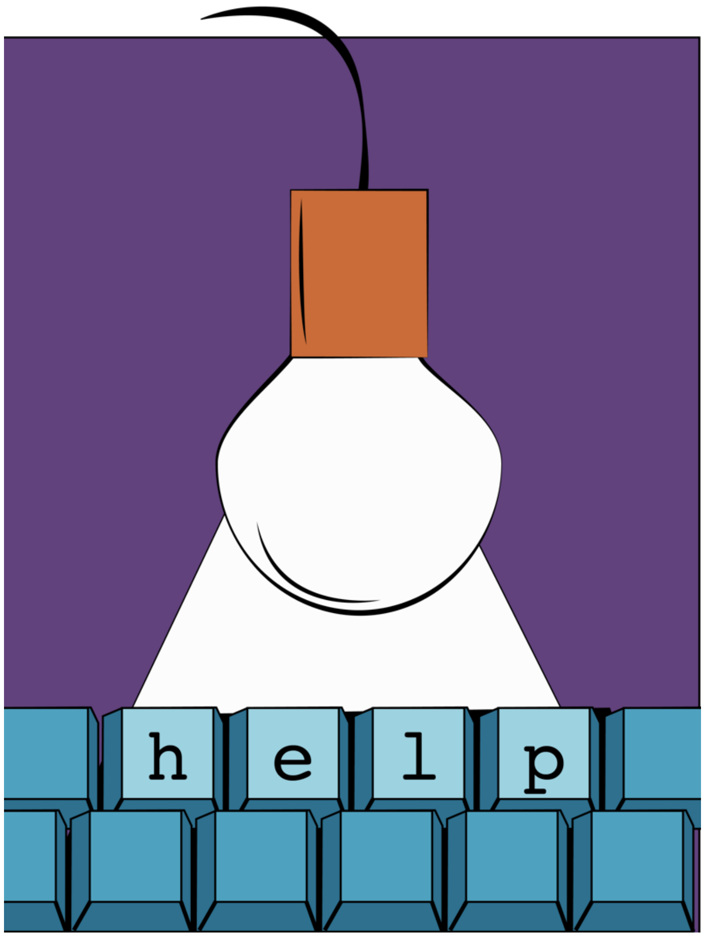


“It’s Like Coding in the Dark”:

The need for learning cultures within coding teams

Catherine Hicks
Catharsis Consulting
2022

Suggested citation: Hicks, C. *It’s Like Coding in the Dark: The need for learning cultures within coding teams* [White Paper], Catharsis Consulting. [link]

EXECUTIVE SUMMARY

- This report presents data from a qualitative research project with software engineers and developers. Twenty-five full-time code writers completed a "debugging" task and an in-depth interview on their learning, problem-solving, and feedback experiences while onboarding to an unfamiliar, collaborative codebase. This report applies a learning science lens to inform an understanding of how to help code writers can thrive and collaborate.

- Themes from these interviews revealed an important learning tension: the work that code writers needed to do to *understand* code often did not feel like what was rewarded in the *evaluation* of their work. Code review often did not recognize code writers' effort when it did not result in lines of code. Despite stated ideals about knowledge sharing (e.g., documentation and collaboration), this work was often contradicted with negative cues from colleagues about what was "truly" valued. This tension was exacerbated by code writers' fears about "not looking like an engineer," and their desire to perform to the expectations of their environments.

- Code writers navigated this by divesting from their own learning and from the "invisible" work of knowledge transfer, leaving future collaborators without guidance in their own ramp-up to unfamiliar code. As a result, code writers frequently expressed a poignant loneliness, even in highly resourced teams.

- This maladaptive cycle can be understood as *Learning Debt*. Research from learning science describes how environments that discourage sharing mistakes and valuing "in-draft" effort lead to long-term costs in people's motivation, wellbeing, and learning. Under this discouragement, even formal processes which are ostensibly meant to maintain productivity and provide support (e.g., code review, conversations with senior code authors) can reinforce these negative norms.

RECOMMENDATIONS

- **Involve people in defining how their "success" is measured.** Code writers across all career levels had strong feelings and insights about meaningful work, and frequently spoke to specific tasks they knew were valuable that their workplace missed.

- **Encourage more developmental feedback, separate from performance evaluation.** Critiquing output is necessary in a production environment. However, code writers spoke to how their own personal development *and* support for others' development was rarely given space or credit in review experiences. Leaders should make space for learning and development goals, and include knowledge sharing work as an important output. These efforts should be separate from performance evaluation. Collaborative learning requires psychological safety, and learners cannot experience the freedom of openly sharing mistakes and "in-draft" learning while defending their expertise and finished work.

- **Give technical teams more time for collaboration and documentation, and make documentation "count."** Simply put, documentation and other "mundane" tasks of knowledge sharing were the first sacrifice to time pressure.

- **Identify opportunities for celebrating and sharing collaborative support and examples of active problem-solving.** Many code writers described a strong desire to share their problem-solving, but lack of opportunity. Junior code writers also noted the outsize impact that senior teammates had on learning culture and the desire to "learn how to learn" from seeing real problem-solving in action.

- **Make the costs of learning debt visible.** The costs of discouraging learning are borne most immediately by individuals. These effects are almost certainly compounded and felt more strongly by people with marginalized identities, who are systematically less supported at work. Yet learning debt's cost can be invisible in short-term, conventional productivity metrics, particularly when these metrics fail to measure understanding and collaboration. For researchers, leaders, and practitioners working on driving change in these environments, it is important to think broadly about how to measure the health and long-term impact of a learning culture.

Authorship: Catharsis (www.catharsisinsight.com) is an evidence science consultancy which provides both strategic innovation consulting and applied behavioral research projects. We frequently work on topics of social impact, learning, data science and equity, learning, hiring and assessment, and health. Occasionally, we are able to conduct research studies on topics of interest, with a goal toward contributing to behavioral science and human wellbeing in our fields of expertise. In this report, we wanted to explore how a learning science lens could deepen our understanding of code writers' experiences.

INTRODUCTION

Collaborating on code is a difficult but necessary precondition to the production and maintenance of current technologies. Modern software engineering requires not only code creation, but sharing and iterating on this creation between multiple authors, over different timepoints, and within multiple decision contexts. This complexity has resulted in many classic topics around managing the knowledge work of code: tech debt, developer productivity, and unique needs in tech management training are some examples that merely scratch the surface of research on code collaboration (e.g., Edwards, 2003; Kalliamvakou, et al., 2017).

However, much of this conversation has focused on technical systems and productivity output. As a learning scientist who has worked with many engineering teams, I became deeply interested in what a *learning* lens could bring to questions of code collaboration: what could we hear from how code writers[1] perceived their own learning and the expectations of their environment around learning? Why do so many code writers struggle to learn, even in highly resourced environments which have many formal processes for code review and many stated ideals of life-long skill development and growth? And can people involved in coding teams, from junior members to senior leaders, use insights from learning science to improve their experience in this work?

To explore these questions, I interviewed 25 full-time code writers on their experiences working within collaborative codebases. These interviews provide a window into the lived experience of code writers' collaboration, their reflections on barriers to it, and their strategies for overcoming these barriers. In these interviews, code writers share *how they see their own learning*, their strategies and experiences of moving from *code observers* to *code creators*, and where they experienced barriers on this journey.

Overall, participants' lived experiences of learning looked very different from their workplaces' stated ideals.

Regardless of seniority, participants spoke to paradoxes in navigating an ebb and flow of *scrutiny* and *isolation* during code writing. Nearly every participant reflected that a key learning strategy for successful work was creating an *accurate mental model of an unfamiliar codebase* and *exploratory programming*, and yet most perceived this activity as far less incentivized and valued in their environment than it deserved. Even in highly resourced environments–most of the code writers worked at large high tech companies with many explicit review structures–participants felt that *learning* during code writing was *lonely*, *difficult*, and *undervalued*.

Strikingly, code writers described a pressure to *perform* coding output that created an implicit discouragement of spending time on work that they knew would support future learning and future collaborators. One code writer described the process in a quote that led to the title of this report:

> *"It's like coding in the dark. Every once in a while someone comes in to turn on the lights and stare at you, like review, but then you feel like you have to defend something. But mostly I feel like I'm just sitting here with all the lights off."*

Summarizing across these lived experiences, I use the term ***Learning Debt*** to describe the consequences of devaluing code writers' learning.

[1] There are many debates about how to label different roles which produce code. The majority of our participants identified themselves as "software engineers," but several, who were junior in their careers or pursuing roles that they saw as hybrid (e.g., data science, QA), debated their own belonging in this label. These debates were playful, wry, and insightful, and I wanted to reflect that. Therefore I refer to participants as "code writers," in order to center experience more than job title. Every participant had a core responsibility to produce collaborative code in their role.

