures (RQ2 result). Our results can inform future tools that aim to bridge theory and practice in software engineering education.

8 Discussion on the Role of PM in Professional Development and Sensemaking Real-World Code

Our controlled evaluations of SE, KM, and RM demonstrate that the PM framework can systematically enhance learners' process of exploring (RQ1) professional codebases, organizing knowledge,

and transferring conceptual understanding to real-world settings. Across varied structures that non-experts easily miss—JavaScript variable transformations, granular CSS techniques, and Python design patterns—PM scaffolds supported learners in reaching their learning outcomes (RQ2) while overcoming common obstacles, including code complexity, surface-level focus, lack of transfer, and awareness gaps. By structuring exploration [114], enabling active engagement, and promoting reflection [88, 145], PM tools provided learners with a systematic starting point and process for structured professional development.

Addressing Code Complexity through an Ordered Sensemaking Approach. Learners attempting to make sense of professional code often struggle with starting/focus points and knowledge organization. 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 [96, 133]. In reality, learners should also leverage *top-down* PC strategies where *high-level* abstractions guide the understanding of *low-level* details [111, 130, 133]. While feature location tools like FireCrystal, Unravel, and Telescope help in linking UI elements to underlying code [65, 66, 103], our work goes further by providing learners a *high-level* process for uncovering deeper architectures (e.g., *dataflow*, *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 dataflow governs specific value transformations*, *apparent finesse stems from visual nuances which stem from specific CSS intricacies*, *architectures dictate code-level implementations*) [33].

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 dataflow. 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) [6, 114] is seen in SE’s chunked exercise progression, KM’s steps/cycles, and RM’s analogical mapping tables. The PM tools seem to break through complexity by leveraging visual intuition, linking prior knowledge, and surfacing key 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 ordering³⁵ 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

³⁵ A 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.

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. Future instructional tool designers may potentially benefit from embedding these intentional processes in learning tools for real-world source code exploration.

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 events 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 PM can reinforce user exploration biases, leading to narrow, reactive sensemaking rather than goal-directed inquiry [97, 102]. While earlier tools such as Isopleth [64] and Ply [85] 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 transfer [120]. More so than control users, PM tool users *interacted* (PM5) through active comparison to advance their mental models around how components coordinate or why techniques apply in specific 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 [134], 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 may 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 comprehension pathways. This suggests that future learning tools focused on real-world code exploration could benefit from interactive comparison, which may prevent surface-level focus.

Throughout, linking prior knowledge 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, potentially preventing them from stopping at surface-level observations of disparate codebase artifacts.

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 [47, 59]. 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 [6, 45]. 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 (compared to the control) 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 and across examples. Similarly, RM users could connect multiple coordinating components and appeared to perform analogical transfer of design pattern theory into professional practice (but not across examples). In the future, learners could complete several examples on SE and RM and then map between them to gain meta insights and develop proficiency.

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 [73]. The map supports iterative externalization and organization—activities critical for transfer [6]. 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.

PM's Contribution. By shaping sensemaking, promoting interactive comparisons, and scaffolding transfer, PM strives to enhance learners' ability to construct and apply knowledge across examples of interest. Our framework (and software) creates a learning process out of raw codebases, centered around intuitive sensemaking orders, linking prior knowledge, and mapping knowledge across examples and domains. This helps learners navigate complexity with clear starting points and potentially perform transfer. It is flexible in that different tools can instantiate PM across different domains and knowledge structures, and yet potentially yield similar learning outcomes. These findings help position our contribution to the research literature on PC and scaffolding. While PM is grounded in known disciplinary best practices, unlike existing work, the PM tools and studies show the framework can help researchers and tool designers generate best-practice-infused processes to support expertise development through real-world code exploration across diverse domains. In general, our PM 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 pedagogical purposes.

- While introductory CS education and expert PC 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* [51].
- Although best practices and design heuristics 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.

In this way, process management offers computing education researchers a useful scaffolding framework for building tools that can help intermediate-level learners navigate complexity, reflect on rationale, and transfer conceptual knowledge into production-grade environments—core needs in the professionalization of computing education.

9 Limitations and Declarations

Despite the promising results, we must note some limitations.³⁶

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 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 potentially generalize the exercise generators so they can be ported to new languages and frameworks (of interest to CSEd researchers) 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.

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 ~12 hour for SE (extremes: 4:10 hour and 13 hour), ~6 to ~19 hour for KM, and ~7 to ~18 hour 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. ~1.5 hour for SE, ~8 vs. ~3 hour for KM, ~9 vs. ~4 hour 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 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, future work should explore

³⁶N.B. Use of AI Assistance: The first author used AI-based writing tools (ChatGPT, Grammarly, and Claude) for drafting and editing, ensuring all content was reviewed, fact-checked, and aligned with research objectives, with intellectual contributions and conclusions remaining entirely their own.

optimizing engagement durations while preserving learning depth, maintaining motivation, and tracking progress.

5 vs. 5 Tool Evaluation. 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 is an early-stage investigation of PM, prioritizing depth of analysis over broad generalizability. 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. Another potential limitation of these system evaluations is that talk/think-aloud interventions and back-and-forth exchanges with participants 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 [128], 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, essential part of our study design rather than a confound. Moreover, we standardized each intervention type, so systems such as LLMs can potentially automate some.

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 [82]. 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). 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/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 simple demo examples would be unlikely to affect the pre/post-test, where participants could not view code or use tools such as CDT. While training after the pre-test and immediately before tool use might have made CDT more fresh-in-mind for KM tool use, our statistics compare within each PM tool, so the training order was internally consistent within each tool’s evaluation and should not have affected our numerical results. However, qualitative cross-study insights might be stronger if we had a more uniform control tool training order across PM tools, and this remains a limitation.

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.

Technological Limitations. Across our systems, we show that professional codebases can be transformed into structured conceptual learning processes, but key challenges for self-paced, in-the-wild replication remain unsolved—including imperfect design-pattern detection, incomplete end-to-end automation for preprocessing and setup, difficulty automatically finding high-quality examples, limits in learner modeling/system intelligence and real-time intervention, and practical constraints around dynamic analysis, UI generation, and motivating learners. Finally, we interpret our results relative to widely accessible tools (e.g., CDT) as controls rather than human tutors (who are not as scalable). Though participants in both conditions could search for tutorials online, and control participants had linked references and suggested activities, future controls can incorporate more online learning tools and pedagogically designed tutorials.

Despite these limitations, we believe our work makes a valuable contribution, as described next.

10 Process Management in the Age of AI

The impact of process management can extend 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 [121] that satisfy stakeholders and maintain key relationships [81, 113, 135]. Hence, it is plausible that only true experts will remain essential amidst an AI transformation. 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* [81, 113, 135].

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 Research Opportunities—Scaling PM with LLMs. Looking forward, LLMs offer a potential avenue for scaling PM tools [105]. Future work can explore if fine-tuned LLMs with retrieval capabilities can automate some scaffolding interventions, such as prompting learners to articulate insights fully or encouraging feature exploration [104]. 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 and opening up opportunities for large-scale research “in the wild.”

In this vein, we aim to automate some wizarded interventions performed by the experimenter and work towards deploying the PM systems in the wild. While making UX fixes and hosting tools

are routine, we present LLM prompts in Appendix B (Table B1) to provide insight into how we might proceed on the AI front during tool use (and pre/post).

Table A1 and subsequent prose in Appendix A summarize the types of experimenter interventions (e.g., reread the question, explain why/how/what, use features) we layered onto sessions to ensure participants could engage fully, articulate reasoning, resolve internal inconsistencies, and leverage the tool before moving on. We wanted participants to reach their full potential, thinking through steps, processes, and reflections carefully according to best practices from the learning sciences. Some nudges to PM tool users were in a sense PM interventions themselves, and did not reveal the answer but prompted learners to use the tools to surface missing details or visit diagrams/features so we could gauge them after genuine exploration. 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 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,” visit the UI, or log their reasoning) could be automated with rules or classifiers, but higher-touch moves may need LLM systems that have the context of the codebase and rubrics. Systems 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 problem, and we do not know if LLMs can achieve this today. Still, we outline in Table B1 (and Table A1) what such systems must detect, while noting that implementing all of these checks may not be required; determining which truly merit automation, and how to balance costs to learners and implementers, remain open research.

To investigate how LLMs can support some of our wizarded interventions, we built a minimal AI agent (proof-of-concept) that has context of the codebase and learner responses. It can craft why/how/what-style follow-ups and we demonstrate its capabilities in Appendix C (Table C1). The AI follow-ups were generally appropriate and in fact more specific. However, by using better memory, retrieval, fine-tuning, and reinforcement learning environments, future research could potentially make the agent more scoped (focused on only the question at hand) and efficient (e.g., using one follow-up question to simultaneously trigger 2 or 3 interventions/behaviors). We present this as a step towards a future where AI can help us scaffold learners when they are stuck, course-correct their processes in real-time according to known best practices, and potentially even find relevant professional examples.

