A Cumulative Culture Theory for Developer Problem-Solving

CATHERINE M. HICKS, Developer Success Lab

ANA HEVESI, Uploop

"If, however, one avoids the linear, progressive, Time's- (killing)-arrow mode of the Techno-Heroic, and redefines technology and science as primarily cultural carrier bag rather than weapon of domination…"

- Ursula Le Guin, The Carrier Bag Theory of Fiction

1 INTRODUCTION

Software developers work together to produce innovation in software that achieves extraordinary goals such as coordinating air travel, running banking systems, capturing and processing unprecedented amounts of data from scientific and medical instruments, and maintaining global communications. As an integral part of this work, developers invent, modify and share technological solutions to suit their problem-solving purposes. Arguably, the diversity and uniqueness of human innovation defines humanity's success on the planet, and our collective tool use is an exemplary way that innovation is expressed (Carr et al., 2016; Dean et al., 2014; Rawlings & Legare, 2021; Vaesen, 2012).

Understanding how developers problem-solve within ecosystems of practice, tooling, and social contexts is a critical step in determining which factors dampen, aid or accelerate software innovation. However, industry conceptions of developer problem-solving often focus on overly simplistic measures of output, over-extrapolate from small case studies, rely on conventional definitions of "programming" and short-term definitions of performance, fail to integrate the new economic features of the open collaborative innovation that marks software progress, and fail to integrate rich bodies of evidence about problem-solving from the social sciences (Baldwin & Von Hippel, 2011; Nichols, 2019; Piva et al., 2012; Sadowski & Zimmerman, 2019; Shaw, 2022; Hicks, 2023; Hicks, Lee & Ramsey, 2024, Lee & Hicks, 2024).

We propose an alternative to individualistic explanations for developer problem-solving: a ***Cumulative Culture*** theory for developer problem-solving. This paper aims to provide an interdisciplinary introduction to underappreciated elements of developers' communal, social cognition which are required for software development creativity and problem-solving, either empowering or constraining the solutions that developers access and implement.

To inform this introduction, we drew from our diverse experiences as a psychological scientist who has conducted large-scale empirical research with software teams and in many achievement contexts (e.g., leading developer-focused studies on technology learning cultures), and a developer experience practitioner who has solved many applied challenges while scaling real-world developer communities (e.g., leading developer-focused work in the early Node.js ecosystem and at organizations such as Stack Overflow). We synthesize evidence across modern cognitive science and psychology along with case studies from communities of practice where we have personal lived experience with

Authors' addresses: Catherine M. Hicks, Developer Success Lab, http://drcathicks.com/; Ana Hevesi, Uploop, https://www.uploop.dev/.

the impacts of social communities on developers. We propose that despite a conventional emphasis on individualistic explanations, developers' problem-solving (and hence, many of the central innovation cycles in software) is better described as a cumulative culture where collective social learning (rather than solitary and isolated genius) plays a key role in the transmission of solutions, the scaffolding of individual productivity, and the overall velocity of innovation.

In the following sections, first we address some common misconceptions about the underlying drivers of complex knowledge work, exploring associated opportunities to learn from underrecruited empirical findings on social learning. In this we theorize about how the many audiences interested in understanding software developers (practitioners, researchers, and organizations invested in producing technological solutions within software ecosystems) can move toward a broader theoretical model of developer problem-solving, presenting a cumulative culture framework to describe causal cycles in innovation and technological solution-crafting. Finally, we conclude with several case studies within software engineering–Node.js, Rust language learning, and Stack Overflow community dynamics–and consider how a cumulative culture lens helps to describe developer experiences within them.

1.1 Towards A Broader Science for Developer Problem-Solving

Imagine that a software developer at a midsize organization begins work on a new technological problem identified as part of feature work designated a priority by her cross-functional team. Her initial materials included input from product research, advice from her manager about a few possible solutions she might explore, assignment to a specific part of the problem, and an example use case, along with her own skillsets developed over the last several years of learning a new programming language. She is new to the company, and therefore new to the codebase, and so devotes some time to understanding and testing how her area of work might relate to existing technologies. She also realizes that the use case has been defined very generally, but that there are specific decisions which need clarification, and reaches out to a member of the product team for that clarification. While beginning to draft a solution, she identifies an unexpected gap in her knowledge about a framework which is being used in the codebase. By the end of the day, she reaches out to a senior member of her team to share her early idea for how to solve the problem, and is excited to learn that her team has a tool they use to implement a particular part of the solution which will help her work move more swiftly. They make plans to meet the next day, and conscious of the need to maintain her thinking, she documents a few notes about her in-progress work to look at the next morning.

1.1.1 Cognitive Mechanisms & Developer Problem-Solving. During even one workday, progressing in software development requires the completion of many complex cognitive tasks. As such, we theorize that developer problem-solving likely recruits core cognitive mechanisms such as the **executive functions** (e.g., inhibition, working memory, attentional control, cognitive flexibility), alongside **creativity**, **planning**, **causal reasoning** and **social learning**. Human problem-solving is multifaceted, and this is therefore not an exhaustive list, but a fruitful starting point to consider cognitive mechanisms which play a role in technological output. For example, these cognitive processes working in concert have been described as foundational to how humanity creates new solutions while using tools, and these processes have also been empirically measured across diverse cross-cultural studies investigating how people develop the capacity to generate novel solutions using tools (Rawlings & Legare, 2021; see Rawlings, 2022 for a summary of this literature). Software development problem-solving is often assumed by naive observers to happen primarily

during the construction of novel lines of code (Shaw, 2022). In our view, software development problem-solving is required just as robustly by many non-code-writing tasks such as making decisions about third-party resources, knitting together complex infrastructures, convincing and persuading others' solutions to match one's own, hunting for causal explanations of observed events, and engaging with the norms of software development groups.

Rather than searching the core cognitive mechanisms for a silver bullet, or positioning one mechanism as "the driver of developer productivity," these diverse capabilities should be understood as operating in concert during complex problem-solving. Together, they are called upon to support different needs throughout a longitudinal process (Rawlings & Legare, 2021). For instance, a developer needs to inhibit their focus on a particular solution and perform task-switching when they recognize an unexpected barrier, but also must maintain attentional control and ignore irrelevant distractors in order to efficiently work out a full solution. A developer needs cognitive flexibility in order to introduce novel and surprising solutions to their broader team, but must be capable of committing to a shared plan and reproducing agreed-upon techniques. A developer in one role may often recruit creativity and divergent thinking, contributing many novel solutions, but a developer in another role may find it more appropriate to spend the majority of their days navigating previously-written code and developing a fine-grained understanding of maintenance tasks, aided by long-term memory, reflection and retrieval. A developer may need to rely on their working memory to efficiently navigate through a codebase and to hold task rules in place while considering modifications of a solution, but also needs to engage in longer-term cycles of learning, spaced retrieval and iteration in order to build new skills (Bjork et al., 2013). The diversity of cognitive contributions to problem-solving is validated not only in the scientific literature, but by accounts from practitioners. Developers widely report valuing a diverse range of tasks, and see many forms of problem-solving as essential to technological progress (Masood et al., 2022; Hicks, 2023).

The software industry generally recognizes that relying on unnecessarily narrow task definitions and their metrics (e.g., a count of lines of code, the rate of commits within a selected window of time) to measure developer productivity may operationalize a proxy measure for some types of *programming* performance, but inevitably fails to encompass the full range of solution-crafting required for *software development* (Sadowski & Zimmerman, 2009; Winters et al., 2020; Hicks, Lee & Ramsey, 2024). Similarly, we argue it is critical to take a holistic view of the cognitive and psychological underpinnings of developer problem-solving. Proposing unnecessarily narrow cognitive task-based origins of developer ability, as when claiming that a specific cognitive mechanism is the cognitive "driver" of developer productivity, does not reflect the reality of complex work (a paraphrase of real examples we have heard repeated by industry practitioners include: "the best developers must have an extreme ability to attain attentional flow state," "engineering brains have faster processing speed," or "cognitive load is the reason teams struggle"). Proposing that a single factor in cognition is solely responsible for societal innovation is similar to proposing that the only meaningful information about a piece of software is how many lines of code it has. Prioritizing certain psychological and cognitive processes can be both adaptive and nonadaptive depending on situated goals and context.