Paralleling tech debt, learning debt is a cumulative failure to support learning, created when code writers' investment in long-term understanding is disincentivized. For example, when code writers experience code review scenarios as focused on criticism and legitimacy tests, rather than effort and development, code writers feel forced to defend their reputations and output and disguise large portions of their "true" work. Compounding the learning debt, code writers make choices to keep reflective, consumptive, and conceptual learning *buried and individual*. Learning debt further accumulates in a workplace as code writers feel discouraged from sustainability efforts such as teaching, documentation, or initiating helpful dialogue. As new collaborators enter this cycle, they must once again "code in the dark."

It is important to note that the focus of these interviews was not on workplaces that were described as punitive, hostile, intolerable or adverse. Those environments certainly exist. But the code writers interviewed in this project spoke of valuing their work and deep engagement within their roles. Even so, learning debt was evident throughout their experiences.

Despite this struggle to maintain room and time for learning, code writers spoke to the value of learning and their desire for learning moments to be more capturable. Their solutions were grounded and simple: rather than requests for complex tooling or technological silver bullets, most code writers expressed a poignant desire for *more time to learn, more time to talk to people*, and *more recognition of the learning and reflective work that drove an accurate understanding of collaborative code.*

METHODS

Learning Science background to the interview script

People continuously seek out feedback from their environments about the expectations others have for their own learning (Schunk, 2012). In these interviews, I wanted to learn more about how this behavior happened in code work. *Metacognition* is a general term for beliefs that people hold as they reason about their social environment, their skills, and regulate their own behavior (Veenman, 2006). In learning, many of these metacognitive beliefs can be connected to behavioral decision-making. An example of a metacognitive belief that code writers might hold is whether or not mistakes are proof of incompetence. In a highly *performance-oriented* environment, people avoid sharing in-draft work (e.g., Harackiewicz, Barron, et al., 2000). With this in mind, I focused interviews on participants' stories of learning as well as what they felt those experiences told them *about* learning.

In pilot testing the interview script, I identified several defining experiences where code writers might reveal metacognitive beliefs about their learning environments. These experiences were 1) making mistakes or errors and repercussions for them 2) seeking out help-seeking and disclosure 3) tools used for self-regulation, storing knowledge, and making forward progress through unfamiliar tasks. The resulting interview script can be found in Appendix A.

Co-designing consent

Twenty-five code writers were recruited from direct email to listservs, forums, and from research participants who invited participation to their friends, colleagues, and/or students via email. As this was not intended to be a representative quantitative study, I used social recruiting as a starting place, but did not know any of these participants personally. All participants worked at a workplace with headquarters in the California Bay Area, but six lived in other locations in the United States and worked remotely.

In line with Catharsis' research principle of co-designing with participants (Greenhalgh, Hinton, et al., 2019), I led each participant through a *consent dialogue* along with a more conventional consent form. In this dialogue, I explored how participants wanted their stories presented. This enabled me to not only ensure informed consent for these discussions about personal and emotional work experience, but also ask how participants preferred any identity information to be presented. While participants represented a diverse range of identities, after this discussion, some participants preferred their specific nationality, ethnic, and/or racial identification to not be shared in the context of research on their work. This category is therefore excluded for all.

When possible, it is also a Catharsis analysis principle to present combinatorial characteristics to better reflect intersectional experiences (e.g., “10 female participants, of which 3 were junior career”; see Cole, 2009). However, for greater confidentiality in this study, participants preferred characteristics presented separately. Table 1 describes participants according to these co-designed preferences. Most participants were full-time employees of their workplaces, but four participants were PhD students completing a full-time internship at a large tech company.

Table 1: Participant Characteristics

Gender		Employer		Experience	
Male	15	Large tech company	10	Junior	9
Female	8	Code Bootcamp	3	Mid-career	10
Nonbinary	2	Startup	8	Senior	6
		Large tech company (internship)	4		

Research session preparation

Code writers completed hour-long sessions which contained both an active work session and a semi-structured interview. Most sessions were conducted remotely over video call, but five were conducted in person. Qualitative interviews are social and dynamic, and participants’ reflections may sometimes bring up sensitive experiences they did not expect to disclose. In order to respect this, I also concluded each interview with an additional *consent dialogue*, checking with participants on their comfort with sharing topics that had surfaced during the conversation. Participants were allowed to withdraw any content from the study at any time, with no questions asked; all code writers felt comfortable re-assenting to participation after the interview was finished and full interviews were included in the analysis.

In preparation for the session, code writers were asked to focus on the process of moving from an unfamiliar observer to familiar contributor in a collaborative codebase. I asked code writers to focus on three touchpoints:

> 1) Moments they were onboarding to (or “ramping up in”) an unfamiliar codebase.
>
> 2) Moments they began to write their own decision into a collaborative codebase, and moments they solved a “bug” within a collaborative codebase.
>
> 3) How code review and any other feedback points were experienced in this problem-solving process, and how they shared back their problem-solving with collaborators.

These tasks are not mutually exclusive. For instance, many code writers described working on discrete debugging tasks as a means of continuing their overall “ramp-up” into understanding a codebase. Code writers would often review shared code feedback histories as they explored. It was common that code writers would discuss switching between tasks, or refer to a previous code review from a different area of code to aid in understanding a new collaborative codebase. However, in pilot testing the interview script, I found that these three topics were helpful focus points that elicited conversation about the learning journey of encountering unfamiliar code.

Structure of research sessions

Sessions had two components: an active coding task to establish trust and prompt context, and a semi-structured interview to dive deeper into participants’ experiences.

1. First, participants explored an *active work session.* Prior to the session, participants were prompted to bring a real code task which they could focus on during our time. Active work sessions ranged from 15-30 minutes, determined by the participant’s comfort and when they felt “done.” Where needed, participants also obtained the permission of their supervisor for this conversation. For the sake of confidentiality of their work product, I did not observe any lines of code. Instead, I used a “talk aloud” methodology (Landauer, 1988), and participants narrated their decisions and explorations during the example task. The majority of participants worked on a debugging task, while several brought in a task such as exploring a particular function, or researching the history of connections within several layers of dependence in their codebase.

Like all measurement, interviews are not a perfect reproduction of experience and interview answers should also be understood as an operationalization. In pilot tests of the interview script, several code writers reflected that it was difficult to answer questions in the abstract about problem-solving, and that translating "code thinking" into words could feel like a specific skill on its own. This echoes work from Human-Computer Interaction research on how even important collaboration during code search sometimes builds on nonverbal social communication (e.g., D'Angelo & Begel, 2017).

One challenge inherent to any study on these topics is that people do not always find it easy to accurately self-assess their own behaviors and beliefs about learning. Focusing on "in-draft" work rather than a final product or output has been found to prompt deeper consideration of the problem-solving at hand, versus focusing on surface issues or "aesthetics" (e.g., Hicks et al., 2016). Similarly, "process feedback" which considers *how* work is done, not just *what* work is done, prompts deeper behavioral impact and motivation, along with diminished threat (London & Smither, 2002). I hoped that by accompanying participants in a real task, the interviews would benefit from drawing attention to their process.

I was also aware that the interviews asked participants to reflect on uncomfortable experiences, such as moments they felt confused, distracted, or disheartened in code writing. Another research principle of participant co-design at Catharsis is to consider psychological safety a core requirement of our methodologies. Therefore the aim of this active work session task was twofold: to help elicit more accurate conversations by grounding code writers in their own active problem-solving, and to create an atmosphere of reflection and psychological safety as code writers shared work behaviors that were seen as imperfect, incomplete, or messy, and as a researcher, I could model a respectful appreciation and unconditional observation of their work.

And indeed, as participants narrated their work during the active work sessions, the tone of their conversations moved from initial "ideal" descriptions of protocols around code work, towards more tangible, authentic, "how it really works" commentary (see Appendix B for further comment on this).

2. Second, participants completed a semi-structured interview. I used a core bank of questions to probe into code writers' experiences with learning while working with collaborative code, while allowing code writers to elaborate and expand on any of the threads that were brought up either by the questions, or by the active work session (Appendix A).

Themes from all session content were identified using open coding (e.g., Maguire & Delahunt, 2017; Vaismoradi, Turunen, & Bondas, 2013). While not every theme came up in every interview, in order to be counted as a theme, a topic had to be mentioned by at least 23/25 participants. As with many qualitative interview projects, this approach was inductive and exploratory, and is meant to serve as a reflection point for noticing common patterns in what are inherently individual experiences (Gibbs, 2007; Braun, & Clarke, 2012).