11 Conclusion

In this work, we proposed and evaluated a process management framework for professional development through real-world code exploration. We built and evaluated three PM systems—SE, KM, and RM—that systematically scaffold learners’ sensemaking of authentic JavaScript, CSS, and Python codebases. Our results show 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. 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, Cursor) [150]. Embedded in the context of their emotions and relations [113, 135], human experts with conceptual understanding can perform sensemaking, do compliance [121] checks, and guarantee domain-level precision [19, 29, 81]. PM operates at this pivotal intersection for computing education researchers, where software can scaffold learners to the point where they can build the

conceptual knowledge needed to evaluate and refine machine-generated outputs in real-world scenarios. This is relevant to CSEd because the active, symbiotic, and pedagogical relationship between automated tools and human developers may prevail from the first computing lesson to the most advanced stages of enterprise product development.

Acknowledgments

We thank the Delta Lab, DTR, RALE SIG, Segal Design, and Garage communities at Northwestern for their support, advice, time, and ideas. We also thank the participants in our studies for their time.

References

- [1] Gaurav Agerwala and Len Bass. 2024. Teaching software architecture design-building intuition. In *Proceedings of the 1st International Workshop on Designing Software*, 27–33.
- [2] Ekansh Agrawal, Omair Alam, Chetan Goenka, Medha Iyer, Isabela Moise, Ashish Pandian, and Bren Paul. 2024. Code compass: A study on the challenges of navigating unfamiliar codebases. arXiv:2405.06271. Retrieved from <https://arxiv.org/abs/2405.06271>
- [3] Mohammed Ghazi Al-Obeidallah, Miltos Petridis, and Stelios Kapetanakis. 2016. A survey on design pattern detection approaches. *International Journal of Software Engineering* 7, 3 (2016), 41–59.
- [4] Omar Alghamdi, Sarah Clinch, Mohammad Alhamadi, and Caroline Jay. 2023. Novice programmers strategies for online resource use and their impact on source code. In *Proceedings of 2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering*. IEEE, 92–104.
- [5] Awny Alnusair, Tian Zhao, and Gongjun Yan. 2014. Rule-based detection of design patterns in program code. *International Journal on Software Tools for Technology Transfer* 16, 3 (2014), 315–334.
- [6] Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman. 2010. How learning works. Retrieved from <https://books.google.com/books?hl=en&lr=&id=6nGaD-wAAJ&oi=fnd&pg=PR13&dq=How+learning+works++Ambrose,+Michael+W.+Bridges,+Michele+DiPietro&ots=Ki-PWYRtOX&sig=6RUFs3wHDOdxpfvLy20G21dhueE#v=onepage&q=How%20learning%20works%20%20Ambrose%2C%20Michael%20W.%20Bridges%2C%20Michele%20DiPietro&f=false>
- [7] Lorin W. Anderson and David R. Krathwohl. 2001. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives: Complete Edition*. Addison Wesley Longman, Inc.
- [8] Jameleh Asaad and Elena Avksentieva. 2024. A review of approaches to detecting software design patterns. In *Proceedings of 2024 35th Conference of Open Innovations Association (FRUCT)*. IEEE, 142–148.
- [9] Sasha A. Barab, Kurt D. Squire, and William Dueber. 2000. A co-evolutionary model for supporting the emergence of authenticity. *Educational Technology Research and Development* 48, 2 (2000), 37–62.
- [10] Pedro Luis Saraiva Barbosa, Rafael Augusto Ferreira do Carmo, João P. P. Gomes, and Windson Viana. 2024. Adaptive learning in computer science education: A scoping review. *Education and Information Technologies* 29, 8 (2024), 9139–9188.
- [11] Matthew Barr, Oana Andrei, Alistair Morrison, and Syed Waqar Nabi. 2024. 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, 81–87.
- [12] Andrew Begel and Amy J. Ko. 2019. Learning outside the classroom. Retrieved from [https://scholar.google.com/scholar?hl=en&as_sdt=0%2C50&q=Andrew+Begel+and+Amy+J.+Ko+Learning+outside+the+classroom&btnG=\(there+is+only+1+result\)](https://scholar.google.com/scholar?hl=en&as_sdt=0%2C50&q=Andrew+Begel+and+Amy+J.+Ko+Learning+outside+the+classroom&btnG=(there+is+only+1+result))
- [13] Brian R. Belland, Andrew E. Walker, Nam Ju Kim, and Mason Lefler. 2017. Synthesizing results from empirical research on computer-based scaffolding in STEM education: A meta-analysis. *Review of Educational Research* 87, 2 (2017), 309–344.
- [14] Federico Bergenti and Agostino Poggi. 2002. Improving UML designs using automatic design pattern detection. In *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*. World Scientific, 771–784. Retrieved from https://www.worldscientific.com/doi/abs/10.1142/9789812389701_0033
- [15] Mario Luca Bernardi and Giuseppe Antonio Di Lucca. 2010. Model-driven detection of design patterns. In *Proceedings of 2010 IEEE International Conference on Software Maintenance*. IEEE, 1–5.
- [16] Anica Betz, Sabrina Flake, Marcel Mierwald, and Marie Vanderbeke. 2016. *Modelling Authenticity in Teaching and Learning Contexts: A Contribution to Theory Development and Empirical Investigation of the Construct*. International Society of the Learning Sciences.

- [17] Sambasiva Bhatta and Ashok Goel. 1992. 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, 1–10.
- [18] Anastasiia Birillo. 2024. 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*, 1–2.
- [19] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent. arXiv:2403.17134. Retrieved from <https://arxiv.org/abs/2403.17134>
- [20] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 513–522.
- [21] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1589–1598.
- [22] Zana Bućinca, Maja Barbara Malaya, and Krzysztof Z. Gajos. 2021. 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* 5, CSCW1 (2021), 1–21.
- [23] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, 473–484.
- [24] Brian Burg, Amy J. Ko, and Michael D. Ernst. 2015. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 259–268.
- [25] Felix Cammaerts and Monique Snoeck. 2024. Assessing the testing skills transfer of model-based testing on testing skill acquisition. *Software and Systems Modeling* 23, 4 (2024), 953–971.
- [26] Kerry Shih-Ping Chang and Brad A. Myers. 2012. WebCrystal. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 3205–3214.
- [27] T. H. Chi Michelene. 2009. Three types of conceptual change: Belief revision, mental model transformation, and categorical shift. In *International Handbook of Research on Conceptual Change*. Routledge, 89–110. Retrieved from <https://www.taylorfrancis.com/chapters/edit/10.4324/9780203874813-9/three-types-conceptual-change-belief-revision-mental-model-transformation-categorical-shift-michelene-chi>
- [28] Abdullah Chihada, Saeed Jalili, Seyed Mohammad Hossein Hasheminejad, and Mohammad Hossein Zangooei. 2015. Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing* 26 (2015), 357–367.
- [29] Andrei Chiş, Tudor Girba, and Oscar Nierstrasz. 2014. The moldable debugger. In *Proceedings of International Conference on Software Language Engineering*. Springer, 102–121.
- [30] Codecademy. 2021. Learn to Code—For Free. Retrieved from <https://www.codecademy.com/>
- [31] Code.org. 2021. Learn Today, Build a Brighter Tomorrow. Retrieved from <https://code.org/>
- [32] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. 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*, 1–15.
- [33] Brian Danielak. 2022. How code takes shape: Studying a student’s program evolution. *Cognition and Instruction* 40, 6 (2022), 1–38.
- [34] Betsy DiSalvo. 2014. Graphical qualities of educational technology: Using drag-and-drop and text-based programs for introductory computer science. *IEEE Computer Graphics and Applications* 34, 6 (2014), 12–15.
- [35] Andrea A. DiSessa. 1993. Toward an epistemology of physics. *Cognition and Instruction* 10, 2–3 (1993), 105–225.
- [36] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95.
- [37] Brian Dorn and Mark Guzdial. 2010. 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*, 703–712.
- [38] Brian Dorn, Adam Stankiewicz, and Chris Roggi. 2013. Lost while searching: Difficulties in information seeking among end-user programmers. *Proceedings of the American Society for Information Science and Technology* 50, 1 (2013), 1–10.
- [39] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive load theory in computing education research: A review. *ACM Transactions on Computing Education* 22, 4 (2022), 1–27.
- [40] Rutwa Engineer, Volodymyr Yaremchuk, Eren Suner, Omar Khamis, Alex Apostolu, and Arthur Ng. 2024. 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*, 309–309.
- [41] Ethan Fast and Michael S. Bernstein. 2016. Meta. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 259–270.

