Digital Nudges for Encouraging Developer Behaviors

Dwayne Christian Brown, Jr.

North Carolina State University
dcbrow10@ncsu.edu
http://www4.ncsu.edu/~dcbrow10/

Abstract. Software engineering researchers create tools and practices designed to help developers accomplish programming tasks. Unfortunately, software engineers often ignore these useful resources in practice. While automated recommender systems have been created to automatically increase awareness and encourage adoption of developer actions, research shows that face-to-face recommendations between colleagues is still the most effective mode of discovery for software engineers. To improve the effectiveness of automated tool recommendations, I propose integrating concepts from *nudge theory*, a behavioral science framework that examines how to influence human behavior and improve decision-making. This work seeks to apply this theory into software engineering to explore the impact of nudges for improving developer behavior and introducing ***developer recommendation choice architectures*** to design and frame decisions in the context of adopting programming tools and practices. The contributions of this work are: 1) a *conceptual framework* explaining how to apply concepts from nudge theory when making recommendations to software developers, 2) a *set of experiments* that support and evaluate the conceptual framework, and 3) an *automated recommender system*, nudge-bot, that utilizes the proposed framework to recommend useful developer behaviors. My goal is to demonstrate that automated nudges can encourage software engineers to adopt beneficial developer behaviors.

Thesis Statement

By incorporating ***developer recommendation choice architectures*** into recommendations for software engineers, we can ***nudge*** developers to adopt behaviors useful for improving code quality and developer productivity.

1 Introduction

1.1 Developer Recommendations

Humans make approximately 35,000 decisions every day.[1] Likewise, software engineers are also frequently faced with choices to make while developing code. As our society becomes more dependent on software products [3], it is becoming increasingly important to find ways to improve the behavior and decision-making of software developers. In his book *The New Kingmakers: How Developers Conquered the World*, Stephen O'Grady describes the influence of software engineers' choices on the economy and society by noting "Developers are the most-important constituency in technology. They have the power to make or break business, whether by their preferences, their passions, or their own products...Developers are now the real decision makers in technology. Learning how to best negotiate with these New Kingmakers, therefore, could mean the difference between success and failure" [89, p. 3-4]. Furthermore, the Hacker Noon blog refers to decision-making as "the most undervalued skill in software engineering" and "the most important skill in software development", even moreso than coding skills.[2] and Li and colleagues discovered that the ability to make effective decisions is an important characteristic of being a great software engineer [68].

While decision-making is an important aspect of software engineering, developers often make less than ideal choices in their work. For example, software engineers often avoid adopting useful *developer behaviors*, or processes designed to support programmers in completing software development tasks. For instance, Johnson and colleagues found that developers rarely use static analysis tools to automatically check for defects and prevent errors in code [58]. To help developers make better decisions, software engineering researchers have explored creating recommender systems to automatically guide users. The ACM International Conference on Recommender Systems (RecSys) defines recommender systems as "software applications that aim to support users in their decision-making while interacting with large information spaces" [3]. Fischer and colleagues also argue that *active help systems* that can automatically make recommendations to users completing tasks are more effective than *passive help systems* requiring users to seek help [42]. Similarly, recommendation systems for software engineering are designed to actively assist developers in completing various tasks and provide information when making decisions in their work [92]. For example, Spyglass is an automated recommender system for software engineers that suggests code navigation tools in the Eclipse integrated development environment (IDE) to help developers save time and effort while searching through code to complete programming tasks [112].

[1] `https://go.roberts.edu/leadingedge/the-great-choices-of-strategic-leaders`

[2] `https://hackernoon.com/decision-making-the-most-undervalued-skill-in-software-engineering-f9b8e5835ca6`

[3] `https://recsys.acm.org/`, as quoted by [92]

While many automated suggestion approaches have been developed to help developers make better choices, research shows that face-to-face recommendations between humans are the most effective. Murphy-Hill and colleagues explored how developers discover new software engineering tools and found that *peer interactions*, or the process of discovering tools from coworkers during normal work activities, are more effective compared to other technical approaches [86]. The main reason developers provided for peer interactions being the most effective mode of tool discovery is the respect and trust built between colleagues. However, even though face-to-face interactions are an effective method for developer recommendations, they are becoming less practical for recommendations in software engineering. For example, Murphy-Hill also discovered that peer interactions occur infrequently among developers in the workplace [86]. There are many barriers to peer interactions in industry, such as mandated tools and processes by management, *physical isolation* where programmers are increasingly working alone remotely, and *developer inertia* where software engineers do not feel the need to share or adopt new practices and tools [85]. Additionally, many automated systems have proven ineffective for improving developer decision-making. For example, Viriyakattiyaporn and colleagues found that the inability to deliver suggestions in a timely manner discouraged programmers from adopting recommendations to improve code navigation with Spyglass [111]. Thus, this points to a need for new paradigms for making effective recommendations to developers.

1.2 Motivating Example

To understand the impact of the decline of peer interactions and inadequacy of automated recommendation systems for software engineering on decision-making for software engineers, consider the example of Cassius. Cassius is an experienced software engineer maintaining several popular open source JavaScript projects on GitHub. However, he is unaware of several major bugs that exist in his repositories because he does not implement any static analysis tools in his projects. This is primarily because he is not familiar with useful tools to help developers automatically find and prevent defects in JavaScript code. Additionally, he is not compelled to use these tools because he has not had any exposure to them through his work and doesn't want to go through the hassle of integrating new systems into his workflow and development environment. Cassius normally works from home remotely, so he does not have many opportunities to learn about useful tools and processes during face-to-face interactions with peers. Various automated recommendations have also been ineffective in persuading Cassius to adopt new tools and practices. He often receives automated emails suggesting new tools, but usually ignores the these messages as marketing and spam. He also frequently observes pop-ups and tool tips recommending useful tools and features in his integrated development environment (IDE), but Cassius usually disregards those as well.

One day, Cassius notices a new pull request on one of his repositories. The pull request introduces Cassius to ESLint[4], an open source static analysis tool for finding errors in JavaScript code. He notices the description provides valuable information such as what the tool does, how to use it, and an example bug reported within his project's code. After further inspection, he notices the PR also modified the build configuration files to automatically add the tool for it to run during the continuous integration builds for his project. He just needs to merge the pull requests to add the static analysis tool to his repositories. Furthermore, he notices the same pull request was opened by a bot on several of his other JavaScript repositories. Due to the feedback, location, and accessibility of the recommendation, Cassius decides to merge the pull requests and adopt static code analysis into his repositories. In this proposal, I will discuss research that investigates effective designs for automated recommendations to improve developer behavior.