FINDINGS

High Level Summary. This section describes the overall themes from these interviews. Themes were synthesized into three main groups, mapped onto the journey participants described in chronological stages of increasing code understanding: 1) *active learning*, 2) *code review*, and 3) *environment reinforcement*. In each stage, learning challenges emerged not only from individuals, but from the constraints of their workplaces.

1. In the first stage, code writers described the ramp up to an unfamiliar codebase as a process of detecting an implicit mental model of the decisions that built that codebase. Code writers did this with **active learning.** Active learning is experiential, a process of learning by testing conceptual mental models with tangible examples (Prince, 2004). One shared description from nearly all participants was moving from iteratively "breaking" small pieces of code or small logical connections before one was ready to enter "production." Active learning was more nonlinear than eventual code output. It required back-and-forth detective work to find clues for previous coders' decisions, such as identifying critical "anchor points" in code from which to explore.

Code writers narrated active experiments such as producing errors, misconceptions, and "toy" versions of code that were continuously tweaked and iterated. The majority of this exploration did not yield code creation that would be shared with others. Instead, code writers focused on scaffolding their understanding until they felt confident they could contribute. Code writers described on this stage as necessary, highly valuable, and laden with *productive mistakes*. However, this work was also described as largely unshared, invisible to others, and subject by anxiety about time pressure.

2. In **code review,** a significant theme that emerged was *fear of disclosure*. Code writers emphasized that the feedback in these formal processes frequently did not match their active learning of their previous stage. Instead, many review moments put social pressure on code writers to justify their output, rather than have a developmental conversation about learning. Awareness of time constraint further forced code writers into choosing between trade-offs between performing a "productive" identity, versus accurately discussing the work behind the code. Sharing *the decision-making behind the code* was disincentivized, and the productive lived mistakes of active learning were rendered invisible.

3. As a consequence of this tension, code writers returned to their code creation **environment** to experience reinforcing cues which underlined the divide between solitary learning and their strategic, outward-focused performance. Against a background of persistent time pressure and organizational barriers like asynchronous communication, code writers spoke of reducing documentation, reducing the capture of decisional context, and remaining vigilant to social cues in inconsistent or hostile social communication and feedback.

Top Themes across Interviews
Total is a count of distinct mentions within the theme topic

Themes	Total
Active Learning	
Breaking things to test assumptions	42
Making hidden context explicit	33
Getting exposure to specific examples of abstract ideas or principles	30
Personal skill development in tools and languages	21
Identifying anchor points for understanding	19
Sharing mental models with collaborator(s)	17
Code Review	
Fears about disclosure in code review	39
Guidelines not matching reality in code review	32
Protecting against reputation threat in code review	15
Environment Reinforces Learning Loss	
Inconsistent, skeptical or hostile communication from colleagues	31
Lack of time/or time delays	25
Remote communication, asynchronous communication	17

Not all descriptions of learning during code work were negative. Where positive, code writers described a rich mental and emotional benefit from shared understanding, knowledge scaffolding, and valued their own skills at creating these structures as well as sharing them with others (e.g., in pair programming).

However, most conversations underlined even these descriptions with the awareness that this productive learning activity was fundamentally *covert* compared to "final product" code output.

Deeper Dive. This section explores how research on both individual learning beliefs and learning environments can help to contextualize code writers' experiences. Cues from code writers' environments reinforced metacognitive beliefs that emphasize performance over process. Subthemes are explored in this deeper dive and related to supporting research on learning. For **Stage 1,** themes about the effort required to develop accurate mental models are supported by research on obtaining *tacit knowledge* and transitioning toward *expert problem-solving.* For **Stage 2**, themes around frustration with a code review focus on *performance* and *output* over developmental work and knowledge work that is not reflected in "lines of code" are supported by research on *essentialism* and metacognitive beliefs about *brilliance.* And for **Stage 3**, themes around inconsistent cues about learning and social cues are supported by work on the impacts of *maladaptive performance cultures* and their discouragement of learning.

Stage 1. Active Learning: *direct, tangible, experimentation-driven learning, typically done alone.*

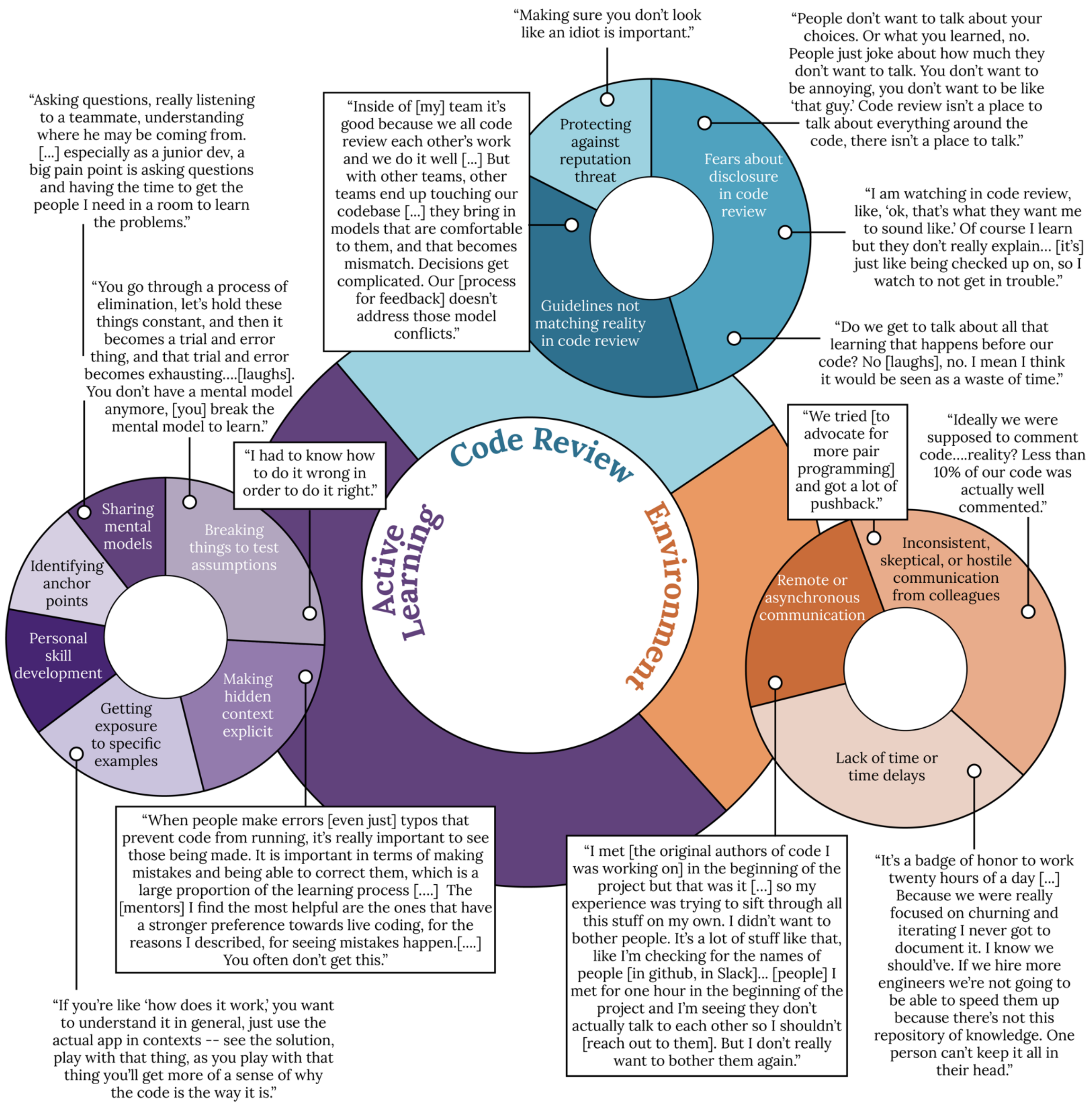
Active learning was defined by code writers as acquiring conceptual understanding, and then testing their conceptual mental model with tangible experimentations via a process of *breaking things in order to test assumptions.* This activity was central to developing an understanding of code. It was also highly observational and reflective. Code writers emphasized the need to observe, learn, and systematically test prior to explicit code creation that would be visible to others.