- [42] Dror G. Feitelson. 2023. From code complexity metrics to program comprehension. *Communications of the ACM* 66, 5 (2023), 52–61.
- [43] Francesca Arcelli Fontana and Marco Zanoni. 2011. A tool for design pattern detection and software architecture reconstruction. *Information Sciences* 181, 7 (2011), 1306–1324.
- [44] Nate Furman and Jim Sibthorp. 2013. Leveraging experiential learning techniques for transfer. *New Directions for Adult and Continuing Education* 2013, 137 (2013), 17–26.
- [45] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison Wesley.
- [46] Vahid Garousi, Gorkem Giray, Eray Tuzun, Cagatay Catal, and Michael Felderer. 2019. Closing the gap between software engineering education and industrial needs. *IEEE Software* 37, 2 (2019), 68–77.
- [47] Dedre Gentner. 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive Science* 7, 2 (1983), 155–170.
- [48] Dedre Gentner and Albert L. Stevens. 2014. *Mental Models*. Psychology Press.
- [49] Mary L. Gick and Keith J. Holyoak. 1983. Schema induction and analogical transfer. *Cognitive Psychology* 15, 1 (1983), 1–38.
- [50] Google. 2025. Google Analytics. Retrieved February 24, 2025 from <http://analytics.google.com/>.
- [51] NSF Grant. 2017. Turning the Entire Web into Scaffolded Examples to Bridge Conceptual Knowledge Gaps for Novice Web Developers. Retrieved from <https://ui.adsabs.harvard.edu/abs/2017nsf....1735977Z/abstract>
- [52] Wouter Groeneveld, Joost Vennekens, and Kris Aerts. 2019. Software engineering education beyond the technical. arXiv:1910.09865. Retrieved from <https://arxiv.org/abs/1910.09865>
- [53] Philip Guo. 2021. Ten million users and ten years later: Python tutor’s design guidelines for building scalable and sustainable research software in academia. In *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology*, 1235–1251.
- [54] Philip J. Guo. 2013. Online python tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 579–584.
- [55] Chetna Gupta and Varun Gupta. 2023. C4 skills in the engineering graduate: A study to align software engineering education with market-driven software industry needs. *IEEE Transactions on Education* 67, 1 (2023), 31–43.
- [56] Mark Guzdial. 1994. Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments* 4, 1 (1994), 1–44.
- [57] Addy Hallie and Lydia Osmani. 2021. *Learning Patterns*. Retrieved from https://books.google.com/books?hl=en&lr=&id=BnJLEAAAJ&oi=fnd&pg=PT2&dq=Addy+Hallie+and+Lydia+Osmani.+2021.+Learning+Patterns.&ots=CQArQ7btEH&sig=7YPYal7krCTSch_wvj3sdFPOa4c#v=onepage&q=Addy%20Hallie%20and%20Lydia%20Osmani.%202021.%20Learning%20Patterns.&f=false
- [58] Idit Harel and Seymour Ert. (Eds.). 1991. *Pap Constructionism*. Ablex Publishing.
- [59] R. Haskell. 2001. *Transfer of Learning: Cognition Instruction, and Reasoning*. Academic, San Diego, CA.
- [60] Andrew Head, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *Proceedings of 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 3–12.
- [61] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. 2023. Synthesizing research on programmers’ mental models of programs, tasks and concepts—A systematic literature review. *Information and Software Technology* 164 (2023), 107300.
- [62] Jan Herrington and Ron Oliver. 2000. An instructional design framework for authentic learning environments. *Educational Technology Research and Development* 48, 3 (2000), 23–48.
- [63] Dirk Heuzeroth, Stefan Mandel, and Welf Lowe. 2003. Generating design pattern detectors from pattern specifications. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*. IEEE, 245–248.
- [64] Joshua Hibschan, Darren Gergle, Eleanor O’Rourke, and Haoqi Zhang. 2019. Isopleth: Supporting sensemaking of professional web applications to create readily available learning experiences. *ACM Transactions on Computer-Human Interaction* 26, 3 (2019), 1–42.
- [65] Joshua Hibschan and Haoqi Zhang. 2015. 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*, 270–279.
- [66] Joshua Hibschan and Haoqi Zhang. 2016. Telescope: Fine-tuned discovery of interactive web UI feature implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 233–245.
- [67] Lauren Himbeault and Celine Latulipe. 2024. An initial exploration of code diagram query effectiveness. In *Proceedings of the 26th Western Canadian Conference on Computing Education*, 1–7.
- [68] Cindy E. Hmelo-Silver and Merav Green Pfeffer. 2004. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. *Cognitive Science* 28, 1 (2004), 127–138.

- [69] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using annotations for sensemaking about code. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, 1–16.
- [70] Hotjar. 2025. Website heatmaps & behavior analytics tools. Retrieved February 24, 2025 from <https://hotjar.com/>
- [71] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2023. 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*, 1–12.
- [72] Julie S. Hui, Darren Gergle, and Elizabeth M. Gerber. 2018. Introassist: A tool to support writing introductory help requests. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–13.
- [73] Edwin Hutchins. 1995. How a cockpit remembers its speeds. *Cognitive Science* 19, 3 (1995), 265–288.
- [74] Stack Exchange Inc. 2021. Stack Overflow Insights—Developer Hiring, Marketing, and User Research. Retrieved from <https://insights.stackoverflow.com/survey/>
- [75] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. 2019. 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*, 27–52.
- [76] Leonora Kaldaras, Karen D. Wang, Jocelyn E. Nardo, Argenta Price, Katherine Perkins, Carl Wieman, and Shima Salehi. 2024. Employing technology-enhanced feedback and scaffolding to support the development of deep science understanding using computer simulations. *International Journal of STEM Education* 11, 1 (2024), 30.
- [77] Pankaj Kamthan. 2007. On the prospects and concerns of integrating open source software environment in software engineering education. *Journal of Information Technology Education: Research* 6, 1 (2007), 45–64.
- [78] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education* 22, 2 (2021), 1–33.
- [79] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 151–158.
- [80] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Proceedings of 2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.
- [81] Amy J. Ko, Brad A. Myers, Michael Coblenz, and Htet Htet Aung. 2025. A retrospective on how developers seek, relate, and collect information about code. *IEEE Transactions on Software Engineering* 51, 3 (2025), 848–851.
- [82] Feni Kontogianni, Lorraine Hope, Paul J. Taylor, Aldert Vrij, and Fiona Gabbert. 2020. “Tell me more about this...”: An examination of the efficacy of follow-up open questions following an initial account. *Applied Cognitive Psychology* 34, 5 (2020), 972–983.
- [83] LeetCode. 2025. System Design for Interviews and Beyond. Retrieved March 13, 2025 from <https://leetcode.com/explore/learn/card/system-design/>
- [84] Minzhi Li, Taiwei Shi, Caleb Ziemis, Min-Yen Kan, Nancy F. Chen, Zhengyuan Liu, and Diyi Yang. 2023. Coannotating: Uncertainty-guided work allocation between human and large language models for data annotation. arXiv:2310.15638. Retrieved from <https://arxiv.org/abs/2310.15638>
- [85] Sarah Lim. 2017. Ply. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 130–135.
- [86] Shu Lin, Na Meng, Dennis Kafura, and Wenxin Li. 2021. PDL: Scaffolding problem solving in programming courses. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 185–191.
- [87] Yuhan Lin, David Weintrop, and Jason McKenna. 2024. Is block-based programming “real programming”? In *Proceedings of the 18th International Conference of the Learning Sciences (ICLS ’24)*. International Society of the Learning Sciences, 2165–2166.
- [88] Marcia C. Linn and Michael J. Clancy. 1992. The case for case studies of programming problems. *Communications of the ACM* 35, 3 (1992), 121–132.
- [89] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not seeing the Forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin* 38, 3 (2006), 118–122.
- [90] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Taylor, Aniket Kittur, and Brad A. Myers. 2019. Unakite. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, 67–80.
- [91] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education* 22, 4 (2022), 1–31.
- [92] Iris Ma, Alberto Krone-Martins, and Cristina Videira Lopes. 2024. Integrating AI tutors in a programming course. In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1*, 130–136.
- [93] S. Magazine. 2013. *Understanding Advanced JavaScript*. Smashing Media GmbH. Retrieved from <https://books.google.com/books?id=r4RjQT4V5CUC>