1.3 Research Overview

My research goal is, given a developer who is unaware of a useful developer behavior in a development situation, identify the most effective strategy to convince them to adopt the behavior. This goal can also be summed up in the following research question posed by Greg Wilson, software engineering researcher and co-founder of Software Carpentry,[5] who tweeted:

"I think the most interesting topic for software engineering research in the next ten years is, *'How do we get working programmers to actually adopt better practices?'*".[6]

To encourage developers to adopt better practices, this work will analyze and evaluate developer behavior adoption through the lens of behavioral science and economics. Behavioral science research has examined how to encourage humans to make better decisions, embrace beneficial behaviors, and accept new ideas. One framework used to influence human behavior and decision-making is *nudge theory*. A *nudge* refers to any factor that impacts how people make a decision without providing incentives or banning alternatives [105]. Referring back to the motivating example, the automated pull request Cassius receives is an example of a nudge. Cassius is not rewarded for accepting the recommendation and merging the pull request into his project. He is not prohibited from using other static analysis tools besides ESLint or or just simply ignoring the suggestion and closing the PR. In this thesis, I will argue that implementing recommendations to software engineers as nudges can improve developer behavior. To study the impact of integrating nudge theory into developer recommendations, I will adhere to the following plan of work: 1) determine what makes recommendations

[4] `https://eslint.org/`
[5] `https://software-carpentry.org/`
[6] `https://twitter.com/gvwilson/status/1142245508464795649?s=20`

effective to developers, 2) examine existing tools for recommending developer behaviors, and 3) create new automated systems and strategies to improve the effectiveness of recommendations to developers.

1.4 Research Contributions

The expected contributions of the research for this thesis include:

- a *conceptual framework* that uses nudge theory to characterize failures and propose improvements to developer recommendations,
- a set of *experiments* to evaluate and provide evidence for the conceptual framework, and
- *nudge-bot*, an *automated recommender system* to nudge software engineers to adopt developer behaviors based on our framework.

2 Background

This section provides background information on two concepts, *nudge theory* and *developer behaviors*, that are key for the research presented in this proposal.

2.1 Nudge Theory

To encourage adoption of developer behaviors, we plan to incorporate concepts from *nudge theory*. A nudge is defined as any factor "that alters behavior in a predictable way without forbidding alternatives or significantly changing economic incentives" [105, p. 6]. Nudges impact how humans make common everyday decisions, such as encouraging people to recycle more by increasing the size of recycling bins[7] and re-labelling trash cans as "landfills"[8]. In these cases, people still have the option not to recycle and are not rewarded for recycling, but are still persuaded by the size and naming of bins. Nudges are also used on a much larger scale to impact human behavior and decision-making. For example, the UK government implemented a Behavioural Insights Team[9], also known as the Nudge Unit, to improve behavior and decisions by citizens. An example of a nudge from this team includes encouraging companies to improve the recruitment of women, promotion of female employees, and reduce the gender pay gap by providing guides and feedback to companies encouraging actions such as including multiple women in the recruitment process, encouraging salary negotiations, introducing programs focused on increasing and fostering diversity, and more.[10] Similar nudge unit teams are becoming more popular around the

[7] http://nudges.org/2011/05/02/a-strategy-for-recycling-change-the-recyling-bin-to-garbage-bin-ratio/

[8] http://nudges.org/2010/08/25/its-not-a-garbage-can-its-a-small-landfill/

[9] https://www.gov.uk/government/organisations/behavioural-insights-team

[10] https://www.bi.team/blogs/new-for-employers-the-latest-evidence-on-what-works-to-reduce-the-gender-pay-gap/

world and have been implemented in other countries such as the United States, Denmark, and Italy [9]. Nudges are useful for automated recommendations to software engineers because they are interventions that are "easy and cheap" [105, p. 6]. Nudging for software engineers involves allowing alternative options for developers in addition to not providing incentives to encourage the selection of a desired target behavior. In this work, I aim to study the impact of using nudges to improve the decision-making of software engineers when faced with the choice to adopt or ignore useful developer behaviors.

Digital Nudges. Digital nudging refers to using technology and user interface design elements to nudge user behaviors in digital choice environments [113]. For example, the FitBit[11] smart watch nudges users to increase physical activity and adopt healthier lifestyle behaviors by monitoring exercise activity, providing feedback to users, and presenting data collected from friends and other users [113]. Weinmann argues understanding digital nudges is becoming increasingly important as more decisions are being made online because the designs of these systems will "always (either deliberately or accidentally) influences people's choices" [113, p. 433]. Additionally, Mirsch and colleagues argue that implementing digital nudging provides more advantages because they are "easier, faster and cheaper" and provide a lot more specified functionality compared to traditional nudges [79, p. 635]. While most prior work examining digital nudges examines their impact on the decision-making of software users, there is limited work exploring how they influence the choices and behavior of software developers. Software engineers are constantly presented with decisions in digital choice environments while writing code, such as whether or not to adopt useful programming behaviors and practices in their work.

Choice Architecture. Nudges and digital nudges are useful for improving human behavior because of their ability to influence the context and environment surrounding decision-making, or *choice architecture* [106]. Thaler and Sunstein note "nudges are everywhere" and "choice architecture, both good and bad, is pervasive and unavoidable...Choice architects can preserve freedom of choice while also nudging people in directions that will improve their lives" [105, p. 255]. Choice architecture is based on the fact that the presentation of choices often impacts decisions made. For example, one specific concept in choice architecture is the "default rule", which suggests decision-makers are most likely to select default options when making decisions. For example, the Washington State Parks Department modified the default for drivers to opt-out of an optional state park fee and raised over $1 million to support their state parks.[12] To increase the effectiveness of developer recommendations, I plan to use choice architecture to suggest design implications to automatically present developer behavior recommendations in digital choice environments.

[11] `https://www.fitbit.com/`

[12] `http://nudges.org/2009/10/21/switching-the-default-rule-to-save-state-parks-in-washington-state/`

Throughout this proposal, the term *nudge* is used to describe the implementation of digital nudges to improve developer behavior. This includes designing digital choice architectures that suggest beneficial practices to software engineers without providing incentives, restricting options, or forcing actions. Furthermore, while nudge theory can be applied to many different facets of software engineering such as the design of IDEs, programming languages, and physical workspaces, this work primarily focuses on implementing nudges and improving choice architectures for developers while completing programming tasks. The primary developer decision-making environment used for this research is GitHub, a popular online code hosting site with over 31 million developers, 96 million repositories, and 1 billion of code contributions.[13] To create these nudges, I plan to design and evaluate automated recommendations with software robots, or *bots*, to recommend developer behaviors. Thus, this work aims to discover if nudges can encourage developers to adopt useful software engineering practices when faced with choices during real-world programming situations.