Learning environment lens: *Active learning, reflection, and "deep processing" without code production are necessary first steps for code writers to feel confident in becoming code creators.*

The learning strategies described by participants as the most efficient way to gain code familiarity were not surprising. Our participants' focus on and love for active problem-solving echoed the large body of research on the cognitive aspects of programming. (e.g., Détienne, 2001). For example, programmers use just-in-time searches and "interleave" code writing and web search as a strategy for both learning and remembering (e.g., Brandt, et al., 2009).

Similarly, from learning science, research on the *tacit knowledge* needed for domain expertise argues that many of the highly skilled problem-solving patterns experts go through are neither documented nor explicit in a work environment (e.g. Fritz, 2014; Ryan & O'Connor, 2009). And active learning, such as using specific worked use cases, role plays of solutions, and just-in-time learning, can be a beneficial way to learn in comparison to rote lecture or passive material reading (e.g., Freeman, 2014; Yang, 2014).

This is also a productive strategy for collaborating in code. Expert programmers "work out" solutions, pushing forward their knowledge with both top-down and bottom-up explorations (Rist, 1991). Juniors in these environments therefore face the continual challenge of interpreting artifacts of previous work, often without access to the invisible decision-making processes that created them. This active exploration of how code reflects conceptual decision-making in both obvious and subtle ways was described by our participants as a fundamental detective work of code collaboration.



"Making sure you don't look like an idiot is important."
"People don't want to talk about your choices. Or what you learned, no. People just joke about how much they don't want to talk. You don't want to be annoying, you don't want to be like 'that guy.' Code review isn't a place to talk about everything around the code, there isn't a place to talk."
"Asking questions, really listening to a teammate, understanding where he may be coming from. [...] especially as a junior dev, a big pain point is asking questions and having the time to get the people I need in a room to learn the problems."
"Inside of [my] team it's good because we all code review each other's work and we do it well [...] But with other teams, other teams end up touching our codebase [...] they bring in models that are comfortable to them, and that becomes mismatch. Decisions get complicated. Our [process for feedback] doesn't address those model conflicts."
Protecting against reputation threat
Fears about disclosure in code review
Guidelines not matching reality in code review
"I am watching in code review, like, 'ok, that's what they want me to sound like.' Of course I learn but they don't really explain... [it's] just like being checked up on, so I watch to not get in trouble."
"You go through a process of elimination, let's hold these things constant, and then it becomes a trial and error thing, and that trial and error becomes exhausting....[laughs]. You don't have a mental model anymore, [you] break the mental model to learn."
"Do we get to talk about all that learning that happens before our code? No [laughs], no. I mean I think it would be seen as a waste of time."
"I had to know how to do it wrong in order to do it right."
Code Review
Active Learning
Environment
"We tried [to advocate for more pair programming] and got a lot of pushback."
"Ideally we were supposed to comment code....reality? Less than 10% of our code was actually well commented."
Sharing mental models
Breaking things to test assumptions
Identifying anchor points
Personal skill development
Getting exposure to specific examples
Making hidden context explicit
Remote or asynchronous communication
Inconsistent, skeptical, or hostile communication from colleagues
Lack of time or time delays
"When people make errors [even just] typos that prevent code from running, it's really important to see those being made. It is important in terms of making mistakes and being able to correct them, which is a large proportion of the learning process [....] The [mentors] I find the most helpful are the ones that have a stronger preference towards live coding, for the reasons I described, for seeing mistakes happen.[....] You often don't get this."
"I met [the original authors of code I was working on] in the beginning of the project but that was it [...] so my experience was trying to sift through all this stuff on my own. I didn't want to bother people. It's a lot of stuff like that, like I'm checking for the names of people [in github, in Slack]... [people] I met for one hour in the beginning of the project and I'm seeing they don't actually talk to each other so I shouldn't [reach out to them]. But I don't really want to bother them again."
"It's a badge of honor to work twenty hours of a day [...] Because we were really focused on churning and iterating I never got to document it. I know we should've. If we hire more engineers we're not going to be able to speed them up because there's not this repository of knowledge. One person can't keep it all in their head."
"If you're like 'how does it work,' you want to understand it in general, just use the actual app in contexts -- see the solution, play with that thing, as you play with that thing you'll get more of a sense of why the code is the way it is."

Figure 1. Illustration of mentions per theme, as in the table of top themes. Within each of the three main categories (Active Learning, Code Review, and Environment), subthemes are shown alongside selected quotes that fell under the subtheme. Additional quotes can be found in Appendix B.

Stage 2. Code Review: *reputation and disclosure bring up conflicts between the guidelines and the real social experience.*

While code writers spoke of numerous types of feedback from their peers and teams, including pull requests, internal messaging, and internal q&a processes, *code review* emerged as the key focusing point for our participants' discussion of gaining feedback on their learning.

Code review was mentioned over 39 distinct times, and was a central moment when tensions arose between learning goals and the performance expectations of code writers' environments. Code writers spoke to many layers of decision-making, not simply about the explicit choices in the code, but on their meta-choices about what was appropriate to bring to the attention of another code writer.

Most of these mentions were critiques. This is not necessarily surprising, as the interview questions directly prompted participants to reflect on moments when they felt they needed more help. This emphasis should not be taken as an overall evaluation of the efficacy of code review itself or an investigation of its rubrics. Yet it was clear to our participants that *formal processes for getting feedback on code faced significant implementation challenges.*

Code writers doubted the *stated goals of review.* Many had the sense that they needed to care more about "sounding like" or fitting the expectations of a reviewer, than accurately discussing their work.

Most code writers agreed that code review processes did not include reward or recognition for their active learning work and development of code understanding.

Because of this, the majority of interviewees surfaced frequent experiences of feeling bereft of both meaningful feedback and a way to communicate those insights to others who might be helped by them.

Transitioning from solitary knowledge work and problem-solving to shared understanding was a complex task. Code writers at all levels of experience spoke with fondness, irritation, and eloquence of how often this task fails. Where review experiences felt positive, participants dwelled on gaining confidence in the *implicit mental models* behind codebases, *getting feedback about the appropriateness of their effort*, and being able to pass this knowledge on to others.

Learning environment lens:
Reputational costs will supersede "developmental" ideals; when code reviews focus on performance over learning, code writers feel pressured to hide their actual learning.

One pattern underlying the themes in this stage was that code writers described many conversations ostensibly focused on feedback feeling like covert 'tests' of their legitimacy in engineering. Code writers therefore could not experience these as opportunities to freely discuss and learn from mistakes.

These experiences echo research on the negative consequences of a performance-oriented culture. In performance-oriented learning cultures, only "final" outputs are acceptable to others, and performance of external metrics such as grades is valued more than mastery of the problem area (Harackiewicz, 2000). Code writers' worry around *sounding like an engineer* or *not getting in trouble*–particularly expressed by junior code writers–echoes research on metacognitive beliefs that some people are "born" good at x, where x might be math, code, or any other technical skill. Many STEM fields fall into the trap of believing that skills come from static, deterministic "brilliance," including engineering (e.g., Cimpian & Leslie, 2017; Meyer, Cimpian & Leslie, 2015).

These metacognitive beliefs are not only maintained by individuals, but also by environments. When formal feedback processes in an environment discourage sharing mistakes, work-in-progress, and difficulty, this can reinforce a "fixed" mindset and the implicit belief that learning activity should remain hidden. Under this environment code writers struggle to share "real" work in its totality, as it may create the impression that a learner was not "born brilliant" (see Canning et al., 2020).

Code writers' learning persisted despite this challenge. Where scrutiny, evaluation and identity threat was covert, the learning strategies of code writers also became covert. Code writers spoke of continuing to invest time and effort in collaborative understanding outside of untrusted formal processes. This learning was buried in personal notes, informal peer-to-peer support, and just-in-time conversations. When choosing what to disclose in formal feedback conversations, code writers emphasized the lack of space for explanations and conceptual decision points.

The "invisible" byproduct of much of learning, mistakes are a key component to advancing understanding. When code review was positive, code writers spoke to the inclusion and valuing of these "invisible" processes. However, it was more frequent that formal processes washed away the "background noise" of mistakes, and enforced erasure of exploration. Code writers were uncertain where this learning should "live" in their collaborative workplace, and were left feeling that much of their most insight-generating effort went unshared.

Stage 3. Environment Reinforces Learning Loss: *tension between learning and performance leads to loneliness.*

Themes in the third stage were less tied to a specific task, and more defined by code writers' reflections on the environmental barriers around them. Despite working in highly resourced environments, code writers went back to code work having experienced a dearth of contextual information and communal support. On the one hand, code writers were told that they were meant to find support in large formal processes for review, feedback, and implementation. But on the other hand, their navigation of these processes revealed profound context gaps in their organizations.