As a second, graver caution, it is critical to note that many generalizations about the cognitive underpinnings of complex work obscure the fact that there is adaptive flexibility in how different people might use diverse abilities to accomplish the same tasks and achieve the same level of collective success. Both cultural and individual context can change how the executive functions are measured and whether people's abilities are accurately understood. Modern cognitive science continues to reconstitute its approach to cognitive mechanisms in light of new, more ecologically valid interpretations of these effects (see Munakata & Michaelson, 2021, Yanaoka et al., 2022 and Gaskins & Alcalá, 2023).

People with different histories necessarily have recruited their cognitive abilities to solve diverse and different problems. Conventional cognitive tests (and therefore evidence about cognition) have favored privileged test takers; emerging recent research suggests that the conventional interpretation of group differences in cognitive task performance has overlooked a strengths-based interpretation of difference. For instance, lower social class individuals perform *better* than higher social class individuals at social cognition tasks (Dietze & Knowles, 2021). This has led researchers studying executive functions to argue that differences previously interpreted as a deficit may instead represent divergent strengths, e.g. arguments for *domain-specific cognition* (Mittal et al., 2015; Fendinger et al., 2023). Supporting this, changing the ecological validity of a cognitive task has been shown to mitigate, or even reverse the direction of, supposed group differences in mechanisms such as working memory compared between adversity-exposed youth and their counterparts (Young et al., 2022). Similarly, better relational memory has been found among adversity-exposed preschoolers than non-adversity-exposed comparisons (Rifkin-Graboi et al., 2021), and high executive functioning has been documented among children in a Yucatec Maya community which is not measured accurately by traditional executive functioning measures (Gaskins & Alcalá, 2023).

Echoing researchers who have been concerned with more accurate and equitable understandings of achievement and innovation across the many fields that study human behavior (e.g. Boykin, 2023), we therefore argue that any actionable, pragmatic and evidence-based model of developer problem-solving should take a holistic, human-centered, strengths-based approach to cognition, start from a place of social context, be mindful of the historical errors associated with our models of human abilities and their measures, and integrate important modern evidence on how real-world adaptive and flexible problem-solving recruits across a diverse suite of abilities to meet the challenges of our lived experience.

1.1.2 How has cognition been considered as a driver of developer problem-solving? We are often primarily interested in improving the outcomes of groups in software work, not merely individuals. However in our experience, isolated, within-the-individual executive functions have typically been foregrounded in industry conversations about cognition in coding. Psychological theories and empirical evidence about broader systems like planning, creativity, and social learning are invoked less frequently as potential explanatory factors for desired technology outcomes. While many researchers have called for the integration of psychological evidence into software research, advancement in this interdisciplinary work is still nascent and called for by researchers increasingly concerned with the role of sociocognitive affordances in technology development (for further commentary, see Hicks, 2024).

In some ways, foregrounding individual executive functions is an understandable starting point for the applied science of developer problem-solving as it seems to substantially reduce the search space for "key factors" that can be used as intervention points to improve developer problem-solving. Toward this aim, some suggested interventions for improving developers' work have centered on aiding an individual's executive functions, and then evaluating task performance in terms of increases to individual output. When venturing into the cognitive science of developer workflows, software research has frequently characterized perceived friction in problem-solving through the lens of Cognitive Load Theory ("CLT": see Gonçalves et al., 2019 for an overview; see Gonçalves et al., 2022 for an empirical example). This approach has yielded tactical recommendations for interventions that could be included in the design of developer tools: Gonçalves et al. (2022) tested how scaffolding task performance for software developers via a structured checklist might aid with a code review task, and did find that this approach improved developers' efficiency. Despite

this beneficial example, we argue that a reductionistic measure of executive functioning has many flaws as a starting place for developer science.

It is important to note the limitations that come from pursuing not simply a tactical and situated intervention, but an entire *model* of problem-solving that posits a sufficient explanation of developer output is provided by a sparsely measured singular cognitive mechanism (e.g., working memory capacity). Many critiques of CLT have noted the risk of overextending small effects in the realm of education to serve as explanatory factors for performance: interventions attempting to reduce cognitive load, and thereby improve performance, have mixed success, are often specific to classroom learning design rather than expert performance at work, and direct impacts from cognitive load on performance can be highly variable in measurement or impossible to directly observe, forcing work to rely on perceptual measures (Ayres & Paas, 2012; Kirschner et al., 2011; Matthews et al., 2018). Software research has likewise reported mixed results, even in small constrained laboratory tasks, in whether cognitive task performance associates with programming tasks (Piantadosi et al., 2023). Overall, executive function training frequently fails to yield substantive gains in learning or achievement, leading researchers to acknowledge that such models of problem-solving may rely too heavily on abstract laboratory-based tasks (Niebaum & Munakata, 2023).

Outside of software research, psychologists' critiques of cognitive load interventions include that there are significant unanswered conceptual questions around the fixedness or malleability of an individual's capacity for intrinsic load, the difficulty of interpreting mixed empirical findings and nonstandardized measurement methods, and the difficulty that individuals have in making subjective reports about the *types* of cognitive load they are experiencing (Schnotz & Kürschner, 2007). Further, it is robustly documented across the psychological sciences that a substantial amount of variance in people's real-world complex skill acquisition and achievement is predicted not only by cognitive factors but by noncognitive factors, suggesting these characteristics frequently present large effects and important targets for improving problem-solving outcomes at the organizational level (see Ackerman et al., 1995; Stajkovic & Luthans, 1998; Kurtessis et al., 2007 for representative examples).

In the above empirical study of code review checklists, even while reporting efficiency gains, Gonçalves et al. (2022) also found that *higher* cognitive load led to better *performance* on a code review task, leading them to distinguish between efficiency and performance goals and comment on the complexity of understanding developer cognition. We are grateful to these researchers for this empirical work, and share this example not as a criticism of this particular study; instead we highlight this as a particularly thoughtful, productive example from software research which uses a cognitive theory as the basis to explore the evidence for and against a pragmatic intervention intended to help developers.

However, in our view, software industry commentary concerned with intervening on "developer productivity" has in many cases taken empirical evidence in such studies as a basis for an overextension of this singular mechanism to propose it as a graspable cause for improvements in software developers' high quality problem-solving. Given this, such commentary proposes reducing cognitive load as a highly generalizable, population-level lever for improving problem-solving, based on assumptions about working memory's contributions to performance over time which might also broadly reference many complex, multifaceted factors such as stress, absence of social support, or even other specific cognitive factors mentioned at the beginning of this section such as attentional focus, inhibition, or memory retrieval (for instance, cognitive load is frequently informally defined as effort in general, such as *"the amount of*

intellectual effort required to execute a task" in Murabito, 2023). Branding cognitive load as "effort" of any and every kind is an argument which could causally link nearly any high-level positive outcome to a reduction in cognitive load.