2.2 Developer Behaviors

Developer behaviors refer to practices designed to support and aid software developers in the completion of programming tasks. An example of one of these beneficial developer behaviors is tool adoption. The IEEE Software Engineering Body of Knowledge (SWEBOK), a suite of widely accepted software engineering practices and standards, suggests using development tools is a "good practice" and can "enhance the chances of success over a wide range of project [103, p. A-4]. Jazayeri further argues that development tools have become so vital to software engineering that tool usage and the ability to switch between tools should be integrated into software engineering education [57]. Researchers and toolsmiths have created tools to help developers save time and effort completing programming tasks and evaluated their impact on development teams and products. For example, static analysis tools are systems which automatically examine code to find and detect errors without running the program. Studies show static analysis tools provide many benefits to projects such as improving code quality [6], preventing errors [19], decreasing debugging time [65], lowering development costs, and reducing developer effort [100]. However, research also shows developers often ignore these tools in practice. Johnson discovered that developers often don't use static analysis tools primarily because of their result understandability, customizability, and false positives in the tool output [58]. Similarly, researchers have explored why software engineers avoid using development tools for security [121], debugging [29], refactoring [82], documentation [45], build automation [91], continuous integration [54], and more.

This developer behavior adoption problem also exists for other beneficial software engineering activities outside of development tool usage. For instance, research shows implementing agile software development methodologies provides benefits to teams such as improved communication, faster releases, increased flexibility in design, and improved code quality [16] and the SWEBOK presents agile

[13] `https://octoverse.github.com/`

as a useful software engineering method [103, p. 9-9]. However, Nerur outlined challenges hindering migration to agile processes that impact the Management and organization, People, Processes, and Technologies preventing teams from adopting this practice [87]. The goal of my work is to make effective recommendations to software engineers to encourage adoption of useful developer behaviors. Tilley and colleagues argue that adoption of research-off-the-shelf (ROTS) software developed for industry practitioners should be a primary goal for software engineering researchers [107]. Furthermore, Wohlin presents general challenges with integrating empirical software engineering research from academia into industry including lack of trust, differing goals, and the transferring of knowledge and technologies [120]. To help bridge the gap between software engineering research and practice, I aim to explore the impact of using nudge theory to increase awareness and encourage adoption of beneficial developer behaviors evaluated by researchers for improving code quality and developer productivity in industry.

3 Preliminary Findings

To develop a conceptual framework for improving developer behavior recommendations, we first performed evaluations examining development tool recommendations to determine effective strategies.

3.1 [interactions] "How Software Users Recommend Tools to Each Other" (Completed, Spring 2017)

Motivation: The first step in my plan of work is to explore what makes effective recommendations to software engineers. While many automated system-to-user recommendation systems have been developed to increase awareness and adoption of development tools, prior work shows that user-to-user tool recommendations, or *peer interactions*, are the most effective method for discovering new tools. There is limited research examining why user-to-user recommendations are so effective, and to better understand why users prefer recommendations from peers we conducted a user study to observe peer interactions and analyze different characteristics of the recommendations. The characteristics we analyzed were motivated by psychology and persuasion theory as well as prior research examining peer interactions, and provide insights into why tool recommendations between peers is effective and implications for improving automated recommender systems.

Peer Interactions: Peer interactions are defined as the process of discovering tools from colleagues during normal work activities [86]. Murphy-Hill and colleagues examined different modes of tool discovery in software engineering, including peer interactions, random tool encounters, tutorials, discussion threads, written descriptions, and social media. They discovered that peer interactions were the most effective way developers reported learning about new software engineering tools [85,86]. There are two types of peer interactions that differ on

how the recommendation is made between two colleagues: *peer observation* refers to when a user sees a colleague using an unfamiliar tool that they are unaware of and *peer recommendation* is when a user sees a colleague completing a task inefficiently and suggests a tool.

Research Question:

RQ What characteristics of peer interactions make recommendations effective?

Methodology: To evaluate peer interactions in this study, we designed a mixed-methods approach to collect qualitative and quantitative data collected from observing participants.

Data Collection. To evaluate the effectiveness of peer interactions, we observed pairs of software users completing data analysis tasks. We recruited undergraduate and graduate students at North Carolina State University as well as professional analysts from the NC State Laboratory for Analytic Sciences[14] (LAS) to participate in our study. For the remainder of this proposal, we will refer to student participants with the S- prefix and LAS participants with the L- prefix. Overall, 13 pairs of colleagues participated in this user study, seven pairs of students and six pairs of professional analysts. We requested participants complete a questionnaire survey to collect demographic information and conducted a semi-structured post-interview to gather more data for our results.

The study tasks involved analyzing data from the Titanic shipwreck and solving problems based on the Kaggle data science competition [59]. For our study we did not examine for correctness in task completion, but were interested in how participants recommended tools to each other to solve the tasks. We allowed participants to use the software of their choice for the tasks, but prohibited Internet use to prevent participants form looking up information about the tasks or how to use tools during the study. More information on the tasks, datasets, and study materials are publicly available online.[15] For each session, we screen and voice recorded the participants while they completed the tasks.

Identifying peer interactions. To recognize peer interactions between participants completing the study tasks, we developed a model based of of the GOMS (Goals, Operators, Methods, and Selection rules) model in Human-Computer Interaction [37]. This model for recognizing peer interactions, shown in Figure 1, is defined for two colleagues working together in a pair programming scenario where the user actively operating the keyboard and mouse is the driver and their peer is the navigator [32]. In Task Analysis, both peers analyze the task and develop a strategy to complete it. Task Execution refers to when the driver begins executing their strategy and the navigator notices a mismatch. Finally, in the

[14] `https://ncsu-las.org/`
[15] `http://www4.ncsu.edu/~dcbrow10/files/peer-interaction/study.html`

Dialogue a tool is recommended by the navigator inquiring about the driver's tool or suggesting a new tool to complete the task more efficiently.