- [94] Katerina Mangaroska, Kshitij Sharma, Michail Giannakos, Hallvard Trætteberg, and Pierre Dillenbourg. 2018. Gaze insights into debugging behavior using learner-centred analysis. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, 350–359.
- [95] Arthur Marques, Giovanni Viviani, and Gail C. Murphy. 2021. Assessing semantic frames to support program comprehension activities. In *Proceedings of 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 13–24.
- [96] Tanya J. McGill and Simone E. Volet. 1997. A conceptual framework for analyzing students' knowledge of programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297.
- [97] Megan Monroe and Casey Dugan. 2015. Disperse: Enabling web-based visualization in multi-screen and multi-user environments. In *Proceedings of IFIP Conference on Human-Computer Interaction*. Springer, 418–435.
- [98] Luke Moors, Andrew Luxton-Reilly, and Paul Denny. 2018. Transitioning from block-based to text-based programming languages. In *Proceedings of 2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. IEEE, 57–64.
- [99] Luana Müller, Milene Selbach Silveira, and Clarisse S. de Souza. 2019. Source code comprehension and appropriation by novice programmers: Understanding novice programmers' perception about source code reuse. *Journal on Interactive Systems* 10, 2 (2019), 96–109.
- [100] Rania Mzid, Ilyes Rezgui, and Tewfik Ziadi. 2024. Attention-based method for design pattern detection. In *Proceedings of European Conference on Software Architecture*. Springer, 86–101.
- [101] Charles Naumer, Karen Fisher, and Brenda Dervin. 2008. Sense-Making: A methodological perspective. In *Proceedings of SCHI Conference on Human Factors in Computing Systems (CHI '08)*, Vol. 8.
- [102] Den Odell. 2014. *Pro JavaScript Development: Coding, Capabilities, and Tooling*. Apress.
- [103] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–108.
- [104] OpenAI. 2024. OpenAI Fine-Tuning Guide. Retrieved March 2024 from <https://platform.openai.com/docs/guides/fine-tuning>
- [105] OpenAI. 2024. OpenAI Prompt Engineering Guide. Retrieved March 2024 from <https://platform.openai.com/docs/guides/prompt-engineering>
- [106] OpenAI. 2024. RAG. Retrieved from <https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts>
- [107] Addy Osmani. 2012. *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. O'Reilly Media, Inc.
- [108] Sushant Kumar Pandey, Sivajeet Chand, Jennifer Horkoff, Mirosław Staron, Mirosław Ochodek, and Darko Durisic. 2025. Design pattern recognition: A study of large language models. *Empirical Software Engineering* 30, 3 (2025), 69.
- [109] Wilson Libardo Pantoja Yopez, Julio Ariel Hurtado, Luis Mariano Bibbo, Alejandro Fernandez, and Ajay Bandi. 2023. Towards a software architecture training pattern language. In *Proceedings of the 30th Conference on Pattern Languages of Programs*, 1–33.
- [110] Yulia Pechorina, Keith Anderson, and Paul Denny. 2023. Metacodenition: Scaffolding the problem-solving process for novice programmers. In *Proceedings of the 25th Australasian Computing Education Conference*, 59–68.
- [111] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341.
- [112] Peter Pirolli and Stuart Card. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*, Vol. 5, 2–4.
- [113] Katyanna Quach. 2024. Google AI Chatbot More Empathetic Than Real Doctors in Tests. *The Register* (16 January 2024). Retrieved from https://www.theregister.com/2024/01/16/google_ai_chatbot_heathcare, archived at <https://web.archive.org/web/20241212173224/>
- [114] Chris Quintana, Brian J. Reiser, Elizabeth A. Davis, Joseph Krajcik, Eric Fretz, Ravit Golan Duncan, Eleni Kyza, Daniel Edelson, and Elliot Soloway. 2004. A scaffolding design framework for software to support science inquiry. *Journal of the Learning Sciences* 13, 3 (2004), 337–386.
- [115] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.
- [116] Margaret Riel. 1998. Education in the 21st century: Just-in-time learning or learning communities. In *Proceedings of 4th Annual Conference of the Emirates Center for Strategic Studies and Research*.
- [117] David E. Rumelhart. 2017. Schemata: The building blocks of cognition. In *Theoretical Issues in Reading Comprehension*. Routledge, 33–58. Retrieved from <https://www.taylorfrancis.com/chapters/edit/10.4324/9781315107493-4/schemata-building-blocks-cognition-david-rumelhart>
- [118] Roger Schank and Robert Abelson. 2013. *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures*. Psychology Press.
- [119] Daniela Schriebl, Andreas Müller, and Nicolas Robin. 2023. Modelling authenticity in science education. *Science & Education* 32, 4 (2023), 1021–1048.

- [120] Daniel L. Schwartz and John D. Bransford. 1998. A time for telling. *Cognition and Instruction* 16, 4 (1998), 475–5223.
- [121] Secureframe. 2025. Compliance Frameworks. Retrieved March 12, 2025 from <https://secureframe.com/hub/grc/compliance-frameworks>.
- [122] David Williamson Shaffer and Mitchel Resnick. 1999. “Thick” authenticity. *Journal of Interactive Learning Research* 10, 2 (1999), 195–216.
- [123] Anshul Shah, Anya Chernova, Elena Tomson, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. 2025. 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*, 1050–1056.
- [124] Anshul Shah, Jerry Yu, Thanh Tong, and Adalbert Gerald Soosai Raj. 2024. 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*, 1209–1215.
- [125] Zohreh Sharafi, Ian Bertram, Michael Flanagan, and Westley Weimer. 2020. Eyes on code: A study on developers’ code navigation strategies. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1692–1704.
- [126] Joshua Shi, Armaan Shah, Garrett Hedman, and Eleanor O’Rourke. 2019. Pyrus: Designing a collaborative programming game to promote problem solving behaviors. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 1–12.
- [127] Naaz Sibia. 2024. Enhancing novice programmers? Understanding through interactive visualizations and multiple representations. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, 1–2.
- [128] Robert S. Siegler and Kevin Crowley. 1991. The microgenetic method: A direct means for studying cognitive development. *The American Psychologist* 46, 6 (1991), 606.
- [129] Kyle Simpson. 2015. *You Don’t Know JS: Async & Performance*. O’Reilly Media, Inc.
- [130] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* 5 (1984), 595–609.
- [131] Juha Sorva. 2007. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2 (2007), 1–31.
- [132] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education* 13, 4 (2013), 1–64.
- [133] Margaret-Anne Storey. 2006. Theories, tools and research methods in program comprehension. *Software Quality Journal* 14, 3 (2006), 187–208.
- [134] Lucille Alice Suchman. 1987. *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press.
- [135] Maria Tomprou and Min Kyung Lee. 2022. Employment relationships in algorithmic management: A psychological contract perspective. *Computers in Human Behavior* 126 (2022), 106997.
- [136] Caryn Thiên Ngân Tran, Max Kanwal, Kristin Fasiang, and Eleanor O’Rourke. 2025. 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*, 93–108.
- [137] Ashok Kumar Veerasamy, Daryl D’Souza, and Mikko-Jussi Laakso. 2016. Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology Systems* 45, 1 (2016), 50–73.
- [138] Lev Semenovich Vygotsky and Michael Cole. 1978. *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press.
- [139] Google Web. 2021. Web Fundamentals. Retrieved from <https://developers.google.com/web/fundamentals>
- [140] Karl E. Weick. 1995. *Sensemaking in Organizations*, Vol. 3. Sage.
- [141] David Weintrop and Uri Wilensky. 2015. 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*, 199–208.
- [142] Mark Weiser. 1982. Programmers use slices when debugging. *Communications of the ACM* 25, 7 (1982), 446–452.
- [143] Mark Weiser and Joan Shertz. 1983. Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies* 19, 4 (1983), 391–398.
- [144] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2023. A think-aloud study of novice debugging. *ACM Transactions on Computing Education* 23, 2 (2023), 1–38.
- [145] Barbara Y. White and John R. Frederiksen. 1998. Inquiry, modeling, and metacognition: Making science accessible to all students. *Cognition and Instruction* 16, 1 (1998), 3–118.
- [146] Christine T. Wolf and Drew Paine. 2020. Sensemaking practices in the everyday work of AI/ML software engineering. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 86–92.

[147] David Wood, Jerome Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry* 17, 2 (1976), 89–100.

[148] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[149] Jeong Yang, Young Lee, and Kai H. Chang. 2018. Evaluations of JaguarCode: A web-based object-oriented programming environment with static and dynamic visualization. *Journal of Systems and Software* 145 (2018), 147–163.

[150] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Practices and challenges of using GitHub Copilot: An empirical study. arXiv:2303.08733. Retrieved from <https://arxiv.org/abs/2303.08733>

[151] Lanqin Zheng, Yuanyi Zhen, Jiayu Niu, and Lu Zhong. 2022. An exploratory study on fade-in versus fade-out scaffolding for novice programmers in online collaborative programming settings. *Journal of Computing in Higher Education* 34, 2 (2022), 489–516.

Appendices

A PM Tool Intervention Types and Automatability Assessment

Table A1. Catalog of Select Tool Use Intervention Types and Assessment of Their Automatability

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 <i>interface</i> 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 <i>classifier</i> or context-free <i>zero-shot prompting</i> 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 <i>context-aware LLM</i> 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 <i>context-aware LLM</i> , 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 <i>rule-based</i> : <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 <i>simple classifier</i> 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 <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 <i>zero-shot prompt</i> 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.