Even granting this high level of operationalization, the relationship between performance and effort is widely acknowledged by learning science as deeply complex depending on task context, a relationship which performers frequently misperceive due to persistent systematic biases about what effort leads to long-term learning and performance (see Bjork, 2013). Even should we grant their existence, individual effects do not readily flow to group-level recommendations when benefits compete. Some positive changes could just as well necessitate an *increase* in effort, and decreasing one individual's "load" may often require an *increase* in another's during task performance (e.g., *"Adding comments is also a good way to reduce cognitive load,"* in Franco, 2024). Even within education, researchers' investigations into the complexity of learner goals and task contexts suggest that it may well be impossible to make a generalized statement about whether pursuing a reduction or increase in cognitive load overall is a desirable aim outside of a situated context, and that "types" of cognitive load in complex learning are not yet well understood (Kalyuga & Singh, 2016). As stated by one of the originators of CLT, Sweller et al. 2019: *"If emotions, stress and uncertainty are seen as undesirable states for learning, one might say that they cause extraneous load that should be decreased by preventing these states. But if emotion, stress and uncertainty are seen as an integral element of the task that must be learned [...] they contribute to intrinsic cognitive load and must be dealt with in another way. Then, future research should contribute to identifying instructional interventions that help learners deal more effectively with stress, emotions and uncertainty."*

A reductionist, isolated, and individual-without-groups model of a cognitive "driver" for developer problem-solving is troubling not only because the selection of a single cognitive mechanism is not justified by the executive functions literature as a sufficient causal explanation, but also because it readily leads to the conclusion that we should exclude large groups of individuals from software development based on their perceived deficits in a capacity like working memory. This example of the impact of theory is not itself a theoretical one; groups whose potential and abilities have been systematically mismeasured by insufficient, and in many cases biased, models of ability have long been excluded from specific career pathways, including in Psychology and STEM fields, in some cases an act of exclusion that has used measurements of "working memory" constructs specifically (see Walton & Spencer, 2009; Boykin, 2023).

Similar arguments could be made, and counterarguments waged, when reifying any other solitary cognitive mechanism as the seat of problem-solving. Given all of above, it is our argument that constraining our explanations to a specific measure of one of the executive functions, and proposing this as a sufficient explanation for how developers produce technological innovation, is both an unsound translation of the cognitive science of problem-solving, and unduly centers an *individual ability* explanation of developer work, e.g. "a developer creates better or worse technological solutions depending on the capacity of her working memory." This elides the social-innovation exchanges occurring within technological communities of practice, the complex concert of the executive functions, the role of social cognition, and how all of these contribute to field-level problem-solving. An analogy can be drawn from education, the original context for cognitive load theory: while reducing non-relevant cognitive load has been acknowledged as a beneficial lens to occasionally consider when redesigning pedagogy, no substantive learning science theory proposes that an individual student's potential to perform a complex adult job, help others achieve, and live a meaningful life in society, is either explained or predictable solely with a measure of their likelihood of being susceptible to cognitive load, nor that new interventions in education can be successfully evaluated only in terms of their reduction of cognitive load.

We argue, therefore, that both the software industry and research about the people in it need to draw upon a broader, robust set of theories to develop evidence about the complex, multifaceted nature of developer problem-solving. In the following section, we introduce a few key research areas that we believe can help broaden conceptions of developer problem-solving explanatory models. We focus in particular on social learning, creativity, and causal reasoning, and how they function together under a *cumulative culture* of shared problem-solving and innovation across developer communities.

2 A CUMULATIVE CULTURE & DEVELOPER PROBLEM-SOLVING

A developer tasked with crafting a technological solution must first recognize both the initial problem and the goal of a solution, then gain an accurate understanding of their technological opportunities and constraints, and finally develop and assess that a solution achieves the goal, potentially engaging in many steps of iteration and refinement. Along the way, a developer might simultaneously consider how their actions help them to meet other complex social-psychological needs such as protecting their status, belonging and identity as a technical person (Brockner & Weisenfeld, 2016; Brockner & Sherman, 2019; Hicks et al., 2024; Hicks, 2024), weighing interdependencies of solutions and impacts on future solutions (Avgeriou et al., 2023; Ernst et al., 2015), and assessing how best to transmit the details of their solution well enough for it to be accepted (Bachelli & Bird, 2013; Ford et al., 2019). However, outside of extreme cases developers also do not problem-solve in the absence of any cultural history of innovation. Even while sitting at a computer and opening a blank screen to write code, developers bring with them their education, current discussions of craft, organizational norms, patterns taught by mentors or peers, and various technology abstractions and automations, all of which can have a bolstering or dampening impact on the moment of problem-solving.

Developers' solution crafting can be considered either (1) innovation (developing new solutions) or (2) implementation (deploying others' solutions into a new context)–many complex, multi-part solutions require developers to do both. To accomplish the first, developers navigating through technological possibilities and designing unforeseen technological solutions may be thought of as performing *tool innovation*, which Rawlings and Legare (2021) define as "*[d]esigning new tools, or using old tools in novel ways, to solve new problems.*" To accomplish the second, developers focus more heavily on solution implementation, or recognizing, evaluating and reusing already known solutions. While the first category may capitalize on cognitive flexibility and creativity, the second category may capitalize more on our cognitive capacity for imitating and reproducing others' solutions (Tomasello et al., 2015).

Social learning is a defining characteristic of how human cultures pass on and take advantage of moments of innovation, and has been proposed as a central aspect of our species-wide shared libraries of problem-solving (Herrmann, 2007; Elias, 2012). Social learning occurs when learning is obtained via social observations and interactions (Vygotsky, 1978; Heyes, 1994; Legare, 2017). While novel solutions may be generated by individuals, social learning and imitation of those innovations keeps solutions alive. Many species display aspects of social learning, changing their behavior in response to social information, but the transmission of complex social learning marks human society and has been termed ***Cumulative Culture*** (Legare, 2017). Cumulative culture includes both the teaching which ensures that knowledge is transmitted, and the imitation that ensures such knowledge is received.

Social learning is a core cognitive mechanism for human beings attempting to navigate a complex world; that mechanism is thought to drive a collective, communal intelligence which helps to scaffold individual problem-solving. One example of social learning's centrality in human cognition is that novel tool innovation from individuals develops more gradually than imitation across individuals. Across cultures, developmental psychology research has documented that young children are notably better at imitating solutions than creating them, and that complex novel tool innovation develops more slowly than imitative learning (Rawlings, 2022). Throughout our cognitive development, the prioritization of imitation and social learning has been theorized as an adaptive focus that helps us confront the fact that *"novel objects, in the form of artifacts and tools, saturate our world, and we must understand and use an enormous array of them from a very early age"* (Carr et al., 2016). At a species level, rather than pinning our survival on our own solitary innovation, we rely on the innovation of those who came before us.

Informed by this empirical evidence across cultures, we argue for a recontextualization of developer problem-solving. Developers solve problems not only with the power of their individual executive functions, but by taking advantage of this broader capacity for cumulative culture and social learning. This perspective helps us to move beyond an overly individualistic account of developer problem-solving, in which solitary programmers reason from first principles to generate solutions relying only on e.g. their individual working memory, attentional capacity, and individual creativity. Individual cognition matters, but by interacting with complex software development ecosystems and intertwined social contexts, individual developers also begin problem-solving in dialogue with a vast library of others' solutions, and modify and experiment with others' problems and solutions. Improvements can then feed back into collective knowledge, lifting the innovation capital of the entire group. This scaffolding of solutions is key to technological progress (Dean et al., 2014) and the "free revealing" of new solutions marks economies driven by open collaborative innovation (Baldwin & Von Hippel, 2011). One hallmark of cumulative culture is that when products and practices are repeatedly modified, this testing and transmission increases the efficiency, efficacy or complexity of the solutions available in a culture (Dean et al., 2014). This has been described as a cultural "ratcheting" (Tennie et al., 2009). Technological innovations are particularly salient examples of this collective modification.