Fig. 1: Recommendation Model

Characterizing peer interactions. We explored five characteristics of peer interactions: Politeness, Persuasiveness, Receptiveness, Time Pressure, and Tool Observability. These characteristics were motivated from research in psychology and qualitative results from Murphy-Hill's prior work on peer interactions [85]. We used prior work in other fields to compile a list of criteria for these characteristics defined with examples from this evaluation in Table 1. For each characteristic, we analyzed comments made in the dialogue between participants during peer interactions to observe these criteria.

1) Politeness: Research suggests politeness is important for making effective recommendations. For example, Whitworth suggests the reason Microsoft's Clippy recommender system was unsuccessful is because it was impolite [115]. Murphy-Hill and colleagues found that respect and trust were important factors in peer interactions for software engineers [85]. To measure politeness, we used Leech's six maxims for politeness: Tact, Generosity, Approbation, Modesty, Agreement, and Sympathy [66].

2) Persuasiveness: Prior work in persuasion theory suggests it is vital for making effective suggestions. For example, O'Keefe argues that in a wide variety of settings from courtrooms to families, "human decision making is shaped by persuasive communication" [90, p. 31]. Fogg also suggests that persuasiveness is necessary to convince users to adopt desired behaviors through software [44]. Shen et al. present three features of persuasive messages that were used to measure persuasiveness in peer recommendations between participants in this study: Content, Structure, and Style [97].

3) Receptiveness: Prior work shows that receptivity is important for making effective suggestions. For example, Fogg outlined best practices for creating persuasive technologies to persuade users to adopt target behaviors. One key practice is to choose an "audience that is most likely to be receptive to the targeted behavior" [44]. Fogg provides two criteria to define a receptive audience: Demonstrate Desire and Familiarity. Users who demonstrate desire express interest in discovering, using, or learning more information about target behaviors while users with familiarity express knowledge about the target behavior or environment.

4) Time Pressure Research suggests time pressure can impact the outcome of recommendations. For example, Andrews and Smith found that time constraints impact decision-making in marketing [4]. Additionally, Murphy-Hill and

colleague identified time pressure as a barrier to peer interactions in the form of project deadlines [85]. We did not strictly enforce a time limit for completing tasks in this study, but the suggested time was one hour. We measured time pressure by looking for statements mentioning time from participants and categorized peer interactions as either having time pressure or not.

5) Tool Observability Murphy-Hill and colleagues suggest that recommendations from systems should have noticeable causes and effects [85]. To analyze this, we examined the observability of tools recommended between participants in the study. Observability refers to whether or not tools are visible to users through a graphical user interface. To determine if the perception of tools impacts recommendations, we analyzed tools suggested between peers in our study and categorized them as Observable or Non-observable.

Determining the effectiveness of peer interactions. To measure effectiveness, each software tool recommendation between participants was categorized as *effective*, *ineffective*, and *unknown*. For effective recommendations, the recommendee used a tool after it was suggested by their partner for the remainder of their session for a majority of the opportunities it was applicable. For ineffective recommendations, the recommendee mostly ignored a tool recommended by their partner when they had a chance to utilize it in the study. Finally, unknown recommendations were the case where there was not another opportunity for the recommendee to use a suggested tool for the rest of their study session. Two researchers independently viewed recordings of each session to note instances of tool recommendations and categorize the peer interactions based on the criteria defined for politeness (Cohen's $\kappa = 0.50$), persuasiveness (Cohen's $\kappa = 0.28$), and receptiveness (Cohen's $\kappa = 0.51$). The coders came together to discuss and resolve any disagreements.

Results: In total, we discovered 142 total recommendations between participants in our user study: 71 effective; 35 ineffective; and 36 unknown. Table 2 presents the breakdown of effectiveness for each characteristic we examined in our study. Out of the peer interaction characteristics we analyzed, *receptiveness* was the only characteristic that significantly impacted the outcome of a tool recommendation between peers (Wilcoxon, $p = 0.0002$, $\alpha = .05$). These results indicate that the receptiveness of users is what makes recommendations to developers effective. In this study, we defined receptiveness using two criteria from prior work by Fogg on designing persuasive technology: ***Demonstrate Desire*** and ***Familiarity*** [44]. We suggest integrating these concepts into automated recommendations for developer behaviors to software engineers to improve the effectiveness of suggestions and increase adoption of useful programming tools and practices. These results were published in the 2017 Visual Languages and Human-Centric Computing (VL/HCC) conference [26].

Table 1: Peer interaction characteristics from `interactions` study

Politeness Criteria		
Tact	Definition	Minimize cost and maximize benefit to peer
	Polite	"We can do all of it together, just sort by level." - S9
	Impolite	"We can do a histogram...which is always sort of a pain in the butt to do in Excel." - L14
Generosity	Definition	Minimize benefit and maximize cost to self
	Polite	"CONCATENATE you can do. I can do this for you, very easily." - S10
	Impolite	"Maybe you should write a python script for this." - L6
Approbation	Definition	Minimize dispraise and maximize praise of peer
	Polite	"I'm not as good at the Excel stuff as you are." - L5
	Impolite	"This[partner's suggestion] is useless." - S14
Modesty	Definition	Minimize praise and maximize dispraise of self
	Polite	"From whatever limited knowledge of data analysis I have, I think you need to create a linear regression model..." - S14
	Impolite	"I'm very good at Paint." - S10
Agreement	Definition	Minimize disagreement and maximize agreement between peers
	Polite	"Do you want to use Python?" - S8
	Impolite	"No, no, no...Don't you want it comma separated? That's what I'm doing." - S14
Sympathy	Definition	Minimize antipathy and maximize sympathy between peers
	Polite	"We can try JMP..." ["I haven't done anything in JMP."] "Neither have I!" - L14
	Impolite	"It doesn't matter how you do it." - L16
Persuasiveness Criteria		
Content	Definition	Recommender provides credible sources to verify use of the tool
	Persuasive	"Go here, go to Data. Highlight that...Data, Sort, and it lets you pick two." - L8
	Unpersuasive	"Let's try to text filter, right?" - S5
Structure	Definition	Messages are organized by climax-anticlimax order of arguments and conclusion explicitness
	Persuasive	"I know that SUMIF is a type of function that allows you to combine the capabilities of SUM over a range with a condition that needs to be met." - S3
	Unpersuasive	"There's a thing on Excel where you can do that, where you can say if it is this value, include, if it is not, exclude...Yeah, IF." - S11
Style	Definition	Messages should avoid hedging, hesitating, questioning intonations, and powerless language
	Persuasive	"Click on title and do a Ctrl-A" - S13
	Unpersuasive	"I guess we're going to have to use some math calculations here, or a pivot table." - L9
Receptiveness Criteria		
Demonstrate Desire	Definition	Participant shows interest in using or learning more about a tool
	Receptive	"That was cool, how [the column] just populated." - S4
	Unreceptive	["So you want to use R for it?"] "No, no, no..." - S14
Familiarity	Definition	Participant explicitly expresses familiarity with the environment
	Receptive	"Control shift...how do I select it completely?" - S2
	Unreceptive	"I've never done anything in JMP." - L10
Time Pressure Criteria		
Time Pressure	Definition	Participant makes statement regarding time to complete tasks
	Yes	[Python script] "Yeah, that would work, if we had time." - L5
	No	No comments about time
Tool Observability Criteria		
Observability	Definition	The ability to view a tool through a GUI
	Observable	"Let's deploy a histogram...Insert, Recommended Charts..." - S7
	Non-Observable	"Control-Shift-End" - S1