This paradox crystallized the third constellation of themes: code writers themselves learned to reinforce the divide between valuable but secretive learning, and collaborative knowledge sharing. They made choices to cut their learning off from sharing, collaboration, and knowledge storage, such as not creating documentation, not commenting code, and not reaching out to colleagues.

Code writers' reflections on their environments surfaced barriers that might sound familiar to any worker at any large organization: *time pressures*, *asynchronous communication*, and *inconsistent access to experts*. Expressed in relationship to producing code, however, these became further clues about the implicit expectation that *sharing the process was not a smart strategy*. Code writers made pragmatic, and wry observations on how it was unstrategic to capture and share their own conceptual learning about the collaborative codebase, even when they knew it could help others. In these choices, code writers themselves recognized that they reinforced the learning norms of an environment that had already put them "in the dark" as learners.

Learning environment lens: *learning culture can be improved by rewarding "invisible" learning work that helps others, such as pair programming and documentation. Conversely, learning culture is damaged by social cues that disparage this work.*

Discouragement and devaluing of learning has real consequences on both wellbeing and productivity. Further research on performance-oriented environments that discourage mistake and process sharing has shown that these strategies can look highly productive in the short-term, but ultimately result in long-term stress on learners (Harackiewicz, et al., 2008).

Focusing on output and productivity is a necessary concern for any workplace. However, devaluing the *process that builds sustainable work* is fundamentally demotivating. Coupled with this devaluing, metacognitive beliefs such as tying *identity to performance quality* can lead to deeply negative dynamics, such as thriving when one's performance matches the internal concept of a "performer," but crumpling when mistakes arise in the normal course of learning, as mistakes are not believed to be part of intelligence, performance, or achievement (Elliot & Church, 1997).

Signals for what metacognitive beliefs are expected can come from a day-to-day, working environment. Code writers spoke of using *infrastructure as a clue about performance and learning expectations.*

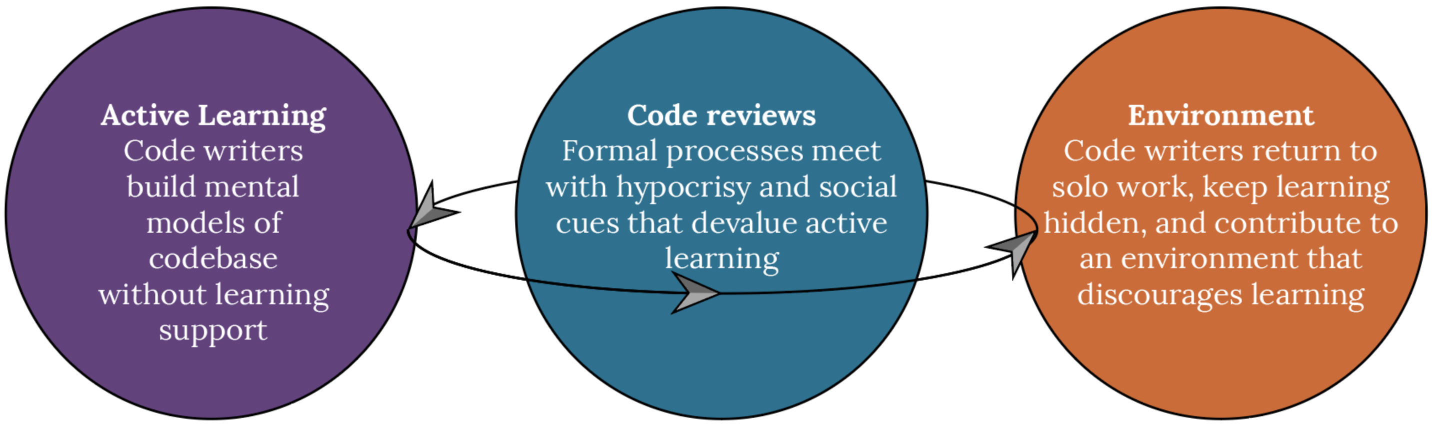
Code writers looked to artifacts left by other code writers, like documentation, pull request histories, and even the cadence of internal messaging, in trying to interpret what were acceptable ways to ask for help.

Code writers pointed out contradictions between spoken ideals and observed behavior when questioning *who* the artifacts of code collaboration were truly supporting. And when this contradiction was felt by code writers, the implicit message won out. Code writers spoke of reading between the lines of missing documentation and colleague irritation, and these signals served as a "gut check" against the ways they were told to ask for help. This experience was particularly vivid for code writers who were junior in their careers.

When to initiate shifts from solo work to group collaborations also felt uncertain. Many of the quotes on this topic emphasized the intensity of the fear that engineers felt in being "out of pace" with other workflows. For junior code writers in particular, this pressure felt deeply negative. Under this context of time pressure, it seems likely that code review was weighted with outsize pressure as a place to obtain help and subsequent discouragement when help was not centered, as a result of the *lack of other opportunities for feedback.*

Quite simply, the result of this conflict between performance goals and learning goals was loneliness.

Code writers reflected on moments they wished they could have talked to someone, frictions in experiencing "real" work versus performative work, and wishes to share the value of their unvalued learning. This loneliness was exacerbated by asynchronous communication and decision making. Code writers were often working backwards and across time and distance to infer the mental models behind collaborative code. For the code writers who were remote, the uneven access to artifacts created around code held even more power over their perceptions of the environment.


Active Learning
Code writers build mental models of codebase without learning support
Code reviews
Formal processes meet with hypocrisy and social cues that devalue active learning
Environment
Code writers return to solo work, keep learning hidden, and contribute to an environment that discourages learning

A Learning Debt Cycle

Ramping up to new collaborative code, code writers navigated a complex landscape of competing needs while problem-solving. And despite formal structures of feedback and review, many of these code writers expressed not only a lack of resolution between these competing needs, but outright hostility to their learning needs.

This experience can be described as a ***Learning Debt Cycle.*** Similar to "tech debt," the common metaphor used to refer to a wide range of accumulating unresolved problems in software development (e.g. Kruchten et al., 2012; Fairbanks, 2020), I use the term *Learning Debt* to center the knowledge, personal development, and expert problem-solving of code writers, rather than the efficiency of production or systems. Learning is a dynamic, long-term process which knits together mistakes, reflection, and observation, and these are activities frequently missed or outright obscured by short-term, efficiency-focused metrics.

For these code writers, learning was a necessary foundational activity that was often discouraged and rendered invisible.

The cycle of Learning Debt begins at a code writers' earliest encounter with a collaborative codebase. During this ramp up, participants took in *cues about organizational learning expectations* from both the code infrastructure itself (frequency of pull requests, authorship paths, conflicts between "ideal" and "real" solutions in code) and from social conversations and social cues (e.g., availability of senior engineers for questions, observations of how peers are treated). This is commensurate with effects found in learning science on the role of *initiating* encounters (e.g., syllabi language in college classes) in setting normative expectations which long-term effects on learners' expectations (Canning et al., 2021).

As code writers moved from a code observation and translation into code creation, they invested in necessary **active learning** strategies (including "breaking" code, "tracking" bugs, and building a mental model) to initiate learning. During code review, however, these strategies were often met with **conflicts from formal help-seeking processes,** which introduced contradiction, reputational pressure, and hostility to learning. As code writers returned to their working environments, these high-value feedback moments reinforced **a conflict between performance and learning goals,** and code writers were left to store and share their conceptual learning outside of the formal processes.

CONCLUSIONS & RECOMMENDATIONS

Collaborating on codebases is such a foundational activity that it is tempting for engineering teams to take this activity as a given. Teams might assume that “of course” code is commented, documentation is written, and knowledge is shared. But for the code writers I interviewed, real life differed significantly from the “best practices” described in engineering research. When I asked whether the described learning policies of their workplaces “worked as intended,” many code writers laughed before answering. Lack of documentation, frustration with invisible decisions, and lack of mentorship for junior teammates were frequently treated like rites of passage, perhaps even the inevitable consequence of high quality code work. As one code writer said: “*Having to comment something is a red flag.*” As another said, learning about new code “*fails all the time.*”

When code writers felt their work demanded learning that was not recognized or rewarded, this meant that code review and other feedback situations were filled with *psychological friction.* Psychological friction is an umbrella term for the cognitive and emotional burden created when people feel the need to attend to situational cues which signal a threat to their belonging, reputation, or wellbeing. This friction discourages learning. *Reducing* psychological friction has improved learning experiences for early career knowledge workers in STEM and Law (e.g., Quintanilla et al., 2020; Murphy et al., 2007). To multiple participants in this study, it felt like workplaces had removed *core pieces of learning.* Code writers spoke wistfully of what coding in a learning culture had felt like during their education, or rare instances of pair programming: “*Just being able to build off of others' experiences. I miss it a lot.*”

When there are so many clear benefits, why is it so difficult to maintain a learning culture? Social and psychological cultures at work are created by reinforced behaviors. In this study, code writers' “failures” to document or transfer knowledge were not driven by laziness or lack of care. Rather, code writers spoke to environmental pressures which pushed them to navigate complex tensions between performance and learning goals. And when making learning visible does not feel *safe*, performance culture wins.