The experience of "cultural ratcheting" is likely familiar to any software practitioner. When a software developer starts modifying a section of a complex codebase, they face the difficult task of acclimatizing to context, but their causal reasoning is also buoyed by the examples of problem-solving available in chains of action created by others. When a new technology is successfully taken up by software practitioners, its rapid adoption can drive innovation cycles in diverse, unexpected areas. Ideally, such ratcheting transmits across communities of practice, frequently transcending the boundaries of organizations, and shifting a technological field's entire suite of problem-solving approaches. As described by Shaw (2022), software development *ecosystems* move beyond code and its specific solutions to incorporate many resources, which rely on open resource coalitions: "*These open resource coalitions transcend the classical closed software system concepts that a software system has boundaries, that it remains under the developer's control, and that it is dominated by lines of code written for the purpose.*" A software cumulative culture can be seen as the social-psychological process by which software developers transcend the *software system* to navigate on the level of the *software development ecosystem.*

Social learning also becomes a powerful mechanism to consider when we take a life-span view of developer problem-solving. Social learning creates more efficient pathways not just within existing technologies, but for the next generation of technical solutions passed between individual careers. For example, a junior developer gains exposure to "future

thinking" by examining the choices made by past developers, more rapidly and efficiently gaining a vocabulary for such scenarios than they would have come up with on their own. Future thinking allows us to reason about chains of cause and effect and perform "mental time travel" (Vale et al., 2012), and cumulative culture provides a developer with access to the latent solution space of previous groups (Tennie et al., 2009). Stated in the other direction, cumulative culture and its sharing allows past developers to "stash" potential solutions that are not yet empowered by the current reality, passing their intellectual capital and not-yet-deployed solutions forward. Examples of this abound in scientific progress. One such example of cultural ratcheting can be seen in the long history of mRNA experiments and eventual, globally impactful creation of mRNA vaccines (Dolgin, 2021). Our ability to imitate and recontextualize others' solutions and apply them to new contexts unimagined by those others comes so naturally to us, it can be easy to take this cycle for granted. However, on the level of groups, cultures and civilizations, successfully engaging in this process is a driving force for innovation.

We hypothesize that understanding how *collective solutions* are shared between developers, and keeping such transmission functional, may prove a more powerful explanatory factor in technology innovation than any individual cognition or performance variance among developers. It is also likely more *accessible* to those who wish to intervene on and increase technical innovation. While interventions attempting to improve working memory have often proven impractical, unethical, or unlikely to succeed, interventions attempting to improve cycles of adaptive response between individual-and-social systems, incorporating improvements to the psychological context that promotes social sharing and teaching, have emerged as intervention targets which associate with lasting objective outcomes in achievement and performance (see Walton, 2014; Hicks, 2024; Lichand et al., 2024).

Another important aspect of cumulative culture is that it helps individuals both learn from and imitate processes of creativity. For instance, developers' participation in interest-driven technical communities which share niche, unusual, or obscure solutions may bolster developer problem-solving along with well-documented social benefits from support, role modeling, and sense of belonging (Townley, 2020; Trinkenreich et al., 2023). Alongside these social benefits, exposure to infrequently-noticed features of a problem can cognitively benefit developer problem-solving because this helps individuals to overcome functional fixedness, i.e., the cognitive bias which discourages people from seeing novel solutions (McCaffrey, 2012).

Divergent thinking, or generating multiple, diverse solutions, is frequently the focus of investigations into factors that drive creativity and innovation (de Vries & Lubart, 2019). However, convergent thinking is also important in a cumulative culture, and may be equally important for technological progress across codebases, particularly within developers' cumulative cultures. When multiple developers converge on a single solution, integrating many iterations, this can become a key element of software progress particularly as solutions need to meet increasing goals of sustainability, maintainability, or resilience in new contexts (Russo & Ciancarini, 2017). For example, developers rapidly interacting with many parts of a novel codebase may need to swiftly identify a solution that works not only for them, but for others who interact with their code later. Responding to unexpected security incidents that impact or disable global software is another salient example of a problem that requires developers to congregate around a highly portable, generalizable solution. Convergent thinking may also be a central skill for early career developers who seek to rapidly onboard into the collective solution space of their colleagues and the communities of practitioners around the technologies they use. Convergent thinking has been less frequently studied compared to divergent thinking, but there is emerging evidence

that it is a distinct from divergent thinking in both the developmental trajectory (Eon Duval, Frick & Denervaud, 2023) and between individuals and cultures (de Vries & Lubart, 2019).

Returning to the example of a code review process, a lens of social learning also broadens our characterization of the problem-solving that occurs in a code review dialogue between two developers. A developer considering the codebase around her accesses not only her working memory, but the innovations developed by many others to craft more efficient solutions. In a healthy review process she absorbs new information from a reviewer that accelerates her own problem-solving; in an unhealthy review process, she may absorb negative messages that impede directly on her ability to solve problems (Egelman et al., 2020; Lee & Hicks, 2024). Across tooling, a social learning lens can also help to unpack the disparities that emerge when we consider the impact of developer tools. An engineering organization with a robust culture of information flow and sharing may persist to an unexpected degree over friction-filled tool interfaces by easily porting solutions to newcomers (Hicks, 2024).

Beyond team processes and interpersonal interactions, the ecosystem of developer tooling is a particularly key location where developers may feel empowered or constrained to experiment with novel solutions, and social learning is a particularly notable feature of the way that developers interact with their tools. Collective communal learning can help developers scaffold problem-solving in the arena of new and emerging technical capacities, and shared creativity can help developers move past functional fixedness. To foster cumulative culture, we hypothesize that software organizations need to encourage both imitative and innovative work for developers: both "imitative" on-ramps for learners, but also enough flexibility that new innovations are encouraged and shared enough to catch on with others, such that unusual and unexpected technological advances are not gate-kept out of the larger community. Being able to share and vet new innovations back with the community, which then determines their uptake or not, is a key piece of cultural intelligence and is central to a successful economy of collaborative innovation (Baldwin & Von Hippel, 2011). As Rawlings and Legare write: *"our capacity for cumulative culture has yielded complexity far beyond the capabilities of solitary brainpower"* (2021).

The cumulative culture lens may be a particularly fruitful way to approach studying the dynamics of productive change which software practitioners have told us are "strongly felt and known," but unmeasured or implicit in how conventional models describe software development problem-solving. For instance, cumulative cultures of innovation and imitation may help to explain the lifecycle of open source in software. Contractions and expansions in developers' ability to share, innovate and flexibly adapt solutions via social learning may be well-suited to help explain why problem-solving decreases in some areas of software development, and accelerates in others. By moving beyond individual ability accounts of "drivers of developer productivity" to social accounts of developer *problem-solving*, our models may better reflect the dynamic and shared nature of software innovation.

3 BUILDING FOR COLLECTIVE INNOVATION

Using the lens of cumulative culture, in this section we share a few examples of innovation in developer communities.

3.1 Node.js and The Power of Imitation

Node.js, first released in 2009, is a JavaScript runtime which paired web server performance improvements with programming syntax many developers already knew well. It was designed to support work with event-driven HTTP servers which had a greater number of concurrent connections than earlier web server technologies, and a programming model where individual applications could no longer block entire processes. This made it possible to serve a greater number of users at once.

These improvements to performance made Node a popular choice for organizations like LinkedIn (Prasad, Norton, & Coatta, 2014), Netflix (Trott & Xiao, 2015), and Walmart (O'Dell, 2012). However, we suggest Node's proliferation among individual developers was owed in significant part to Node allowing them to write JavaScript—a language already known to many programmers tasked with scripting behavior in web browsers—to accomplish server-side tasks.

Here we observe the imitation side of social learning. By using the syntax of a well-known programming language from client-side development, Node scaffolded the capabilities of many technologists who had not previously worked with web servers. They were instantly granted new capabilities, without learning a whole new language.