Table 2: `interactions` Study Results

	Effective		Ineffective		Unknown	
	n	%	*n*	%	*n*	%
Politeness						
Polite	14	52%	5	19%	8	30%
Neutral	52	50%	27	26%	25	24%
Impolite	5	45%	3	27%	3	27%
Persuasiveness						
Persuasive	5	36%	4	29%	5	36%
Unpersuasive	66	52%	31	24%	31	24%
*Receptiveness**						
Receptive	39	61%	9	14%	16	25%
Neutral	27	48%	14	25%	15	27%
Unreceptive	5	23%	12	55%	5	23%
Time Pressure						
Yes	7	37%	7	37%	5	26%
No	64	52%	28	23%	31	25%
Tool Observability						
Observable	57	50%	30	26%	28	24%
Non-Observable	14	52%	5	19%	8	30%
Recommendation Type						
Peer Observation	16	30%	5	9%	32	60%
Peer Recommendation	55	62%	30	34%	4	5%

3.2 [sorry] "Sorry to Bother You: Designing Bots for Effective Recommendations" (Completed, Spring 2019)

Motivation: While recommendations between humans are effective for tool discovery, they are not always the most practical way to increase awareness of useful development tools. For example, Murphy-Hill and colleagues also discovered that peer interactions occur infrequently in the workplace [86]. Furthermore, in *Alone Together* Turkle describes how technology has negatively impacted communication and become a substitute for face-to-face interactions [109]. As development teams become larger and more distributed, effective automated recommendations are necessary to improve tool adoption among software engineers. Research shows bots are useful for automating tasks and improving user effectiveness and efficiency [104]. However, they can also be inconvenient and frustrating during interactions with humans. To better understand the impact of bots and identify a baseline for automated recommendations, we created *tool-recommender-bot* to make development tool recommendations to software engineers on GitHub using a naive *telemarketer design*. In this study, we examined the effectiveness of recommendations from *tool-recommender-bot* and gathered feedback from developers who received a suggestion to better understand user reactions to receiving naive automated recommendations and set the groundwork for designing better solutions in future approaches.

naive *telemarketer design*: To evaluate a basic approach for making automated recommendations, we developed the naive *telemarketer design*. This design behaves similar to a telemarketer in that it "calls" users to deliver a static message that never deviates from the script and lacks the social context necessary to adjust messages, customize recommendations, or respond to questions and feedback. naive *telemarketer design* sends developers a generic message with information about a static analysis tool, displays a generic example featuring a code snippet based on a common programming error, and provides sample output from the tool. This is not the best approach for making recommendations, however we implemented this simple design to better understand how bots influence human behavior and how developers respond to automated recommendations. With this naive design, we developed a simple bot to evaluate and identify a baseline for making automated recommendations for developers. We use our results and feedback from developers to motivate integrating concepts from nudge theory to improve future automated recommendation designs.

Methodology:

Data Collection. Our evaluation sought to determine the effectiveness of our naive *telemarketer design* recommendation approach to developers working on real-world software applications. We randomly sampled public open source software repositories on GitHub used in the evaluation for Repairnator[16] [110], an automated program repair bot [110]. The projects selected for this study had to meet the following criteria:

- written in Java 8 or higher,
- successfully validate and compile with Maven,
- do not already include Error Prone in the build configuration

Based on these criteria, we identified 52 projects for our experiment that received an automated pull request recommendation. The list of projects for this evaluation is available online[17].

Implementing the naive telemarketer design. To evaluate our naive *telemarketer design* approach, we implemented *tool-recommender-bot* to make basic tool recommendations to GitHub developers. *tool-recommender-bot* integrates this simple approach by making generic recommendations as automated pull requests on repositories. On GitHub, pull requests are the preferred method to propose changes to repositories[18]. Automated pull requests have also been used by bots in related work, for example by Mirhosseini and colleagues to encourage GitHub developers to upgrade out-of-date dependencies for repositories [77]. Figure 2

[16] `https://github.com/Spirals-Team/repairnator/blob/master/resources/data/results-buildtool.csv`
[17] `https://go.ncsu.edu/botse-projects`
[18] `https://help.github.com/articles/about-pull-requests/`

presents a screenshot of a recommendation from our system for this study. In this experiment, *tool-recommender-bot* recommendations provided basic information about the Java static analysis tool Error Prone (Fig. 2.A). It also presents a simple coding error in Java, using "`==`" to evaluate string equality instead of the `String.equals()` method (Fig. 2.B1), and the corresponding output from Error Prone reporting a `StringEquality` error[19] (Fig. 2.B2). To make recommendations, *tool-recommender-bot* automatically adds the Error Prone plugin to Maven[20], to Project Object Model (*pom.xml*) configuration files and created automated pull requests with the changes. An example pull request from our system using the naive *telemarketer design* can be found here.[21]

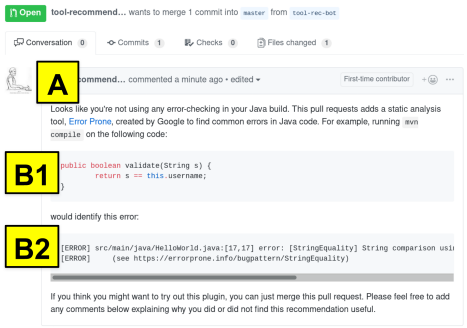


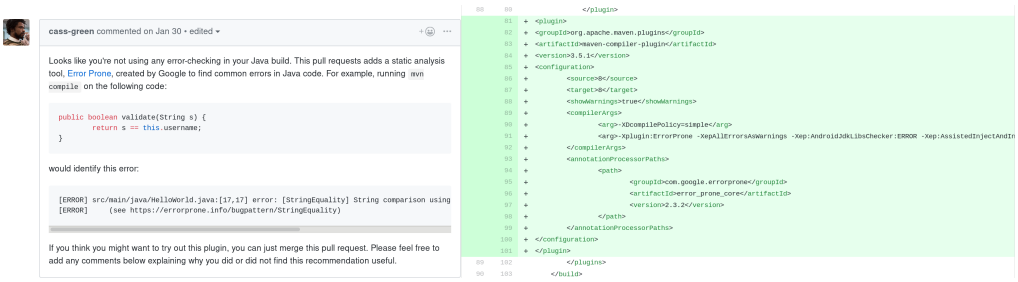
Fig. 2: Example naive *telemarketer design* recommendation