Here, we define each intervention type, including details around signals/heuristics that trigger it and the timing of when and if to intervene in each exercise/session. The remaining interventions are less nuanced and are presented in the prose. While interventions apply to both conditions, due to the nature of the questions and feature diversity for experimental condition participants, they likely experienced more back-and-forth.

Aside from the interventions in Table A1 and those described in the individual study sections, there were some additional, potentially automatable interventions the experimenter made, such as acquainting all participants with UI interface (replaceable using a UI tour), telling them to say more a fixed number of times (~2–3×) per session/question³⁷ upon attempts to move forward (replaceable with a rule-based system), telling them to explore more a fixed number of times (~2–3×) per session/question³⁸ before moving on, encouraging participants to write down (or occasionally draw) everything they are saying if they were not doing so (live transcription features), asking

³⁷Sometimes, 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.

³⁸Sometimes, 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.

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 attempt this), reminding participants of definitions when they need them (a classifier or context-free zero-shot prompts could attempt timing and providing 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).

B Future Work Sketch for Automating PM Interventions

With respect to Table B1 (and Table A1), one can extrapolate each 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.

Table B1. 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

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.”

We present two sample prompts out of several developed.

C AI Agent for Why/How/What-Style Follow-Ups

Table C1. Here, we Present a Minimal Viable Prototype of an AI Agent That Attempts to Craft Why/How/What/...-Style Follow-Ups to Participant Responses

Exercise	Question	Learner Response	AI Follow-Up Samples	AI Reasoning	Experimenter Follow-Up	Experimenter vs. AI Intervention
Ex. 0, which is about the \$remove object, which contains image tiles that are flagged for removal.	As you interact with the page, what's happening to the variable values shown above?	As I move through the page, the value of left and top is being changed gradually. As I locate myself on different points, I could see that the number of images being used was also varying. At some point the code wasn't used at all.	What specific changes do you observe in the \$remove variable as you interact with the page? Can you describe how those changes relate to the images being displayed? ... What happens to the images when the \$remove variable is invoked in the code? How does this affect what you see on the page?	The learner's response touches on the changes in the left and top values as they interact with the page, but it does not directly address the variable \$remove or its role in the context of image management. They mention the varying number of images but lack clarity on how \$remove specifically relates to these changes.	Which [specific] variables were you referring to?	The AI is actually more specific, which is fine in this case.
Ex. 1, which is about how the \$remove variable is set by images on the page.	What is happening in the code? How does it shape the visual output?	I think a LOT OF THING IS GOING ON IN THIS ENTIRE PAGE. So, having only those elements that is important to make the whole page pleasing is important. So the statement given above is basically used to include only the necessary components.	What specific elements are being removed by the \$remove variable, and how does that affect what is displayed on the page? ... Can you explain how the \$map variable relates to the \$remove variable in terms of the elements it contains? ... Suggested hint: Consider dragging the comic to observe how the elements in the \$remove variable change in real-time. This will help you connect the code to the visual output more clearly.	The learner's response indicates an understanding of the importance of including only necessary elements for the visual output. However, it lacks specific details about how the code snippet directly influences the variable values and the visual representation on the page. A more precise explanation of what the code does and its immediate effects on the DOM would strengthen their response.	A reminder about annotation on the elements on the left, and to interact with the page (<i>in the hope they'd connect the elements on the page with the remove variable</i>).	The AI is only targeting the why/how/what/... style questions, but in the future, prompt modifications and fine-tuning (SFT) could enable the interleaving/combining of feature use reminders with the follow-ups.
Ex. 5, which is about how we set \$map's top and left CSS properties to position	What is happening in the code? How does it shape the visual output?	I guess all the pieces of images are placed at a particular position that is being defined in the CSS code. The position left and top are used to keep in place the images to get the desired output.	What specific values do you think the position variable holds, and how do they relate to the placement of the images in the \$map? ... Can you explain how changing the position values would impact the visual layout of the images on the page?	The learner correctly identifies that the left and top CSS properties are used to position the images within the \$map. However, they could elaborate more on how the specific values of the position variable affect the placement of the images and the overall visual output.	The experimenter missed reminding the learner on this one.	The AI provided a more specific follow-up than the human experimenter here.

(Continued)

Table C1. Continued

Exercise	Question	Learner Response	AI Follow-Up Samples	AI Reasoning	Experimenter Follow-Up	Experimenter vs. AI Intervention
Ex. 7, which surfaces how position affects centre which affects name which affects tile which affects the images loaded onto the map.	As you interact with the page, what's happening to the variable values shown above?	The new terms in this exercise are center_last which depends on the value of position [0] and [1]. the value is tile size is fixed and I think it is the size of a single image. the value of position is changing as I scroll through the page. The values in name changes as I move between the tiles.	Can you explain how the values of "centre" and "centre_last" are derived from "position"? What do these values represent in relation to the tiles? ... How does the change in "centre" influence the loading of new images in "\$image"? Can you describe the connection between "name" and the images being displayed?	The learner's response identifies that the value of "position" changes as they interact with the page and recognizes that "tilesize" is fixed, which is a good start. However, they could elaborate more on how "centre" and "centre_last" are calculated from "position," and how these changes affect the loading of new images represented by "name" and "\$image." Overall, the answer lacks detail on the specific relationships and transformations occurring between the variables.	A reminder to cover all the variables, by asking them to look at the set of variables.	The AI provides more specific (and seemingly more helpful) advice than the human intervener. However, the advice arguably goes slightly beyond the scope of the question at hand, which is primarily about the variable values. In the future, prompt modification, SFT, or RL can be leveraged to tune the AI to restrict itself to the question at hand. Still, the follow-up seems reasonable and valuable. ... In another participant session, on this exercise but for the code question, when a participant was investigating variables like centre and tile_name, we did prompt them to cover their relationship in a manner similar to the AI's second follow-up here. This suggests reasonable alignment between the intent of our follow-ups and the AI agent's follow-ups.

The AI agent is built using the LangChain framework (Langchain.com). It has access to (a) descriptions of types of interventions, we perform, with emphasis on the why/how/what...style ones, (b) the SE scoring rubric (which is assumed to have been reviewed by a human expert for each example before being deployed in the wild), (c) an AI-generated natural language description of the UI, (d) the underlying source code of the example, and (e) the abstract of this paper itself. (f) It contains prompts that encourage it to use the background context but focus on the given exercise, question, learner response, and goal at hand in each inference call. (g) It also contains guiding prompts to focus on variables relevant to the exercise, probe for missing reasoning around key learning outcomes, and not give away the answer. Further prompt engineering, supervised fine-tuning (SFT), reinforcement learning (RL), and retrieval (RAG) can potentially improve the accuracy of the agent. Please note that this is a proof-of-concept to automate certain interventions. We do not claim general reliability of such LLM agents or that the human experimenter can be replaced by LLM agents. Future research with more rigorous analysis around accuracy/correctness, precision/recall, timings, robustness across example sets, and volume of testing is needed.

D Adapting SE to a New Codebase: Customization and Workload

What Must Be Configured. To deploy SE on a new JavaScript codebase, an instructor (or TA) proficient in JS typically:

- prepares a runnable copy of the site by downloading the site's frontend and adjusting file paths/endpoints in the HTML source and an ExpressJS one-liner to ensure the example (along with its libraries) still works locally.

- copies in a couple embedded one-liner scripts (provided in the SE repo README) that enable tracing and data capture.
- runs linters (pre-specified in the SE repo) to auto-fix the code, and scripts that update the code so it's traced and its state is tracked.
- interacts with the site (instrumented by the tracer and state tracker) to generate (a) a downloadable run log of function calls, (b) a classification of code components as modifiers, users, etc., and (c) an exercise order based on the run log and code/DOM object classification.
- reviews the auto-generated exercise sequence, (a) optionally pruning near-duplicates edge cases and ones with skewed variable distributions (e.g., one exercise with 10 variables and another with 1), (b) specifying variables or annotations to skip or unfurl, and (c) specifying particular DOM elements of interest, before exporting the bundle for use by the exercise UI generator.

Workload and Expertise. After an initial one-time environment setup (~0.5–1 day the very first time), porting SE to a new site typically requires ~2–4 hours: 30–90 minutes for path/config updates and instrumentation, 15–30 minutes to record exemplar interactions, and 30–60 minutes to review the generated sequence and make tweaks. Required expertise is intermediate-level web development (familiarity with dependencies/paths, DOM/JS).

Learning Curve. Instructors can likely learn the workflow described above in ~60–90 minutes before they proceed to implement it on new sites; students may need ~10–15 minutes of orientation (e.g., what the ordered exercises show, how to read live values, where to write reflections). All time estimates above are preliminary, based on experimenter experience; future deployments will validate them.