Node enabled practitioners who were previously limited to client-side contributions to become "full stack". Rather than a temporary phase, this blurring of lines between disciplines and areas of technical contribution continues today, and may be feeding into a redefinition of what it means to be a developer all together (Shaw, 2022). As described by Holterhoff (2024): *"Like any shift the transition to elevate the frontend has been uneven, but in the particular case of software development it has also been accompanied by a redistribution of roles and a blurring of the lines that traditionally separated front from back end, client from server, and static from interactive…Because the terms frontend, backend, and full-stack don't offer an accurate reflection of the SDLC today they will eventually be replaced by new developer types."*

Since imitation is what ensures innovation propagates among a group, it stands to reason that leveraging imitation in the design of developer-facing technologies more broadly is a powerful strategy for creating long-term changes to the way people work.

3.2 Encoding Intergenerational Learning Into Rust

Rust, a programming language which emphasizes performance and memory safety, has seen significant adoption by individual developers and large organizations. The community stewarding Rust has worked to balance accessibility with security, making the language more available to a wide array of practitioners.

The flagship features of the language make it safer for newcomers to contribute to codebases written in Rust. A unique approach to memory management, known as ownership, allows developers to allocate memory at compile time, reducing runtime errors. This increases security by preventing malicious access to program memory. Rust's type system means that developers are required to predefine the type of data for every variable and value in a program, such as strings, booleans, or integers, and the operations which subsequently may be performed on them. This both reveals the intention one programmer had to their peer, as well as encodes intended behavior. This can be viewed as a primary example of future thinking.

"The Rust compiler's checks ensure stability through feature additions and refactoring. This is in contrast to the brittle legacy code in languages without these checks, which developers are often afraid to modify." (https://doc.rust-lang.org/book/ch00-00-introduction.html)

Rust's evolution and design can be seen in contrast to the older languages of C and C++, which do not enforce memory safety automatically, and are such a frequent source of security vulnerabilities that the US Cybersecurity and Infrastructure Security Agency advises against their use (CISA, 2024). In this light, and perhaps informed by painful lessons learned firsthand, the experienced developers stewarding Rust had a vested interest in attracting newcomers.

Rust demands a lot of new learners (Fulton et al., 2021). Requiring developers to pre-designate so many aspects of their program's behavior upfront necessitates relatively deep working knowledge of foundational principles. The language's adoption despite this can be attributed to an active effort to meet new folks where they are, and help them move forward from there. When a developer begins their journey into Rust, they are offered a wide array of potential learning paths, built and curated by experienced practitioners, and tailored to different learning styles. For example, as of this writing, the "Learn" tab of Rust's website (Rust Team, n.d.) offers links to:

(1) The Book, which is an in-depth dive into language features and behavior
(2) Rustlings, a course which takes place in the command line, and has learners progress by resolving–and thereby familiarizing themselves with–compiler errors
(3) Rust By Example, a curated set of examples and challenges selected to introduce concepts and standard libraries

By encoding the principles of memory safety, accessibility, and future thinking into language behavior and learning materials, and making the barriers between them permeable, Rust has created a vehicle for many types of practitioners to make their way into previously obscured areas of technology work. At its best, Rust is creating a new generation of contributors to areas like game development, embedded systems, and command-line tooling, while reducing adverse consequences for seasoned veterans who maintain these systems.

We do not propose to know the ideal programming language for every situation, an unanswerable question! Instead we offer Rust as an example of how a specific community tackled its adoption challenges. If you want people to do hard things, you make it easier for them to do hard things. Learning culture is one way to do that. Software developers who report a strong culture of shared learning on their teams (e.g., disclosing learning goals to each other, feeling supported by their managers in learning time forming a productive part of work activities) have been shown to face less anxiety and identity threat during times of rapid technology change (Hicks, Lee & Foster-Marks, 2023).

3.3 Stack Overflow and the Gamification of Communal Artifacts

The use of a tool shifts our behavior and our thinking. Viewed through a social lens, tool design can be a way of directing group dynamics. On this point, we look to lessons from Stack Overflow on shaping cumulative cultures.

Stack Overflow is a Q&A site for developer problem-solving which sought to create a library of detailed, high quality answers to every programming question. Its global scale gave practitioners access to peers far beyond their local communities, and became a primary means of learning a new technology. In contrast to traditional forums, questions

and answers are the prime information type, with other features and social rituals designed to support their creation. Stack Overflow was designed to encourage questions and answers which were specific, actionable, and tightly scoped. Those questions which provided context on the problem the asker was trying to solve, and included code snippets, were favored over broader, more personal requests.

The site features voting, causing the most favored posts to be shown first on the page, rather than displaying chronologically. Voting acts as a form of social glue, allowing users to build consensus on what "good" content is (Anderson et al., 2012). Voting also conveys reputation points to posters, where various tiers of "rep" unlock new site capabilities. As a user progresses in privilege levels, they gain the ability to edit and comment on posts, allowing them to clarify intentions and modify artifacts produced by peers. For users most invested in Stack Overflow's grander vision, this provided a means of problem-solving dialogue and shared maintenance. At the highest rep levels, users were able to assist volunteer moderators and staff in triage tasks, stewarding Stack Overflow itself through these small units of work.

Stack Overflow was famous for its gamification, but early leaders and team members invested large quantities of time collaborating and debating with users about the best way to run the site (For examples, see many contributions to discussions, e.g. Atwood, n.d.). We argue this deliberate relationship-building was what animated the system with its initial energy, and the site's success cannot be seen without the context of this community of practice.

Gamification is powerful for funneling group motivations and promoting knowledge-sharing, but at scale, can create a rigidly defined system which is not amenable to new types of contributions and in particular, contributions that violate shared social norms. Without active design against this, communities can be steered by social biases. For instance, in studies of reputation on Stack Overflow, estimated perceived demographics such as being a female user have been documented as systematically disadvantaged in being granted rewards for answers by other users (May, Wachs, & Hannák, 2019). By reassessing how different aspects of a tool or system works for different types of practitioners, it becomes possible to maintain scale while serving more users' interests. Social gamification, in the absence of supportive structures and careful communal design, can produce unintended negative effects like incentivizing extrinsic rather than intrinsic motivation, and can have different behavioral outcomes for different groups (Herzig, Ameling & Schill, 2015).

As Stack Overflow's popularity skyrocketed, and the site was flooded with introductory questions by new developers who didn't understand the rules, seasoned contributors felt the need to protect a space they treasured. Absent tooling to encourage constructive interactions between these distinct groups, tensions rose, users blamed each other, and the site developed a reputation for hostility and shaming (e.g., Ford et al., 2016 writes: *"Participants perceived the style of communication on Stack Overflow as blunt and impersonal."*). Such negative perceptions are commonly voiced by newcomers seeking entrypoints into shared technical communities (Steinmacher et al., 2015). An alternative we can imagine to this trajectory is investing in new functionality which, for example, allowed new posters to start out in a staging zone, with experienced community members able to and rewarded for taking on a mentorship role. However, site traffic remained high, and work was prioritized elsewhere. Tensions resulting from the site's increasing scale began between different generations of users, but evolved into a high conflict relationship between community and parent company (Edwards, 2024).

A well-defined shared vision with well-codified customs can be a boon to a cumulative culture. However, to ensure a group can continue to innovate, perpetual re-negotiation of how to collaborate can become necessary. For example, developers experiencing field-level shifts in beliefs about what it means to be a software developer can report high levels of threat to dearly-held identities (Hicks, Lee & Foster-Marks, 2024). Such shifts may lead community members to change their evaluation of the value of participating in community knowledge efforts altogether (e.g., Edwards, 2024).