Determining the effectiveness of recommendations. To measure the effectiveness of naive *telemarketer design*, we observed the status of automated pull requests from *tool-recommender-bot*. A merged automated pull request from *tool-recommender-bot* indicates an effective recommendation because the developer showed a willingness to try Error Prone and integrate the static analysis tool into the build for their repository by merging our changes into their code base. A closed or ignored pull request left open from our system indicates an ineffective recommendation because the developers did not attempt to integrate the tool into their projects. We observed the automated pull requests for one week to categorize the recommendations. The rate of effectiveness was calculated by measuring the percentage of merged pull requests out of the total sent.

Additionally, we encouraged developers to provide feedback on pull requests by asking them to "Please feel free to add any comments below explaining why you did or did not find this recommendation useful". This was done to gather qualitative data on how developers reacted to receiving naive *telemarketer design* recommendations from *tool-recommender-bot*. We aggregated and analyzed

[19] `http://errorprone.info/bugpattern/StringEquality`
[20] `http://maven.apache.org`
[21] `https://github.com/CSC-326/JSPDemo/pull/2`

the comments made by GitHub developers on our automated recommendations. Figure 3 presents a screenshot of example automated pull requests used in the evaluation for this study.



(a) Pull request recommendation text (b) Pull request diff updating a *pom.xml* file

Fig. 3: Example automated pull request from the `sorry` study

Results: We found that bots with basic approaches are not effective for influencing human behavior. Table 3 presents our findings from the evaluation. In our study, naive *telemarketer design* only made two successful recommendations out of 52 (4%). We also observed 10 closed pull requests and 40 recommendations with no response from developers. We received 18 comment responses from developers, most of which were negative feedback. Five of the comments were related to improper formatting of the *pom.xml* file when adding the Error Prone plugin, and eight were related to our automated pull requests breaking builds for projects. Based on this feedback, we discovered the main drawbacks to the naive *telemarketer design* were a lack of ***social context*** and interfering with ***developer workflow***. To provide implications for future automated recommender systems, we propose integrating concepts from nudge theory to effectively incorporate automated suggestions for developer behaviors in software engineers' social context and development workflow. The results of this research were published and presented at the 1st International Workshop on Bots in Software Engineering[22] (BotSE 2019) at the International Conference on Software Engineering (ICSE) [27].

[22] `https://botse.github.io/`

	n	**Percent**
Merged	2	4%
Closed	10	19%
No Response	40	77%

Table 3: `sorry` Study Results

4 Developer Recommendation Preconditions

Based on the results from these preliminary studies, we uncovered four concepts for making effective developer recommendations. At the minimum, these developer recommendation preconditions are required to make successful recommendations for software engineers. Below we define each of these concepts and provide examples with data collected from the completed evaluations, other software engineering research, and nudge theory literature:

4.1 Demonstrate Desire

In the `interactions` study, we found that participants expressing eagerness to use a recommended tool led to more effective recommendations. For example, during one study session a participant suggested using multi-level sorting functionality in Excel. Their partner, L11, demonstrated a desire to use the tool by responding "Oh! Add level! Yes, awesome!" and adopted multi-level sorting for the remainder of the study tasks. This suggests recommending desirable tools and behaviors can increase adoption among developers. Meanwhile, in another case a participant asked their partner if they wanted to use the R statistical computing software[23] to complete a task, but their partner responds "No, no, no..." (S14). This shows how a lack of desire to use a specific tool can negatively impact the outcome of a recommendation.

Software engineering research also suggests desire impacts the activities and behavior of developers. For example, Senyard and colleagues suggest that the desire of developers is important for motivating programmers to contribute to free and open source software and maintain successful projects [96]. Furthermore, Murphy-Hill and colleagues found that one barrier to peer interactions is *developer inertia*. This refers to when programmers do *not* desire to share or learn about new software engineering tools because they "feel that they do not need to discover a new tool because existing tools will do the job" [85, p. 16]. These are examples of how the desire of software engineers can impact their behavior and decision-making when completing programming tasks.

Behavioral science research shows that humans often make poor decisions based on their desires. This is due to the fact that sometimes damaging behaviors provide short term benefits but have long term costs. Examples of these

[23] `https://www.r-project.org/`

activities include smoking and eating junk food. Sunstein and Thaler argue humans need decision-making help in situations when "choices and their consequences are separated in time...we get the pleasure now and suffer the consequences later...[these] are prime candidates for nudges" [105, p. 75]. Nudges can be used to encourage the adoption of better behaviors by raising awareness of consequences for decisions. For example, nudges to teenagers in Montana were able decrease smoking rates among students by educating them on the dangers of smoking and providing information on the behaviors of peers [69]. In some cases, poor developer behaviors may provide benefits to development teams. For example, Xiao and colleagues found that developers avoid adopting security tools that automatically check for vulnerabilities in code to save time and costs for training and implementing new systems [121]. While avoiding useful developer behaviors may provide some benefits such as saving time and money, ignoring these practices can ultimately have serious consequences for development teams and their products over time. Nudges such as providing improved and more relevant feedback to users are an effective way to persuade developers to adopt better behaviors they may find undesirable by providing information and insight into the long term costs of avoiding these actions.

4.2 Familiarity

Another key takeaway from the `interactions` study is that users are more likely to adopt recommendations for tools and concepts they are familiar with it. In this study, we defined this criteria as users explicitly expressing familiarity with the environment surrounding a recommended tool. An example of an familiarity impacting the outcome of a recommendation in our experiment arose when L8 asked about using the `COUNTIF` function in Excel. L7 was familiar with the function and used the tool replying "Yeah...here we go". However, we also found unfamiliarity negatively influenced adoption when a participant recommended using R to create a plot for analyzing the data, but their partner responded "I dont know R" (S9). In this case, the participant's unfamiliarity with R led to an ineffective recommendation.