E Adapting KM to New Websites: Customization and Workload

What Must Be Configured. To add new examples to KM, an instructor proficient in CSS typically:

- supplies the list of new target sites (3–6 recommended) with URLs (usually downloaded and self-hosted) and site titles;
- imports or curates the list of visually impactful CSS properties/values per site (via Ply [85] visual regression testing output for seeding properties to toggle/tinker);
- verifies iframe embeddings and the postMessage API are working (i.e., tinkering with and toggling properties works on embedded sites);
- locates and includes relevant HTML snippets into the display (this is not automated in our current implementation); and
- selects and adds new CSS concepts by updating the glossary and hover over syntax definitions as needed

Workload and Expertise. With an existing property list, adding a site may take ~1–2 hour (e.g., URL, embedding, HTML, and glossary updates). Building a fresh property list for a site usually takes ~2–4 hours (e.g., configuring and running Ply). The expertise required is intermediate-level CSS (media queries, grid/flex, spacing, etc., for instance, to gut-check Ply's outputs look reasonable) and general web deployment.

Learning Curve. Instructors may spend ~20–30 minutes to test the grouping/compare/map workflow after the ~3–6 hour setup; students may spend ~10–15 minutes in onboarding (e.g., how to group, compare, toggle/tinker, and reflect). These time estimates are preliminary and require validation.

F Adapting RM to a New Codebase: Customization and Workload

What Must Be Configured. To port RM, an instructor typically:

- clones the target Python/Django/Flask repository and ensures it builds (local or containerized);
- selects patterns of interest and disables others' detectors (we have rule modules for Mixins and MV*; by default both are on, but each can be commented out; to add more selected patterns, the instructor would need to edit the RM codebase);
- specifies a path to the code, and classes to exclude to make the activities tractable;
- runs an AST-based parser to extract roles/relations into a data dump
- runs a Python server which loads pattern metadata from the data dump, and presents auto-generated diagrams and activities.

Workload and Expertise. For a mid-sized codebase, a first pass to detect patterns and produce diagrams/activities typically takes ~2–3 hours (open source repo setup 1–2 h if needed; AST run/review 0.5 h; activity generation/review 0.5 h). The expertise needed is intermediate-level Python plus basic framework familiarity (e.g., where models/views live, for instance, to gut-check AST parser outputs look reasonable). The instructor may need to prune redundant activities/diagram outputs on occasion.

Learning Curve. Instructors may need ~60–90 minutes to learn the detect–review–regenerate loop before proceeding to the steps above; and, students may require ~2–5 minutes to familiarize themselves with the interface of diagrams and activities before getting started. These time estimates are preliminary and require validation.

G KM Control

Figure G1 shows the control participants' experience for the KM study.

1) Exploration		
Instructions:	We are trying to view examples and list their visual features in this step. Step A. View "Italic". Step B. Resize the page. Play around with the page width. Step C. Recall what a visual layout feature is, as explained earlier. List as many visual layout features of "Italic" as you can in the left-most column of the "Features" table below. Put "Italic" under the Example column. Step D. Think about which CSS techniques might be behind the visual layout features you noted in the previous steps. Feel free to leverage CDT. D.part1. Fill in the CSS Code responsible for the layout feature. You can consult the CSS / HTML code snippets package for each site. D.part2. Read property definitions in the package as well. D.part3. For each site, identify and copy in the specific lines of CSS leading to each visual layout. D.part4. Bold the part of the syntax that is more relevant to the visual layout, if helpful, i.e. if you are talking about a width being 132 px then grid-template-columns: repeat(auto-fill, minmax(132px, 1fr)) Repeat steps A-D for "Masterclass". Repeat steps A-D for "Flat Icons", then "Smashing Magazine", then "Hero Icons", then "CSS Tricks". Look through your features and sites, and make sure you have comprehensively listed out all visual features, including very specific, granular ones. Make sure you have entered granular, specific CSS code responsible for all visual features.	
Features	Example	Fill in the CSS Code Responsible:
[example] has a grid-like layout	[example] Site A	[example] display:grid;
[example] has a grid-like layout	[example] Site B	[example] display:flex;
[example] characteristic A	[example] Site C	[example] CSS code for characteristic A
[example] characteristic B	[example] Site C	[example] CSS code for characteristic B
[example] characteristic C	[example] Site C	[example] CSS code for characteristic C

Fig. G1. The control participants have access to the KM experience (Figures 16–22) without its primary process management affordances. The control does not have (a) a grouping activity, step 1 of KM PM, (b) a comparing activity to find differences, KM's step 2, (c) structured prompting for code coverage with rationale, KM's Steps 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. Note that the steps only provide them information about the affordances and resources available to them. For instance "step D" says "feel free to leverage CDT" and we do not impose a process of how to do so upon them.

H Complete Scoring Rubrics

Tables H1–H11 provide the complete scoring rubrics used for:

- Evaluating KM participants’ understanding of granular CSS techniques across the three example websites. Each item represents a specific CSS technique or sub-technique that participants could identify, explain, or apply.
- Evaluating RM participants’ understanding of design patterns with specific partial credit criteria.

Examples and justifications are included as well. Since the SE rubric is in the SE section, we only include examples and justifications here.

Table H1. Handsome Frank Rubric Items (1–24) for KM Evaluation

ID	Section	Item
1	Header	1pt for transition
2	Header	1pt for transitions affecting multiple elements, or for animations of elements
3	Grid of Illustrators	1pt for grid or flex
4	Grid of Illustrators	1pt for grid-template-columns or flex-wrap/flex-direction:columns
5	Grid of Illustrators	1pt for grid-template-columns with 4 fr’s or flex-wrap with width 25%
6	Grid of Illustrators	1pt for media query
7	Grid of Illustrators	1pt for scrollable dependent on the media query
8	Grid of Illustrators	1pt for overflow: scroll
9	Grid of Illustrators	1pt for flex at lower widths
10	Watch, Animation, Who’s Frank	1pt for grid or flex as responsible for changing icon sizes
11	Watch, Animation, Who’s Frank	1pt for grid-column or span
12	Watch, Animation, Who’s Frank	1pt for media query
13	Watch, Animation, Who’s Frank	1pt for number of columns dependent on the media query
14	Watch, Animation, Who’s Frank	1pt for display:block (or alternatively, some grid/flex sub-property that ensures 1 column)
15	Watch, Animation, Who’s Frank	1pt for flex (or alternatively grid)
16	Watch, Animation, Who’s Frank	1pt for mentioning adjustments through min/max height, overflow, or width, etc.
17	Illustrators/Read	1pt for grid-column-gap
18	Illustrators/Read	1pt for margin
19	Illustrators/Read	1pt for padding
20	Illustrators/Read	1pt for media query changing amount of gap
21	Read	1pt for transition
22	Read	1pt for overflow hidden
23	Read	1pt for display:grid or display:flex
24	Read	1pt for overflow:hidden; or overflow:scroll;

Table H2. Handsome Frank Rubric Examples for KM Evaluation

Rubric Item ID	Section	Rubric Item	Example Snippet	Score	Justification	Participant ID
1	Header	1pt for transition	"There are no specific transitions and transformations for the website layout."	0	"They expressly deny the presence of any transition."	P1 Post
1	Header	1pt for transition	"on hovering over the menu icon in the top-right corner there is a smooth transition followed by an element resize ... transition: 1 s"	1	Their quote talks about transitions in the header.	P7 Post
4	Grid of Illustrators	1pt for grid-template-columns or flex-wrap/flex-direction:columns	"Some elements on desktop view are displayed in a grid list mode ... grid-template-columns: 1fr 1fr 1fr 1fr;"	1	Stated the specific syntax required.	P2 Post
13	Watch, Animation, Who's Frank	number of columns dependent on the media query	"[Description] Watch section—Two items are placed in a row with no gap ... [on] Mobile—Elements become a single column with two elements [Code] Watch section ... grid-template-columns: 1fr 1fr; ... Mobile @media ... { grid-template-columns: 1fr; }"	1	They use the media query to manipulate the num columns.	P9 Post