4 CONCLUSION

In this review, we have argued that it is time to move past debates about increasing individual developer productivity (which is frequently narrowly operationalized as developer *output*), and shift focus to creating a broader science of collective developer *problem-solving*. Conventionally, where cognitive science is integrated into software research, individual cognitive effects have been treated as possible explanatory factors in developer problem-solving (e.g., working memory capacity). In order to reach our goals of innovation and software practitioner thriving at scale, a focus on the executive functions is overly narrow. In this review, we make a call to action for software researchers to incorporate the emerging sciences of human innovation, creativity, and cumulative culture in how we understand software developer problem-solving.

The models that we invoke to explain developer problem-solving matter. Progress in software development is frequently stereotyped as being the result of solitary work, individual genius, and the result of a rare quality of "innate brilliance," a biased belief which many people hold about STEM fields, which leads to systematic devaluation of groups stereotyped as "less likely to be brilliant," and which has been connected to a systematic failure to recognize the social underpinnings of innovation and the social goals of potential technology practitioners (Diekman et al., 2016; Bian et al., 2018; Shaw, 2022; Hicks et al., 2023).

An influential and still-referenced 1968 case study in software claimed it was possible to detect a "10x developer" who solves tasks much faster than others–a claim which was based on a handful of individuals, a narrowly defined set of laboratory programming tasks, and the interpretation of between-individual differences without accounting for within-individual variation (Sackman et al., 1968; Currie, 2016; Nichols, 2019; Shrikanth et al., 2021). The notion of a "10x developer" became frequently referenced as an enduring idea, despite originating from sparse, uneasy evidence. It is worth noting that the original argument proposed the large-scale automated detection and exclusion of *low* performing developers based on the time it takes them to complete a narrowly-defined laboratory task. This narrative is one example of the common argument that innovation in software development is driven by a few outliers, these outliers are rare, and that teams should be designed around supporting those elite few while excluding the majority (Abbate, 2021).

These narratives about what developer problem-solving *is* must be considered in their historical context. For instance, psychological science has a troubled history supporting the use of poorly measured individual ability "latent factors" to predict human potential and aptitude for complex work across the 1960s and 1970s despite failures to design for equitable assessment, or outright attempts to design for exclusion (Boykin, 2023). This model was imported directly into attempts to detect individual *programmer* ability with measures drawn directly from general "cognitive ability" tests, such as the IBM programming aptitude test (Wolfe, 1971). Despite the chasm between the constructs of such ability

tests and the actual innovation outcomes achieved by real working teams, such limited models of what technological aptitude was or could be were used by real companies to constrain and define who was let in to the field of software development and who, therefore, had the opportunity to solve technological problems in the first place. Our theoretical models about the world can and do change the world.

We argue that today, many different factors continue to come together to reinforce a limited model of developer problem-solving. The idea that technological problem-solving is a fixed, innate trait located within an individual, regardless of their context and learning over time, may be reinforced by a tendency to make essentialist inferences about human ability, overweighting the role of innate skill and underweighting over time learning, within-individual variation, and early mistakes as a core part of problem-solving (Gelman, 2004; Bjork et al., 2013; Haimovitz & Dweck, 2017). Such biases can lead to normative social beliefs that only "certain people" can perform certain types of work (Rhodes & Mandalaywala, 2017). For instance, the belief that there are "math people" and "not-math people", and that early mistakes may signal that one does not have the potential to succeed in a type of work, despite evidence that learning and early mistakes are a vital part of innovation cultures (Murphy & Thomas, 2008). A narrow emphasis on an individual's executive functions may further lead to a reductionistic view of software development problem-solving and who is capable of it, particularly given a history of researchers interpreting cognitive differences as deficits which may be more accurately understood as divergent strengths (Mittal et al., 2015; Fendinger et al., 2023).

It is time for a new model of software problem-solving. We draw on cross-cultural evidence about human development and the sociocognitive mechanisms of innovation to argue that developer problem-solving is better understood as the product of a *cumulative culture*. In a cumulative culture, knowledge, practices, behaviors and innovations are transmitted socially, enabling our collective cultural artifacts to be modified and improved upon (Rawlings & Legare, 2021). Capitalizing on social learning, collective innovation transcends individual capacities by allowing individuals to use and contribute to a shared solution space (Dean et al., 2014). In our view, this more accurately and empathically reflects the lived reality of software practitioners who, contrary to reductionist stereotypes, measurably benefit from being in environments that see social learning as a part of their technical work (Hicks et al., 2023; Hicks, 2024; Hicks et al., 2024), problem-solve in concert with communities of practice (Paasivaara & Lassenius, 2014; Chattopadhyay et al., 2021), include a wide range of end-user programmers and technology-adjacent workers solving software development problems outside of expected contexts (Shaw, 2022), and see their technological progress as operating within, not despite, social needs such as teaching and learning (Storey et al., 2021; Hicks, 2023).

References

Atwood, J. (n.d.) https://meta.stackexchange.com/users/1/jeff-atwood?tab=summary

Abbate, J. (2021). Coding is not empowerment. In T. S. Mullaney, B. Peters, M. Hicks, & K. Philip (Eds.), *Your computer is on fire* (pp. 253–271). MIT Press. https://doi.org/10.7551/mitpress/10993.001.0001

Ackerman, P. L., Kanfer, R., & Goff, M. (1995). Cognitive and noncognitive determinants and consequences of complex skill acquisition. *Journal of Experimental Psychology: Applied*, *1*(4), 270.

Anderson, A., Huttenlocher, D., Kleinberg, J., & Leskovec, J. (2012, August). Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 850-858).

Avgeriou, P., Ozkaya, I., Chatzigeorgiou, A., Ciolkowski, M., Ernst, N. A., Koontz, R. J., ... & Shull, F. (2023, May). Technical debt management: The road ahead for successful software delivery. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering* (ICSE-FoSE) (pp. 15-30). IEEE.

Ayres, P., & Paas, F. (2012). Cognitive load theory: New directions and challenges. *Applied Cognitive Psychology*, *26*(6), 827-832.

Bacchelli, A., & Bird, C. (2013, May). Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering* (ICSE) (pp. 712-721). IEEE.

Baldwin, C., & Von Hippel, E. (2011). Modeling a paradigm shift: From producer innovation to user and open collaborative innovation. *Organization Science*, *22*(6), 1399-1417.

Bian, L., Leslie, S. J., & Cimpian, A. (2018). Evidence of bias against girls and women in contexts that emphasize intellectual ability. *American Psychologist, 73*(9), 1139.

Bjork, R. A., Dunlosky, J., & Kornell, N. (2013). Self-regulated learning: Beliefs, techniques, and illusions. *Annual Review of Psychology, 64*(1), 417-444.

Boykin, C. M. (2023). Constructs, tape measures, and mercury. *Perspectives on Psychological Science*, *18*(1), 39-47.

Burdett, E. R., & Ronfard, S. (2023). Tinkering to innovation: How children refine tools over multiple attempts. *Developmental Psychology, 59*(6), 1006.

Brockner, J., & Sherman, D. K. (2019). Wise interventions in organizations. *Research in Organizational Behavior,* 39, 100125.

Brockner, J., & Wiesenfeld, B. M. (2016). Self-as-object and self-as-subject in the workplace. *Organizational Behavior and Human Decision Processes*, 136, 36-46.

Carr, K., Kendal, R. L., & Flynn, E. G. (2016). Eureka!: What is innovation, how does it develop, and who does it?. *Child Development, 87*(5), 1505-1519.

Chattopadhyay, S., Ford, D., & Zimmermann, T. (2021). Developers who vlog: dismantling stereotypes through community and identity. *Proceedings of the ACM on Human-Computer Interaction, 5*(CSCW2), 1-33.