This challenge is not unique to collaborating on code. Despite a wide-ranging research literature on the importance of learning culture, giving feedback on processes over output, and developmental collaboration, schools and workplaces often fail to invest in developmental feedback, or recognize that maladaptive performance beliefs are emphasized in their environments (e.g., Marsick & Watkins, 2003; Bian et al., 2018). This is a gap on both sides, as social science also frequently fails to provide enough concrete information for workplaces to learn from, even for research on applied interventions (e.g., Premachandra & Lewis Jr, 2020).

But academic research is not the only place to learn about learning. I interviewed people about learning in their real jobs, because I wanted to amplify their specific expertise and insight. The code writers in these interviews had many ideas about how to improve learning with code; they simply also felt discouraged and unsupported in translating this to their environment. Code writers had keen insight into their own productive, active problem-solving, but felt their environments, teams, and leaders failed to adequately understand this productivity.

Multiple areas of engineering research echo these experiences, and can be drawn on to improve them. Engineering environments struggle to understand, capture, and measure productivity, but thinking broadly about multivariate data, developer satisfaction, wellbeing and stress reduction can move us closer to good measures (e.g., Forsgren, Storey et al., 2021). Parallels can even be drawn from areas such as *resilience engineering*, which has made metaphorically resonant arguments that systems must expect and be designed for non-ideal (aka, “real world”) conditions (e.g. Madni & Jackson, 2009). Human beings are not machines, but like machines, they experience cycles of stress and negative impacts from their environments, and functional systems need to build for sustainability and protection.

It is important to note once again that the focus of this study was not on deeply harmful or hostile experiences. All of participants spoke of being happy where they worked. Nevertheless their *learning debt* was real, cumulative, and striking. In some ways, it was surprising that *even* code writers in highly resourced environments struggled to find basic systems of social and knowledge work support. Focusing on *code writers as learners* brought to the surface the ways in which their environments failed to treat learning as a meaningful activity in code collaboration, and failed to design for it.

RECOMMENDATIONS

- **Involve people in defining how their "success" is measured.** Code writers across all career levels had strong feelings and insights about meaningful work, and frequently spoke to specific tasks they knew were valuable that their workplace missed.

- **Encourage more developmental feedback, separate from performance evaluation.** Critiquing output is necessary in a production environment. However, code writers spoke to how their own personal development *and* support for others' development was rarely given space or credit in review experiences. Leaders should make space for learning and development goals, and include knowledge sharing work as an important output. These efforts should be separate from performance evaluation. Collaborative learning requires psychological safety, and learners cannot experience the freedom of openly sharing mistakes and "in-draft" learning while defending their expertise and finished work.

- **Give technical teams more time for collaboration and documentation, and make documentation "count."** Simply put, documentation and other "mundane" tasks of knowledge sharing were the first sacrifice to time pressure.

- **Identify opportunities for celebrating and sharing collaborative support and examples of active problem-solving.** Many code writers described a strong desire to share their problem-solving, but lack of opportunity. Junior code writers also noted the outsize impact that senior teammates had on learning culture and the desire to "learn how to learn" from seeing real problem-solving in action.

- **Make the costs of learning debt visible.** The costs of discouraging learning are borne most immediately by individuals. These effects are almost certainly compounded and felt more strongly by people with marginalized identities, who are systematically less supported at work. Yet learning debt's cost can be invisible in short-term, conventional productivity metrics, particularly when these metrics fail to measure understanding and collaboration. For researchers, leaders, and practitioners working on driving change in these environments, it is important to think broadly about how to measure the health and long-term impact of a learning culture.

REFERENCES

Bian, L., Leslie, S. J., Murphy, M. C., & Cimpian, A. (2018). Messages about brilliance undermine women's interest in educational and professional opportunities. Journal of Experimental Social Psychology, 76, 404-420. https://doi.org/10.1016/j.jesp.2017.11.006.

Braun, V., & Clarke, V. (2012). Thematic analysis. In H. Cooper, P. M. Camic, D. L. Long, A. T. Panter, D. Rindskopf, & K. J. Sher (Eds.), APA handbook of research methods in psychology, Vol. 2. Research designs: Quantitative, qualitative, neuropsychological, and biological (pp. 57–71). American Psychological Association. https://doi.org/10.1037/13620-004

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009, April). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 1589-1598). https://doi.org/10.1145/1518701.1518944

Canning, E. A., Murphy, M. C., Emerson, K. T., Chatman, J. A., Dweck, C. S., & Kray, L. J. (2020). Cultures of genius at work: Organizational mindsets predict cultural norms, trust, and commitment. Personality and Social Psychology Bulletin, 46(4), 626-642.

Canning, E. A., Ozier, E., Williams, H. E., AlRasheed, R., & Murphy, M. C. (2021). Professors Who Signal a Fixed Mindset About Ability Undermine Women’s Performance in STEM. *Social Psychological and Personality Science*, 19485506211030398.

Cimpian, A., & Leslie, S. J. (2017). The brilliance trap. Scientific American, 317(3), 60-65.

Cole, E. R. (2009). Intersectionality and research in psychology. American psychologist, 64(3), 170.

D'Angelo, S., & Begel, A. (2017, May). Improving communication between pair programmers using shared gaze awareness. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (pp. 6245-6290).

Détienne, F. (2001). Software Design–Cognitive Aspects. Springer Science & Business Media.

Edwards, J. S. (2003). Managing software engineers and their knowledge. In Managing software engineering knowledge (pp. 5-27). Springer, Berlin, Heidelberg.

Elliot, A. J., & Church, M. A. (1997). A hierarchical model of approach and avoidance achievement motivation. Journal of personality and social psychology, 72(1), 218.

Fairbanks, G. (2020). Ur-technical debt. IEEE Software, 37(4), 95-98.

Forsgren, N., Storey, M. A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The SPACE of Developer Productivity: There's more to it than you think. *Queue*, 19(1), 20-48.

Freeman, S., Eddy, S. L., McDonough, M., Smith, M. K., Okoroafor, N., Jordt, H., & Wenderoth, M. P. (2014). Active learning increases student performance in science, engineering, and mathematics. Proceedings of the national academy of sciences, 111(23), 8410-8415.

Fritz, T., Murphy, G. C., Murphy-Hill, E., Ou, J., & Hill, E. (2014). Degree-of-knowledge: Modeling a developer's knowledge of code. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(2), 1-42

Gibbs, G. R. (2007). Thematic coding and categorizing. Analyzing qualitative data, 703, 38-56. https://dx.doi.org/10.4135/9781526441867.n4

Greenhalgh, T., Hinton, L., Finlay, T., Macfarlane, A., Fahy, N., Clyde, B., & Chant, A. (2019). Frameworks for supporting patient and public involvement in research: Systematic review and co-design pilot. Health Expectations, 22(4), 785-801.

Harackiewicz, J. M., Barron, K. E., Tauer, J. M., Carter, S. M., & Elliot, A. J. (2000). Short-term and long-term consequences of achievement goals: Predicting interest and performance over time. Journal of educational psychology, 92(2), 316.

Harackiewicz, J. M., Durik, A. M., Barron, K. E., Linnenbrink-Garcia, L., & Tauer, J. M. (2008). The role of achievement goals in the development of interest: Reciprocal relations between achievement goals, interest, and performance. Journal of educational psychology, 100(1), 105.

Hicks, C. M., Pandey, V., Fraser, C. A., & Klemmer, S. (2016, May). Framing feedback: Choosing review environment features that support high quality peer assessment. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (pp. 458-469).

Kalliamvakou, E., Bird, C., Zimmermann, T., Begel, A., DeLine, R., & German, D. M. (2017). What makes a great manager of software engineers?. IEEE Transactions on Software Engineering, 45(1), 87-106.

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. IEEE Software, 29(6), 18-21.

Landauer, T. K. (1988). Research methods in human-computer interaction. In Handbook of human-computer interaction (pp. 905-928). North-Holland.

Madni, A. M., & Jackson, S. (2009). Towards a conceptual framework for resilience engineering. *IEEE Systems Journal*, 3(2), 181-191.