Table H3. Alphabet Icons Rubric Items (25–74) for KM Evaluation

ID	Section	Item
25	Header	1pt for media queries
26	Header	1pt for media queries determining the buttons width
27	Header	1pt for media queries determining the tab labels
28	Header	1pt for media queries determining the modern graphic appearance (top right)
29	Header	1pt for media queries determining gutter/outer widths
30	Header	1pt for media queries determining appearance through display:none
31	Header	1pt for media queries determining appearance through display:flex (or :block or :grid)
32	Header	1pt for overflow
33	Header	1pt for overflow-y
34	Header	1pt for overflow-y:clip
35	Main Icons Grid	1pt for display:grid (or display:flex)
36	Main Icons Grid	1pt for grid-template-columns (or flex-wrap)
37	Main Icons Grid	1pt for repeat auto-fill or auto for grid-template-columns (or flex-wrap)
38	Main Icons Grid	1pt for minmax using some sizing that determines the number of columns (or flex width %)
39	Main Icons Grid	1pt for 1 fr in grid-template-columns
40	Main Icons Grid	1pt for media queries governing the sizes of the icons
41	Main Icons Grid	1pt for grid-template-columns (or flex or flex-wrap) affects icon sizes
42	Main Icons Grid	1pt for grid-template-columns minmax shaping width of the columns (or width %)
43	Main Icons Grid	1pt for the grid-auto-rows or grid-template-rows shapes the height
44	Main Icons Grid	1pt for display:grid (or display:flex) with a min of 4 columns
45	Main Icons Grid	1pt for grid-template-columns (or flex-wrap) with a min of 4 columns
46	Main Icons Grid	1pt for repeat auto-fill (or flex-wrap) with a min of 4 columns
47	Main Icons Grid	1pt for minmax using some sizing that determines the number of columns (or flex width %) with mention of a min of 4 columns
48	Main Icons Grid	1pt for 1 fr in grid-template-columns with mention of a min of 4 columns
49	Main Icons Grid	1pt for gap-x
50	Main Icons Grid	1pt for grid-auto-rows
51	Main Icons Grid	1pt for grid (or flex) default to scale the spacing
52	Main Icons Grid	1pt for grid-gap (or padding or margin if using flex)
53	Icons in Use	1pt for media queries shaping the max-width % (or grid-col-start/end OR flex grow)
54	Icons in Use	1pt for media queries shaping the flex-basis
55	Icons in Use	1pt for max-width 33% for top one
56	Icons in Use	1pt for max-width 50% for top one, second row
57	Icons in Use	1pt for max-width 33% for bottom one
58	Icons in Use	1pt for media queries shaping max-width or flex-basis
59	Not finding what you need	1pt for media queries shaping max-width or flex-basis
60	Icons in Use	1pt for padding (or margin or grid-gap)
61	Icons in Use	1pt for padding-left/padding-right
62	Other Categories	1 display-flex (or display:grid)
63	Other Categories	1pt for flex-basis or max-width: 16.67% (or 1frX6 in the case of grid) for higher width
64	Other Categories	1pt for flex-basis or max-width: 33% (or 1frX3 in the case of grid) for lower width
65	Other Categories	1pt for media query determining which width at which to switch
66	Other Categories	1pt for display-flex (or display:grid), in the context of how icons get smaller at lower page widths
67	Other Categories	1pt for flex-basis or max-width: 16.67% (or 1frX6 in the case of grid) for higher width, in the context of how icons get smaller at lower page widths
68	Other Categories	1pt for flex-basis or max-width: 33% (or 1frX3 in the case of grid) for lower width, in the context of how icons get smaller at lower page widths
69	Other Categories	1pt for media query determining which width at which to switch, in the context of how icons get smaller at lower page widths
70	Other Categories	1pt for display-flex (or display:grid), in the context of how there are a minimum of 3 columns per row
71	Other Categories	1pt for flex-basis or max-width: 16.67% (or 1frX6 in the case of grid) for higher width, in the context of how there are a minimum of 3 columns per row
72	Other Categories	1pt for flex-basis or max-width: 33% (or 1frX3 in the case of grid) for lower width, in the context of how there are a minimum of 3 columns per row
73	Other Categories	1pt for media query determining which width at which to switch, in the context of how there are a minimum of 3 columns per row
74	Other Categories	1pt for grid-template-columns or flex:wrap (which would also work—could just say flex for this), in the context of spacing or consistent gaps between icons

Table H4. Alphabet Icons Rubric Examples for KM Evaluation

Rubric Item ID	Section	Rubric Item	Example Snippet	Score	Justification	Participant ID
28	Header	media queries determining the modern graphic appearance (top right)	“[DESCRIPTION] on small sized devices the top right section disappears [CODE] @media screen(){... top-right-section {display: none;}}”	1	They show that we display:none inside a media query for the top-right-section.	P7 Post
38	Main Icons Grid	minmax using some sizing that determines the number of columns (or flex width %)	“[DESCRIPTION] The arrangement of the free icons appears to be in a grid-like layout structure [CODE] grid-template-columns:repeat(auto-fill, minmax(100px, 1 fr));”	1	They identify the minmax code that determines the size and hence number of columns.	P8 Post
64	Other Categories	flex-basis or max-width: 33% (or 1frX3 in the case of grid) for lower width	“[DESCRIPTION] uses table layout to properly align the items. It has constant spacing. On smaller devices table changes to two row layout [CODE] grid-template-columns for larger devices are also set for 6 columns but for smaller devices it is set to 3.”	0	They get the general visual feature but not the max-width % concept we are testing for	P5 Post

Table H5. Rocksteady Rubric Items (75–80) for KM Evaluation

ID	Section	Item
75	Whole Site	1pt for grid-row-start/end OR flex grow
76	Whole Site	1pt for grid-col-start/end OR flex grow
77	Whole Site	1pt for media queries shaping some of the starts/ends OR flex wrap
78	Whole Site	1pt for media queries shaping some of the spans OR flex wrap OR grid
79	Whole Site	1pt for grid-template-columns OR display:flex
80	Whole Site	1pt for repeat(X, ... 1fr ...)

Table H6. Rocksteady Rubric Examples for KM Evaluation

Rubric Item ID	Section	Rubric Item	Example Snippet	Score	Justification	Participant ID
77	Whole Site	media queries shaping some of the starts/ends OR flex wrap	“[DESCRIPTION] the page is responsive through the use of different display styles and custom grid sizes adapting to different screen sizes. [CODE] element{grid-column-start: 1; grid-column-end: 3; grid-row-end: 5; grid-row-start: 3;} ... @media screen{display: grid; ... grid-column-start: 1; grid-column-end: 5; grid-row-start: 1; grid-row-end: 1;}”	1	They adjusted the grid starts/ends based on the application of the media query.	P7 Post

Table H7. RM Scoring Rubric for Mixins (1–18)

ID	Pattern	Partial Credit
1	Mixins allow CHGV to only adopt SVSM's get_qs while I, A, D adopt both SVSM and TIVSM's get_qs.	0.5 for getting the general idea (that mixins can be mixed and matched, being included alone sometimes, or with other mixins other times) but not the functionality of the mixins and specific methods correctly
2	Multiple versions of a method/prop allow different versions (SVSM's and TIVSM's) of get_qs to filter/tag data one after another.	
3	Stating that SVSM/TIVSM mixins helped PostHog reuse code, adding functionality without enforcing an "is a kind of" parental relationship.	0.5 for just mentioning code reuse because of mixins; 0.75 for mentioning everything except the names of SVSM/TIVSM; 0.5 for everything except "is a kind of" not enforced
4	Stating that SVSM's get_qs is filtering down data to that which the user has access to.	0.5 for just the access restriction
5	Stating that TIVSM's get_qs is retrieving tag data but only for enterprise users. (a) The mixin checks whether the user is licensed in order to determine whether they should be able to view tags. (b) TIVSM's get_qs checks if the user can see tags based on their license, and the class that adopts it applies that tag visibility functionality to specific objects like insights, actions, dashboards.	0.5 for saying mixins are responsible for permissions/access but not TIVSM specifically; 0.5 or identifying a tag-related mixin but not the rest of the details
6	Stating that I, A, D, CHGV's get_qs are trying to visually show data to the user that the user has access to.	0.5 for mentioning some of IADG and access
7	Stating that I, A, D's get_qs are trying to visually tag data for enterprise users.	0.5 for some tagging in some function or class or mixin to visually tag data for users; 1 for mentions of I, A, D, AND G and get_qs and visually tagging data for enterprise users
8	Code reuse of SVSM, TIVSM complex filtering/tagging logic (across I, A, D, CHGV) in PostHog particularly helps with collaboration on a complex codebase.	0.75 if they don't mention TIVSM/SVSM but state everything else; 0.5 if they mention code reuse and some aspect of filtering or permissions or tagging; 0.25 if they only mention reuse after identifying one of the 2 important mixins; 0.5 for everything except the collaboration point
9	Optional tagging features for I, A, D and shared filtering feature among I, A, D, CHGV.	0.5 for this but not describing filtering per se
10	Noting how we compartmentalize logic around filtering/retrieving tags in the SVSM/TIVSM from view preparation logic of I, A, D, CHGV. (a) This is so developers can touch compartmentalized sections of code depending on whether they want to edit the Mixin's tagging/filtering functionality (db-touching) or the more domain-focused classes that adopt that functionality (UI-touching).	0.25 for just mentioning mixins afford some level of generalizability; 0.5 for just mentioning aspects like loose coupling but not specifically about mixin vs. adopter code; 0.75 for everything except the domain-focused vs. db-focused distinction/separation
11	I, A, D, CHGV, or SVSM/TIVSM could be exchanged out and hence respect DI—good for decoupling.	
12	get_qs—both mixins' versions of these run, letting filtering and tagging happen one after another, layering the mixins' functionalities.	
13	the team_id/team property is used to determine the user's team and hence what data they have access to; the organization property might also be used to determine access.	
14	team_id/team of the logged in user is important information that SVSM needs to keep in mind, and that team logic can be shared across I, A, D, CHGV; it is okay if they say separately that team is used in get_qs for filtering and separately that get_qs is reused.	
15	Analytics needs to be used by many team and organization members so this logic around getting the team identity, or filtering by team, is shared across I, A, D, CHGV.	0.25 if they mention team, organization, etc. but do not clearly make this specific point; 0.5 if they talk about team/org and also general code sharing but not this specific point
16	Mixins, esp. with use of super, allow us to prevent multiple inheritance issues, and know which version of a given property/method (get_qs) to run; in fact in the case of PostHog allowing TIVSM and SVSM's versions to run 1 after another, both filtering and tagging; traditional multiple inheritance that retained the child-is-a-kind-of-parent structure wouldn't allow for this level of customization.	0.5 for discussing multiple inheritance and the is-a concept; 0.5 for discussing the sequential tagging and filtering; 0.5 for discussing other patterns where everything must be inherited
17	We do not create instances of SVSM and TIVSM; these classes don't stand alone. Things like permissions checks and tagging are slices of functionality that may be mixed and matched in order to create an efficient implementation for the complex codebase.	0.5 for combining multiple sets of functionality, mixing and matching, and/or mixins serving a different purpose than the classes that adopt them, but not necessarily clearly (a) saying they don't stand alone/(b) namedropping SVSM/TIVSM
18	SVSM/TIVSM's get_qs access the model data from the children I, A, D ViewSets due to the children's super calls.	0.5 for mentioning or implying that they access the model data but not the super part; 0.5 for mentioning super but not the get_qs part