Cybersecurity and Infrastructure Security Agency (CISA). (2024, October 16). Product Security Bad Practices. Retrieved from: https://www.cisa.gov/resources-tools/resources/product-security-bad-practices

Currie, T. C. (2016, June 3). Are you a 10x programmer? Or just a jerk? *The New Stack*. https://thenewstack.io/10x-programmer-just-jerk/

de Vries, H. B., & Lubart, T. I. (2019). Scientific creativity: divergent and convergent thinking and the impact of culture. *The Journal of Creative Behavior, 53*(2), 145-155.

Dean, L. G., Vale, G. L., Laland, K. N., Flynn, E., & Kendal, R. L. (2014). Human cumulative culture: a comparative perspective. *Biological Reviews, 89*(2), 284-301.

Diekman, A. B., Steinberg, M., Brown, E. R., Belanger, A. L., & Clark, E. K. (2017). A goal congruity model of role entry, engagement, and exit: Understanding communal goal processes in STEM gender gaps. *Personality and Social Psychology Review, 21*(2), 142-175.

Dietze, P., & Knowles, E. D. (2021). Social class predicts emotion perception and perspective-taking performance in adults. *Personality and Social Psychology Bulletin, 47*(1), 42-56.

Edwards, B. (2024, May 9). *Stack Overflow users sabotage their posts after OpenAI deal.* Ars Technica. Retrieved Nov 12, 2024 from: https://arstechnica.com/information-technology/2024/05/stack-overflow-users-sabotage-their-posts-after-openai-deal/

Egelman, C. D., Murphy-Hill, E., Kammer, E., Hodges, M. M., Green, C., Jaspan, C., & Lin, J. (2020, June). Predicting developers' negative feelings about code review. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 174-185).

Elias, S. A. (Ed.). (2012). Origins of human innovation and creativity (Vol. 16). Elsevier.

Eon Duval, P., Frick, A., & Denervaud, S. (2023). Divergent and Convergent Thinking across the Schoolyears: A Dynamic Perspective on Creativity Development. *The Journal of Creative Behavior, 57*(2), 186-198.

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015, August). Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 50-60).

Fendinger, N. J., Dietze, P., & Knowles, E. D. (2023). Beyond cognitive deficits: how social class shapes social cognition. *Trends in Cognitive Sciences, 27*(6), 528-538.

Ford, D., Smith, J., Guo, P. J., & Parnin, C. (2016, November). Paradise unplugged: Identifying barriers for female participation on stack overflow. In Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering (pp. 846-857).

Ford, D., Behroozi, M., Serebrenik, A., & Parnin, C. (2019, May). Beyond the code itself: How programmers really look at pull requests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society* (ICSE-SEIS) (pp. 51-60). IEEE.

Franco, L. (2024, September 19). *Paying down tech debt: Further learnings.* The Pragmatic Engineer. Retrieved from: https://blog.pragmaticengineer.com/paying-down-tech-debt-further-learnings

Fulton, K. R., Chan, A., Votipka D., Hicks, M., & Mazurek, M. L., (2021, August). Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. *USENIX. Proceedings of the Seventeenth Symposium on Usable Privacy and Security.* Retrieved from: https://www.usenix.org/system/files/soups2021-fulton.pdf

Gaskins, S., & Alcalá, L. (2023). Studying executive function in culturally meaningful ways. *Journal of Cognition and Development, 24*(2), 260-279.

Gauvain, M., & Perez, S. (2015). Cognitive development and culture. *Handbook of child psychology and developmental science*, 1-43.

Gelman, S. A. (2004). Psychological essentialism in children. *Trends in Cognitive Sciences, 8*(9), 404-409.

Gonçales, L., Farias, K., da Silva, B., & Fessler, J. (2019, May). Measuring the cognitive load of software developers: A systematic mapping study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension* (ICPC) (pp. 42-52). IEEE.

Gonçalves, P. W., Fregnan, E., Baum, T., Schneider, K., & Bacchelli, A. (2022). Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. *Empirical Software Engineering, 27*(4), 99.

Herrmann, E., Call, J., Hernández-Lloreda, M. V., Hare, B., & Tomasello, M. (2007). Humans have evolved specialized skills of social cognition: The cultural intelligence hypothesis. *Science, 317*(5843), 1360-1366.

Herzig, P., Ameling, M., & Schill, A. (2015). Workplace psychology and gamification: Theory and application. In T. Reiners & L. C. Wood (Eds.), *Gamification in education and business* (pp. 451–471). Springer International Publishing. https://doi.org/10.1007/978-3-319-10208-5_23

Heyes, C. M. (1994). Social learning in animals: categories and mechanisms. *Biological Reviews, 69*(2), 207-231.

Hicks, C. M. (2023, June 9). "It's Like Coding in the Dark": The need for learning cultures within coding teams. *Preprint.* https://doi.org/10.31234/osf.io/nz8m3

Hicks, C. M. (2024, January 25). Psychological Affordances Can Provide a Missing Explanatory Layer for Why Interventions to Improve Developer Experience Take Hold or Fail. *Preprint.* https://doi.org/10.31234/osf.io/qz43x

Hicks, C. M., Lee, C. S., & Foster-Marks, K. (2024, April 20). The New Developer: AI Skill Threat, Identity Change & Developer Thriving in the Transition to AI-Assisted Software Development. *Preprint.* https://doi.org/10.31234/osf.io/2gej5

Hicks, C. M., Lee, C. S., & Ramsey, M. (2024). Developer Thriving: four sociocognitive factors that create resilient productivity on software teams. *IEEE Software*, vol. 41, no. 4, pp. 68-77, July-Aug. 2024, doi: 10.1109/MS.2024.3382957

Holterhoff, K. (2024, March 28). The future of frontend is already here. *RedMonk*. Retrieved from: https://redmonk.com/kholterhoff/2024/03/28/the-future-of-frontend-is-already-here/

Kalyuga, S., & Singh, A. M. (2016). Rethinking the boundaries of cognitive load theory in complex learning. *Educational Psychology Review*, *28*, 831-852.

Kirschner, P. A., Ayres, P., & Chandler, P. (2011). Contemporary cognitive load theory research: The good, the bad and the ugly. *Computers in Human Behavior*, 27(1), 99-105.

Kurtessis, J. N., Eisenberger, R., Ford, M. T., Buffardi, L. C., Stewart, K. A., & Adis, C. S. (2017). Perceived organizational support: A meta-analytic evaluation of organizational support theory. *Journal of Management*, *43*(6), 1854-1884.

Lee, C. S., & Hicks, C. M. (2024). Understanding and effectively mitigating code review anxiety. *Empirical Software Engineering*, *29*(6), 161.

Legare, C. H. (2017). Cumulative cultural learning: Development and diversity. *Proceedings of the National Academy of Sciences, 114*(30), 7877-7883.

Legare, C. H., Sobel, D. M., & Callanan, M. (2017). Causal learning is collaborative: Examining explanation and exploration in social contexts. *Psychonomic Bulletin & Review*, 24, 1548-1554.

Lichand, G., Ash, E., Arold, B., Gudino, J., Doria, C. A., Trindade, A., ... & Yeager, D. (2024). Measuring student mindsets at scale in resource-constrained settings: A toolkit with an application to Brazil during the pandemic. *Journal of Research on Adolescence*. https://doi.org/10.1111/jora.13008

Masood, Z., Hoda, R., Blincoe, K., & Damian, D. (2022). Like, dislike, or just do it? How developers approach software development tasks. *Information and Software Technology*, *150*, 106963.

Matthews, G., Wohleber, R. W., & Lin, J. (2018). Stress, skilled performance, and expertise: Overload and beyond. In Ward, P., Schraagen, J. M., Gore, J., & Roth, E. M., (Eds.), *The Oxford Handbook of Expertise*. Oxford University Press.

May, A., Wachs, J., & Hannák, A. (2019). Gender differences in participation and reward on Stack Overflow. *Empirical Software Engineering*, *24*, 1997-2019.