Maguire, M., & Delahunt, B. (2017). Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars. All Ireland Journal of Higher Education, 9(3).

Marsick, V. J., & Watkins, K. E. (2003). Demonstrating the value of an organization's learning culture: the dimensions of the learning organization questionnaire. Advances in Developing Human Resources, 5(2), 132-151. https://doi.org/10.1177/1523422303005002002

Meyer, M., Cimpian, A., & Leslie, S. J. (2015). Women are underrepresented in fields where success is believed to require brilliance. Frontiers in psychology, 6, 235.

Murphy, M. C., Steele, C. M., & Gross, J. J. (2007). Signaling threat: How situational cues affect women in math, science, and engineering settings. Psychological science, 18(10), 879-885.

Premachandra, B., & Lewis Jr, N. A. (2020). Do we report the information that is necessary to give psychology away? A scoping review of the psychological intervention literature 2000–2018. Perspectives on Psychological Science, 1745691620974774. https://doi.org/10.1177/1745691620974774

Prince, M. (2004). Does active learning work? A review of the research. Journal of engineering education, 93(3), 223-231. https://doi.org/10.1002/j.2168-9830.2004.tb00809.x

Quintanilla, V. D., Erman, S., Murphy, M. C., & Walton, G. (2020). Evaluating Productive Mindset Interventions that Promote Excellence on California's Bar Exam.

Rist, R. S. (1991). Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. Human-Computer Interaction, 6(1), 1-46.

Ryan, S., & O'Connor, R. V. (2009). Development of a team measure for tacit knowledge in software development teams. Journal of Systems and Software, 82(2), 229-240. https://doi.org/10.1016/j.jss.2008.05.037

Schunk, D. H. (2012). Learning theories. an educational perspective sixth edition. Pearson.

Singer, Eleanor, and Mick P. Couper. 2017. "Some Methodological Uses of Responses to Open Questions and Other Verbatim Comments in Quantitative Surveys." *Methods, Data, Analyses* 11 (2). https://doi.org/10.12758/MDA.2017.01.

Suter, W. N. 2012. "Chapter 12: Qualitative Data, Analysis, and Design." In *Introduction to Educational Research: A Critical Thinking Approach*, 2nd ed., 342–86. Thousand Oaks, CA: SAGE Publications, Inc. https://doi.org/10.4135/9781483384443.n12.

Vaismoradi, M., Turunen, H., & Bondas, T. (2013). Content analysis and thematic analysis: Implications for conducting a qualitative descriptive study. Nursing & health sciences, 15(3), 398-405.

Veenman, M. V., Van Hout-Wolters, B. H., & Afflerbach, P. (2006). Metacognition and learning: Conceptual and methodological considerations. Metacognition and learning, 1(1), 3-14.

Yang, F. P., Jiau, H. C., & Ssu, K. F. (2014). Beyond plagiarism: an active learning method to analyze causes behind code-similarity. Computers & Education, 70, 161-172.

APPENDIX A: *Initial Interview Script*

Interviews were semi-structured interviews: this means that while the initiating questions were used for all interviewees, participants were allowed to share tangents, observations, and delve into new topics as they arose naturally in the conversation. All interviewees were asked the initiating questions in this script as given below, but were asked follow-up questions and probed to elaborate (e.g., "tell me more about [*what you just said*]"/ "tell me how often you think [*strategy*] works") in ways that were specific to their responses.

[after "debugging" observation]: What did this task make you think about? What do you think was most valuable for your learning?
Probe: any comments that came up during "debugging" observation task

Give me a quick background on how you interact with or write code, in your day-to-day.
How many days out of the week do you write code
Are you collaborating with others
Are you reviewing others' code or managing collaborations

1) Probe: Moments they had "ramped up" or onboarded to an unfamiliar codebase
2) Probe: Moments they began to write their own decision into a collaborative codebase, and moments they solved a "bug" within a collaborative codebase
3) Probe: How code review and any other feedback points were experienced in this problem-solving process

Tell me about how you review collaborative code.
Tell me about how you learn here
Navigate between different files
Prototype and make decisions
Come back to code that you wrote in the past and understand it

Tell me about onboarding into someone else's code, or a codebase that is new to you. How do you learn here?
What does onboarding or ramping-up look like
When do you begin to contribute
How do you ask for help or feedback

Tell me about how you get feedback on code and collaborate on code.
Tell me the general process at your organization for joint projects
Tell me more about code review
Tell me more about pair programming
Are there types of feedback you don't get
Are there moments you wish you could have gotten feedback

Tell me about where you feel like you get credit, or recognition for effort.

Are there things about collaborating on code writing that you wish were easier?

If you can think of one, tell me about a time that it was difficult to ramp somebody up into code that you had written / (if inapplicable) tell me about a time it was difficult for you to ramp up into somebody else's code
Tell me about how you share context that's important for code that you've written
Tell me about how you think other people on your team share context, or decisions about code they've written

Tell me how you share your decisions about code with other people

- *Tell me about how your organization does knowledge sharing*
- *Tell me what you think is important for you to do in your job outside of writing code*
- *Tell me about how often you do documentation/or not*

APPENDIX B: *Selected Respondent Quotes*

Stage 1. Active Learning

“If you’re like ‘how does it work,’ you want to understand it in general, just use the actual app in contexts -- see the solution, play with that thing, as you play with that thing you’ll get more of a sense of why the code is the way it is.”

“Spend 90% of my time talking and communicating and asking questions and 10% of my time writing code, that’s probably right. Asking questions, really listening to a teammate, understanding where he may be coming from. [...] especially as a junior dev, a big pain point is asking questions and having the time to get the people I need in a room to learn the problems.”

“When people make errors [even just] typos that prevent code from running, it’s really important to see those being made. It is important in terms of making mistakes and being able to correct them, which is a large proportion of the learning process [....] The [mentors] I find the most helpful are the ones that have a stronger preference towards live coding, for the reasons I described, for seeing mistakes happen.[....] You often don’t get this.”

“[people tell you] ‘here’s one thing to look for,’ but not necessarily what it looks like when the failure occurs. You’ll hear [senior teammates] saying “oh this typically happens,” but they don’t go into it, so the warning will totally fly over your head because you don’t have a place to store it. [...] Direct interaction with code makes it feel real, tangible, easy to remember.”

“You go through a process of elimination, let’s hold these things constant, and then it becomes a trial and error thing, and that trial and error becomes exhausting....[laughs]. You don’t have a mental model anymore, [you] break the mental model to learn.”

“I’d have this pattern of search where I go down a level and say ‘wait that’s not the right place to go’ and bounce back up; I had to know how to do it wrong in order to do it right.”

Stage 2. Code Review

“Do we get to talk about all that learning that happens before our code? No [laughs], no. I mean I think it would be seen as a waste of time.”

“Code review is trial by fire–it sounds extreme. But you [just] get two million comments back to do it differently. You don’t get to talk about why you did it that way. We’re not necessarily good at the culture. [...] But I’m very much a tinkerer. I need to make mistakes.”

“Inside of [my] team it’s good because we all code review each other’s work and we do it well. We get buy-in before making major changes. But with other teams, other teams end up touching our codebase [...] they bring in models that are comfortable to them, and that becomes mismatch. Decisions get complicated. Our [process for feedback] doesn’t address those model conflicts. Usually the mismatch is better resolved [talking]. So really [laughs] maybe we avoid code review when there are real problems.”

“I am watching in code review, like, ‘ok, that’s what they want me to sound like.’ Of course I learn but they don’t really explain… [it’s] just like being checked up on, so I watch to not get in trouble.”

“People don’t want to talk about your choices. Or what you learned, no. People just joke about how much they don’t want to talk. You don’t want to be annoying, you don’t want to be like ‘that guy.’ Code review isn’t a place to talk about everything around the code, there isn’t a place to talk.”

“[for getting help or feedback] We basically have code review. It’s the only time for feedback. You know, it’s the only time you get to actually talk to somebody. And then it’s not really the reasons you did things. It’s all just, it’s very transactional. Making sure you don’t look like an idiot is important.”

“I guess the thing to think about is that we were always on deadline. We have to do this thing in 48 hours, but those [code review] conversations [...] the scope just expands. Exponentially. It got kind of frustrating. When someone else gets involved [in code review], there’s a burden and pitfalls. [...] So we end up trying to not involve each other and avoid decisions coming up in code review.”

“We review big code changes but…The main people who are talking are typically the most senior…they throw out terms, say high level things. I see these two [junior team members] who don’t say anything but you know when you can look at someone and they’re in silence and they’re not looking engaged? Yeah. I wish it could be like, maybe all of us who are actually figuring it out could get together and talk, but no it’s like, some senior person reviewing and telling them things are wrong. But they don’t really see why you thought you had to do it that way.”