Familiarity can also lead to *developer inertia*, where programmers prefer to stick with their familiar tools and workflow and avoid adopting better behaviors. Prior research also suggests familiarity can impact effectiveness and productivity at work. Goodman and colleagues found that the more knowledge employees have about the workplace and environment, or *work familiarity*, improves performance [47]. Similarly in software engineering, Espinosa and colleagues found that familiarity impacts the completion of development tasks for distributed development teams [40] while de Alwis and colleagues found that unfamiliarity in the Eclipse IDE made developers feel disoriented and negatively impacted productivity [35]. For improving development tool adoption, Murphy-Hill and colleagues propose integrating familiarity into recommender systems by ranking commands based on similarity with collaborative filtering [84]. Finally, Ko and colleagues suggest the majority of developers' time is spent learning about and becoming familiar with unfamiliar code, and code comprehension can impact

other development activities and behaviors such as code navigation, searching, and tool usage [62]. This indicates that familiarity plays a role in developer behavior and impacts their adoption of useful practices.

Research suggests humans are prone to make decisions based on previous experiences and rarely select options that are unfamiliar. Thaler and Sunstein note "it is particularly hard for people to make good decisions when they have trouble translating the choices they face into the experiences they will have...when people have a hard time predicting how their choices will end up affecting their lives...a nudge might be welcomed" [105, p. 77-78]. One example of an unfamiliar problem for many families face is selecting student loans. To account for unfamiliarity in student loan selections, Bettinger and colleagues implemented a nudge to incorporate the Free Application for Federal Student Aid (FAFSA) for student financial aid into the HR Block[24] tax return software, which many are more familiar with and utilize annually to complete their taxes. They found that this nudge made it easier to compare loan options and increased college enrollment among high school seniors [20]. Many software engineers are comfortable with their current toolsets and environments, which leads to an unwillingness to try new useful development tools and processes. Nudges such as providing more information to software engineers when recommending unfamiliar development tools can help increase awareness and inform developers of new and better systems for completing programming tasks over their familiar methods.

4.3 Social Context

In the results from the `sorry` study, we found that one major issue with our naive *telemarketer design* design is its lack of social context. Social context refers to the standard practices and activities necessary to participate in software engineering by interacting with developers and contributing to projects, specifically in open source software. In the `sorry` evaluation, many developers complained that *tool-recommender-bot* did not adhere to formatting guidelines when automatically adding the Error Prone plugin to project *pom.xml* files. The most common social context complaint we received from GitHub users was that *tool-recommender-bot* messed up the whitespace of their project's maven *pom.xml* files when adding the Error Prone plugin (See Figure 3b). One developer replied "The automated tool you use messed up the pom.xml formatting to an extent that I could not see it" (P5). This suggests our bot's inability to adjust to the social context surrounding software development negatively impacted developers' likelihood to adopt recommendations.

Prior work also shows that social context is important in making recommendations to software engineers. Ahmadi goes as far as to argue that software engineering is a social activity [2]. To make effective recommendations, studies show it is important to integrate into this social context. For example, Wessel and colleagues evaluated the usage of bots in open source software and found that their

[24] `https://www.hrblock.com`

inability to integrate into contributors' social context due to limited decision-making and poor feedback was the biggest challenge developers face interacting with bots and desired better interactions with users [114]. Furthermore, prior work found that bots emulating humans receive better responses from developers and are more effective than recognizable bot accounts [80]. Thus, effectively integrating recommendations with the social context of software engineering can impact developer adoption and perception of tools.

In *Nudge*, the authors note "one of the most effective ways to nudge (for good or evil) is via social influence" [105, p. 54]. To study the impact of social nudges, Schultz and colleagues conducted a study that provided feedback to homeowners about the energy usage rates for their neighbors and households in a neighborhood in San Marcos, CA. They found that this nudge was able to drastically decrease usage and improve consumption decisions among users [95]. Social recommendations are popular within software engineering. For example, badges are an effective way to present information about the status and condition of public projects, which can impact the behavior of contributors and other users [108]. In turn, we suggest nudges are an effective method to integrate recommendations for software engineers to adopt useful developer behaviors into the social environment of software engineering.

4.4 Developer Workflow

Another key result from `sorry` was that the naive *telemarketer design* in *tool-recommender-bot* disrupted the workflow of developers, or the processes required to complete programming tasks and deliver software. The most notable example of this was the fact that our automated pull requests for Error Prone often broke builds for repositories. Many projects have adopted continuous integration systems, such as TravisCI,[25] to automatically build and integrate changes into projects. However, modifying projects to add a new static analysis tool often introduced many new errors and caused the build to fail. An example of this from our evaluation can be seen in Figure 4. Out of the 52 pull requests made, at least 17 resulted in a broken build. Many developers complained about this in their feedback on *tool-recommender-bot*, including P18 who commented "This PR failed automatic checks, I think it should be closed". This interruption of developer workflow often discouraged users from merging pull requests from our system and accepting the recommendation.

Other work in software engineering also notes the importance of integration into the workflow of developers. For example, Sadowski found that one of the primary reasons developers at Google ignore static analysis tool warnings is because they are not integrated into their workflow [93]. Additionally, Viriyakattiyaporn and colleagues found that the inability to deliver suggestions within appropriate workflows discouraged programmers from following recommendations to improve code navigation with Spyglass [111] while Johnson and colleagues discovered that software engineers reported avoiding static analysis tools in their work due to

[25] `https://travis-ci.org/`

a lack of customizability and poor integration into their existing processes [58]. This points to a need to integrate recommendations to developer within their workflow to increase the adoption of useful behaviors to improve code quality and developer productivity.

Nudges are useful for integrating recommendations into user workflows. For instance, a study conducted on Yale seniors found that a lecture on the risks of tetanus was ineffective (3%) in convincing students to get a shot at the health center. However, providing a campus map to students with the health center circled in the same lecture convinced over nine times more people (28%) to get the shot [67]. Even though the first group of students knew the location of the health center, nudges such as providing more details allowed students to know how to fit a visit into their weekly schedule and normal workflow. Software engineering research shows that integration into developer workflows is vital. To improve on the lack of tool adoption at Google, Sadowski implemented Tricorder to run numerous program analysis tools on code during code reviews [94]. Similarly, Balachandran found that integrating tools into the code review process with Review Bot at VMWare was able to reduce developer effort and improved code quality during code inspections [8]. These studies provide examples of how integration into developer workflow increased adoption of development tools. We believe this concept is key to improving the effectiveness of developer recommendations and encouraging adoption of useful developer behaviors.