McCaffrey, T. (2012). Innovation relies on the obscure: A key to overcoming the classic problem of functional fixedness. *Psychological Science*, *23*(3), 215-218.

Mittal, C., Griskevicius, V., Simpson, J. A., Sung, S., & Young, E. S. (2015). Cognitive adaptations to stressful environments: When childhood adversity enhances adult executive function. *Journal of Personality and Social Psychology*, *109*(4), 604.

Munakata, Y., & Michaelson, L. E. (2021). Executive functions in social context: Implications for conceptualizing, measuring, and supporting developmental trajectories. *Annual Review of Developmental Psychology*, *3*(1), 139-163.

Murabito, M. (2023, November 16). *Platform engineering reduces cognitive load and raises developer productivity*. The New Stack. https://thenewstack.io/platform-engineering-reduces-cognitive-load-and-raises-developer-productivity/ Retrieved October 6, 2024

Nichols, W. R. (2019). The end to the myth of individual programmer productivity. *IEEE Software, 36*(5), 71-75.

Niebaum, J. C., & Munakata, Y. (2023). Why doesn't executive function training improve academic achievement? Rethinking individual differences, relevance, and engagement from a contextual framework. *Journal of Cognition and Development*, *24*(2), 241-259.

O'Dell, J., (2012, January 4). Why Walmart is Using Node.js. *VentureBeat.* Retrieved from: https://venturebeat.com/dev/why-walmart-is-using-node-js/

Paasivaara, M., & Lassenius, C. (2014). Communities of practice in a large distributed agile software development organization–Case Ericsson. *Information and Software Technology, 56*(12), 1556-1577.

Piantadosi, V., Scalabrino, S., Serebrenik, A., Novielli, N., & Oliveto, R. (2023). Do attention and memory explain the performance of software developers?. *Empirical Software Engineering*, *28*(5), 112.

Piva, E., Rentocchini, F., & Rossi-Lamastra, C. (2012). Is open source software about innovation? Collaborations with the open source community and innovation performance of software entrepreneurial ventures. *Journal of Small Business Management*, *50*(2), 340-364.

Prasad, K., Norton, K., & Coatta, T. (2014). Node at Linkedln: The Pursuit of Thinner, Lighter, Faster. *Communications of the ACM*, *57*(2), 44-51.

Rawlings, B. S. (2022). After a decade of tool innovation, what comes next?. *Child Development Perspectives, 16*(2), 118-124.

Rawlings, B., & Legare, C. H. (2021). Toddlers, tools, and tech: The cognitive ontogenesis of innovation. *Trends in Cognitive Sciences, 25*(1), 81-92.

Rhodes, M., & Mandalaywala, T. M. (2017). The development and developmental consequences of social essentialism. *Wiley Interdisciplinary Reviews: Cognitive Science, 8*(4), e1437.

Rifkin-Graboi, A., Goh, S.-K.-Y., Chong, H. J., Tsotsi, S., Sim, L. W., Tan, K. H., Chong, Y. S., & Meaney, M. J. (2021). Caregiving adversity during infancy and preschool cognitive function: Adaptations to context? *Journal of Developmental Origins of Health and Disease, 12*(6), 890–901. https://doi.org/10.1017/S2040 174420001348

Russo, D., & Ciancarini, P. (2017). Towards antifragile software architectures. *Procedia Computer Science*, *109*, 929-934.

Rust Team. (n.d.) Learn Rust. Retrieved from: https://www.rust-lang.org/learn

Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM, 11*(1), 3–11.

Sadowski, C., & Zimmermann, T. (Eds.). (2019). *Rethinking productivity in software engineering* (p. 310). Springer Nature.

Schnotz, W., & Kürschner, C. (2007). A reconsideration of cognitive load theory. *Educational Psychology Review*, 19, 469-508.

Selenko, E., Bankins, S., Shoss, M., Warburton, J., & Restubog, S. L. D. (2022). Artificial intelligence and the future of work: A functional-identity perspective. *Current Directions in Psychological Science, 31*(3), 272-279.

Shaw, M. (2022). Myths and mythconceptions: What does it mean to be a programming language, anyhow?. *Proceedings of the ACM on Programming Languages*, *4*(HOPL), 1-44.

Shrikanth, N. C., Nichols, W., Fahid, F. M., & Menzies, T. (2021). Assessing practitioner beliefs about software engineering. *Empirical Software Engineering, 26*(4), 73.

Stajkovic, A. D., & Luthans, F. (1998). Self-efficacy and work-related performance: A meta-analysis. *Psychological Bulletin*, *124*(2), 240.

Steinmacher, I., Conte, T., Gerosa, M. A., & Redmiles, D. (2015, February). Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing* (pp. 1379-1392).

Storey, M. A., Zimmermann, T., Bird, C., Czerwonka, J., Murphy, B., & Kalliamvakou, E. (2019). Towards a theory of software developer job satisfaction and perceived productivity. *IEEE Transactions on Software Engineering, 47*(10), 2125-2142.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, *12*(2), 257-285.

Sweller, J., Van Merrienboer, J. J., & Paas, F. (2019). Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review*, *31*, 261-292.

Tennie, C., Call, J., & Tomasello, M. (2009). Ratcheting up the ratchet: on the evolution of cumulative culture. Philosophical Transactions of the Royal Society B: Biological Sciences, 364(1528), 2405-2415.

Townley, A. L. (2020). Leveraging communities of practice as professional learning communities in science, technology, engineering, math (STEM) education. *Education Sciences*, *10*(8), 190.

Trinkenreich, B., Stol, K. J., Sarma, A., German, D. M., Gerosa, M. A., & Steinmacher, I. (2023, May). Do i belong? modeling sense of virtual community among linux kernel contributors. In *2023 IEEE/ACM 45th International Conference on Software Engineering* (ICSE) (pp. 319-331). IEEE.

Trott, K., & Xiao, Y. (2015, December 3) Debugging Node.js in Production. *Netflix Technology Blog*. Retrieved from: https://netflixtechblog.com/debugging-node-js-in-production-75901bb10f2d

Vaesen, K. (2012). The cognitive bases of human tool use. *Behavioral and Brain Sciences, 35*(4), 203-218.

Vale, G. L., Flynn, E. G., & Kendal, R. L. (2012). Cumulative culture and future thinking: Is mental time travel a prerequisite to cumulative cultural evolution?. *Learning and Motivation*, *43*(4), 220-230.

Vygotsky, L. S. (1978). Mind in society: The development of higher psychological processes (Vol. 86). Harvard University Press.

Walton, G. M. (2014). The new science of wise psychological interventions. *Current Directions in Psychological Science, 23*(1), 73-82.

Walton, G. M., & Spencer, S. J. (2009). Latent ability: Grades and test scores systematically underestimate the intellectual ability of negatively stereotyped students. *Psychological Science, 20*(9), 1132-1139.

Werner, K. M., & Berkman, E. T. (2024). Motivational Dynamics of Self-Control. *Current Opinion in Psychology*, 101859.

Winters, T., Manshreck, T., & Wright, H. (2020). *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media.

Wolfe, J. M. (1971, January). Perspectives on testing for programming aptitude. In *Proceedings of the 1971 Annual Conference of the ACM* (pp. 268-277).

Yanaoka, K., Michaelson, L. E., Guild, R. M., Dostart, G., Yonehiro, J., Saito, S., & Munakata, Y. (2022). Cultures crossing: The power of habit in delaying gratification. *Psychological Science*, *33*(7), 1172-1181.

Young, E. S., Frankenhuis, W. E., DelPriore, D. J., & Ellis, B. J. (2022). Hidden talents in context: Cognitive performance with abstract versus ecological stimuli among adversity-exposed youth. *Child Development*, *93*(5), 1493-1510.