“But if I was new and trying to work out what this [code] did - I think code in general only makes sense once you understand the context of the code - If I was onboarding someone I’d be demoing it. You have to prepare them: there’s a lot of code here and a lot of it’s good and a lot of it’s bad but you want to prepare them. You want to say here are the hot spots where we’re putting effort into, and also the areas that we’re putting effort into might be messy because of that. People get messed up, people get bad feedback, hate reviews because they don’t understand their effort isn’t in the right place. But sometimes I’m like, no one told them.”

“There are situations where folks are modifying code that I’ve written because something else isn’t working, so they’re trying to play around […]. I want to help. I would love to get there earlier in the process but [the way it works now] a month goes by before I see them. Maybe I get pulled into a code review, but that’s not really when I would’ve helped.”

"My background is not traditional for computer science. It is tough. The hardest parts of the workflow…it's all tough. I can be fairly productive once I figure out the codebase, but figuring it out, it takes me a long time to get there and it feels like you're not, you know you're working really hard but no one sees it."

Stage 3. Environment Reinforces Learning Loss

"I guess documentation is the way I'd 'help' other people but they treat it like a time sink. I end up feeling like I have too much to say so I don't say anything? Like other engineers think you're not really supposed to say too much and clutter up anywhere so I think if I figure out something I think is neat like I solve it, it just stays in my head like maybe I write it down for myself. So it doesn't help anyone."

"If you work with a big company they'll expect you to have documentation - [but] that's not the same documentation as the internal engineer-to-engineer documentation that's needed. I waste a lot of time doing the documentation they want to see. I know it's not really teaching anyone."

"Ideally we were supposed to comment code….reality? Less than 10% of our code was actually well commented."

"We have no documentation of our code, it's very primitive. We haven't been doing too many notes in code unless it's a 'to do.' […] People are better at compromising in person, [but] that gets lost. I have had the same conversation so many times."

"I met [the original authors of code I was working on] in the beginning of the project but that was it […] I guess I was supposed to submit a pull request if I wanted to but I thought they would be like, 'who are you' […] so my experience was trying to sift through all this stuff on my own. I didn't want to bother people. It's a lot of stuff like that, like I'm checking for the names of people [in github, in Slack]… [people] I met for one hour in the beginning of the project and I'm seeing they don't actually talk to each other so I shouldn't [reach out to them]. But I don't really want to bother them again. I don't know if I'm going slower than they'd expect. So there's a big difference between what people say and what they want you to do, yeah."

"As a more junior dev […] I want to become more consistent with the person who's more senior to me, I want to figure out why they've developed those practices. But they never have time for us. You are doing this figuring it out from what you can see [in code]"

"We tried [to advocate for more pair programming] and got a lot of pushback."

"The best way for someone to really understand your code is to sit down with them, go through examples [….] looking at this code, trying to figure out at what point was this project talking to what other project […] reaching out to people in Slack […] building an understanding of how these pieces fit together. That's the best. [points emphatically] the best way. That barely ever happens. Because the developers probably aren't there. People don't have time to talk to you. So then you go to documentation systems. [...] It's strange but, documentation can tell you how people don't have time."

"At my old company I was the one who provided the context, I enjoy and have a bit of skill at keeping the whole web in my head and saying oh, that connects here, that connects here. [...] I'm good at that. But I don't get to do it here. I kind of realized the career cost to acting that way. Because we really intensive, time intensive, I never get to document all that knowledge."

"It's a badge of honor to work twenty hours of a day [...]. Go go go. I was the single point of knowledge for context on this code. [...] Because we were really focused on churning and iterating I never got to document it. I know we should've. If we hire more engineers we're not going to be able to speed them up because there's not this repository of knowledge. One person can't keep it all in their head."

"[Onboarding into the collaborative codebase is] pretty draining and frustrating. Kind of full of despair. You have to be very good at documenting your own progress. No one will do it [for you] or help. [...] I like exploration but knowing there's the time pressure, I personally don't like actively asking questions."

"Most of the time, there's no descriptive message any time somebody changes things [in our shared code]. Or the revision is too big. People aren't good at adding reasoning or key insights because it takes time. And no, it isn't valued. So no you don't get all the key insights."

"The easiest, simplest way [to understand someone else's code] is just to talk to them. That barely ever happens because the developers aren't there and people don't have time to talk to you."

"I am always trying to remind myself I love coding. I didn't know it would feel like this to actually have this job. Super alone."

"The problem with us is that we are all working remotely, and we meet only [once in a while]. During those [few] days ...we set all the tasks for the whole week, month. So you're solving something that didn't come up then, you're alone."

"I'm horrible about commenting. Yeah, [even though] comments are really the only way to leave context for yourself in the future. Time sinks aren't a priority. [laughs] I guess learning is a time sink."

"I miss writing code with other people. We are collaborative in other ways but not with code. [I miss] having other people to bounce ideas off of, just being able to build off of others' experiences. I miss it a lot."

“[when I was a student] every day I was pair programming. It was hard [...] but you would literally learn from other people’s experiences, it’s very productive. Or you’d get to really learn something from explaining to others. No pair programming in the real world. No explaining.”

“We tend to believe code should be self-explanatory, everything that’s not self-explanatory, you add a comment. Having to comment something is a red flag, an indication of messing up.”

“And right now [if] you look at where I left off [writing code], it does need a bunch more documentation that’s giving me a sense of that’s the rationale and why I built it. [...] I can tell you if I was onboarding someone [into code I wrote], I would try to find the next code task and kind of walk them through it. Well that’s ideal. And it’s not there. No one cares if you do it so you know you just, you don’t benefit.”

“I think generally the conflict is [...] a lot of the work we need to be doing is behind the scenes, needing to make things clearer [...] that can be kind of difficult because we have to do that, but we’re [told to prioritize] things we can show physically. Things need to count from an outsider’s perspective.”

“We have pretty good team dynamics. Our project manager’s like, even if [working on code understanding is] not something shiny, in the long-run that will pay off, so do it.”

“Even writing a comment is context switching for me, takes me out of the flow. But then you’re on the other side and it’s all gold, It’s gold because -- all this stuff, all the comments and the people helping, it’s all to help you understand the code. If you can figure out how to traverse [comments and documentation] really well, that can help you in a big way. That’s really hard, it takes time, it takes discipline.”

“If I were just teaching someone? Showing people multiple examples [of why code works] – that would be great. I would take that chance if I were teaching but I can’t here. If I’m just getting someone up to speed at work though, I can’t. [Onboarding someone to collaborative code] fails all the time...someone just shows you a piece of code and then you try it on your own and it’s totally mysterious.”

“From my standpoint I want more access to pair programming. Our company only uses it if you’re blocked. Feels like you have to get into trouble to talk to people. But I’m actually really good at helping people when we pair program.”

“The hardest part [of solving this bug] was the lack of documentation....then it’s like, you know, I get the sense no one cares. Even if the framework is beautiful, no one cares about *me now*, trying to do stuff with it.”

APPENDIX C: *Trust Building & psychological safety after the Active Work Session*

An unexpected outcome of the active work sessions was that participants began to use noticeably more casual, familiar, and warm language after the “bug solving” task. While this is not unexpected (as you spend time with participants and develop a conversation, the conversation often becomes more comfortable), I believe it’s worth calling out because it highlighted a specific theme of *trust*, and how trusting conversations elicit a different type of disclosure. Therefore, in a follow-up qualitative analysis, I examined the immediate five minutes after the active work session.

In ten of the interviews, participants initially referred to themselves by their official role description and title. After the bug session, nine of these participants began to describe themselves with different, more idiosyncratic language. Many of the participants began our interviews discussing explicit feedback structures and used language of “should” and responsibility (e.g., “for code review <u>you need to make sure</u> you have learned the style”, “we’re <u>supposed to</u> make sure we comment”...). After the active work sessions, in which the participants were able to narrate their live problem-solving, much more humor emerged. Participants spoke more openly about “what works” and “what doesn’t work” and their own learning constraints and desires while writing code (e.g., “even the best people can only keep so much in their head,” “Ninety percent of what I realized I want to discuss is really more simple stuff than all the stuff we have meetings about”).

This emergent experience prompted me to think about how *interactions between methods can strengthen insight*. I believe that our interviews on learning in code writing were more authentic, engaged, and felt safer, because I had held space with my participants in the debugging sessions without attempting to provide answers or judge the quality of their code or their problem-solving.