A, ActionViewSet; CHGV, ClickHouseGroupsView; D, DashboardViewSet; Get_qs, Get_Queryset; I, InsightsViewSet; SVSM, StructuredViewSetMixin; TIVSM, TaggedItemViewSetMixin.

Table H8. Mixins Rubric Examples for RM Evaluation

Rubric Item ID	Rubric Item	Example Snippet	Score	Justification	Participant ID
1	Mixins allow CHGV to only adopt SVSM's get_qs while I, A, D adopt both SVSM and TIVSM's get_qs.	"CHGV adopts only TVSM, while all other adopter classes adopt both SVSM and TIVSM; SVSM return queryset with team data after validating its access against teamid; TIVSM checks for license and if valid return tagged objects in queryset else return default queryset without tagged objects. ... get_queryset method in TISVSM AND SVSM."	1	The P3-post data clearly states that CHGV adopts only SVSM, while I, A, and D adopt both SVSM and TIVSM, and explains their functionalities (e.g., get_queryset for team validation and license checks). This directly addresses the rubric, meeting all requirements.	P3 Post
7	Stating that I, A, D's get_qs are trying to visually tag data for enterprise users.	"For example one method in one class (InsightViewSet) which inherited this Mixin, modified this method in such a way that it only loaded tags onto the data if the User requesting for the data was licensed."	0.5	The text explains the use of mixins, modularity, and tagging functionality but does not explicitly tie Insights, Actions, Dashboards, and get_queryset into a unified framework for enterprise-specific visual tagging.	P6 Post
18	SVSM/TIVSM's get_qs access the model data from the children I, A, D ViewSets due to the children's super calls.	"mixins are the blueprint of objects here things are optional ... As mixins are created like that it can adopt the features ... we can edit the mixins also if needed ... As in mixin we can reuse the code and here it also supports multiple inheritance ... with mixin things are loosely coupled."	0	The participant fails to link the access of model data from the children I, A, D in any way, and they don't mention super either.	P1 Post

Table H9. MV* Architecture Rubric Items (17–31) for RM Evaluation with Detailed Scoring Criteria

ID	Pattern	Partial Credit
19	Insights, Dashboards, Actions are all persisted as models with various fields.	0.5 for mentioning some of the models but not all; 0.5 for mentioning models but not describing the persistence aspect (mention of a db is sufficient)
20	Mentions the MV* controllers InsightViewSet, DashboardViewSet, and ActionViewSet.	0.5 for mentioning some but not all 3
21	Mentions URL path or api endpoints of /insights, /dashboards, and /actions.	0.5 for mentioning some but not all 3
22	InsightViewSet accesses the Insight model, mainly just reading data, in the code we're viewing at least. Same for Actions. Same for Dashboards.	0.5 for mentioning many of the models and the fact that data is surfaced from the model into the view, but not (a) the aspect of models and viewsets being connected, or (b) not contextualizing this insight with the specific models we have in the application
23	InsightViewSet filters/tags data (of individual insights) from the model. Same for Actions. Same for Dashboards.	0.75 for all but saying "same for actions/dashboards" e.g., failing to say this)
24	InsightViewSet then sends that data to the templates which is how we see the data (/insights /dashboards /actions). Similar for Actions, Dashboards.	0.5 for just mentioning that controller data is sent to the view; they could just link the models and view via the controller, in their explanation
25	MV* provides separation of views (/insights /actions /dashboards), controllers (InsightViewSet, ActionViewSet, DashboardViewSet), and models (Insight, Action, Dashboard).	0.25 for just mentioning the modularity/separation in general but not naming the specific Insight/Action/Dashboard details in their overall answer
26	Modular in that the Insight, Action, and Dashboard models can be reused by other controllers / views; better if they give an example of one.	0.5 for describing reusability and extensibility without explicitly calling this point out.
27	Says the Insight/Action/Dashboard models, their views, and their controllers, are more easily tested given the separation of concerns. Same for maintainability.	0.5 for not explicitly mentioning the specific models though
28	Saying PostHog has a complex codebase, so MV* helps with: maintenance, collaboration of multiple developers. Ideal answer: for a user analytics dashboard, one needs to keep track of many complex pieces such as user data of the site being analyzed, user data of the team users logged into the analytics dashboard itself, tagging and data permissions, etc. (in a way that links both mixins and MV*).	0.5 for just mentioning maintainability
29	PostHog is an MVT architecture.	0.5 if they are confused between MVT and MVC; 1 if they say it's MVT, but they mention MVC somewhere, but they are sure it's MVT
30	Chosen for loose coupling—multiple developers can collaborate on the open source project.	
31	PostHog's complex functionality, maintained by many developers, makes the linear readability and extreme beginner friendliness of vanilla PHP less of a goal than some of the other goals described here.	0.5 for discussion of readability; 0.25 for justifying complex functionality as a reason for MV*

Table H10. MV* Rubric Examples for RM Evaluation

Rubric Item ID	Rubric Item	Example Snippet	Score	Justification	Participant ID
21	Mentions URL path or api endpoints of /insights, /dashboards, and /actions.	"Views: They are used to manipulate the data, they receive via apis (routes like /insights, /dashboard. /actions etc.) to display the data to the user."	1	They mention the relevant routes.	P2 Post
31	PostHog's complex functionality, maintained by many developers, makes the linear readability and extreme beginner friendliness of vanilla PHP less of a goal than some of the other goals described here.	"MV* helps us separate the visualization, processing and data management inside the posthog codebase. This is good for complex project like posthog. ... Since post hog is a complex project, MV* makes it easy to maintain since a frontend developer can maintain views and a backend developer, models and controllers. ... Monolithic architectures are easy to break ..."	0.25	They don't talk about any advantages of monolithic alternatives (e.g., linear readability for one-off scripting). They don't talk about readability.	P8 Post

Table H11. Rubric Examples for SE Evaluation

Rubric Item ID	Rubric Item	Example Snippet	Score	Justification	Participant ID
2	Each image tile has a unique name—name is being used to determine the image's src.	"The name of the image is generated using the center indices and the corresponding y and x values ... The generated name is used to determine the tile_name at a specific location. ... When one navigates through the comic, the position variable is used to calculate and determine incoming tiles which are determined and assigned to the image variable which paints them to the map variable, positions > center > center_last, tile_name > image ... generate a new image with the name and assign it to the image variable and append it to the map variable."	1	Here, the candidate mentions unique tile names and even how they are used to dynamically load the images to be displayed.	P10 Post
10	The left and top CSS attributes of \$map are set by the position variable. Even if they mix up left and right or top and bottom, it is acceptable.	"clientY and clientX is used to update the image position. ... #scrollingImage position: absolute; ... The translate CSS property is now used to move the image ... update Image image position ... image.style.transform = 'translateY(' + position + 'px)';"	0.5	The candidate does not talk about top/left/bottom/right, just about the CSS properties of images such as position.	P4 Post

Received 7 June 2025; revised 7 January 2026; accepted 22 January 2026