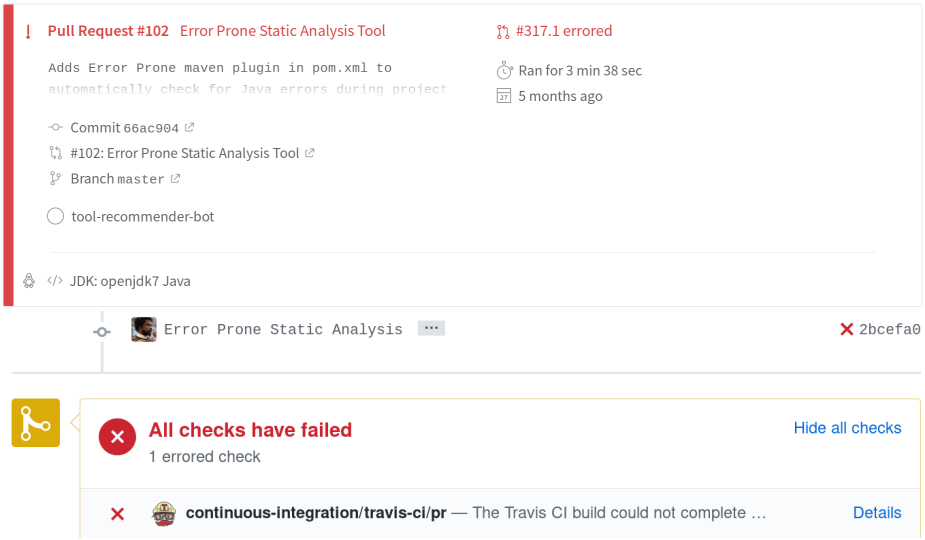


Fig. 4: Examples of automated pull requests from *tool-recommender-bot* causing projects' continuous integration builds to fail

5 Developer Recommendation Choice Architectures

The preliminary results show that receptiveness and development context are required for effective developer recommendations. To improve these recommendations, I present the following ***developer recommendation choice architectures*** to design and frame decisions for software engineers based on nudge theory and software engineering literature: **actionability**, **feedback**, and **locality**. We aim to show that incorporating these choice architectures into developer recommendations can improve the decision-making of software engineers.

5.1 Actionability

Actionability refers to the ease with which users can act on recommendations. In nudge theory, research suggests actionability is a key concept for encouraging humans to make better decisions. A simple nudge is to make target behaviors easy to apply because "many people will take whatever option requires the least effort, or the path of least resistance" [105, p. 85]. For example, one simple nudge to improve the actionability of recommendations is to change the default rule. Madrian and Shea implemented this nudge to encourage employees to enroll in retirement plans. By having users opt-out of 401k plans instead of automatically opting in, they discovered that this improved money saving behaviors and encouraged more employees to join and enroll sooner, with 98% of new employees selecting a plan within 36 months [72]. In this instance, the easiest option to adopt was the default selection. Software engineering research also shows actionability is important to developers. Heckman and colleagues examined the concept of actionability through static analysis notifications in AAITs (actionable alert identification techniques) to help developers identify and resolve defects in code [53]. We propose making automated development tool recommendations actionable will encourage the adoption from developers.

5.2 Feedback

Sunstein and Thaler note that "the best way to help Humans improve their performance is to provide feedback" and "choices can be improved with better and simpler information" [105, p. 92, 204]. For example, most people order familiar and repeated meals at fast food restaurants, however nudges such as providing information on the amount of calories in food and feedback on recommended daily caloric intake encouraged consumers to purchase unfamiliar and healthier meals [119]. In this case, feedback refers to information provided to impact developer behavior. Software engineering researchers also show that feedback to developers is important. For instance, Barik and colleagues examined the impact of compiler error message feedback on how developers resolved problems [10]. Furthermore, Cerezo and colleagues also suggest that user-driven communication can improve the perception of chatbots as opposed to single-purpose bot-driven techniques [30]. To improve the effectiveness of automated recommendations to software engineers, we believe providing useful information and feedback will improve the likelihood developers adopt useful behaviors.

5.3 Locality

Locality refers to the setting of recommendations in the context of developers completing programming tasks. To describe the locality of developer recommendations, we divide this concept into two subcategories: *spatial* and *temporal* locality.

Spatial: Spatial locality refers to the location where recommendations are made. Nudge theory suggests that the location of recommendations matters when encouraging people to adopt useful behaviors. For example, Hanks found that changing the location of vegetables, fruits, etc. in a high school cafeteria increased the purchase and consumption of healthier foods by students [51]. Research shows that the location of recommendations matters to developers and they prefer notifications are placed in convenient locations. Johnson et al. reported that developers felt inconvenienced leaving their normal coding environment to use development tools [58]. Similarly, de Alwis and colleagues found that the inability to locate navigation and displays made developers feel disoriented in the Eclipse IDE [35]. In our prior work, we developed an Eclipse code navigation plugin, FLOWER, which was developed with *in situ* navigation design principles to avoid developer disorientation and switching between views. In our evaluation of this tool, we found that the location of suggestions within the IDE and led to increased efficiency with branchless navigation and positive responses from participants on the user interface [102]. In designing choice architectures for recommendations to developers, we propose making automated suggestions to developers in familiar and convenient locations to target user receptivity and encourage adoption.

Temporal: Temporal locality refers to the timing of when recommendations are made to users. In nudge theory, timing plays a major role in impacting human decision-making. For example, an effective nudge for farmers in Kenya was to change the time of year for fertilizer discounts. This encouraged them to make purchases earlier in time to improve the harvest of crops [39]. Software engineering research also shows that behavioral recommendation timing is important in software engineering. For example, Distefano examined configuring static analysis tools to run at *diff time*, or on patches submitted by developers to review before merging into the code base, and found that this increased the fix rate of reported bugs up to 70% compared to nearly 0% for times outside the development workflow, such as assigning bug lists to developers from overnight builds [38]. To incorporate nudges into software engineering recommendations and choice architecture designs, we propose developing automated systems that make timely suggestions to programmers within their workflow to increase their desire to adopt useful behaviors and practices.

6 Experiments and Evaluations

This section describes in progress and proposed studies to support this thesis. The bracketed text represents short names for each experiment and the text in parentheses displays the status and semester of submission.

6.1 [suggestions] "Understanding the Impact of GitHub *Suggested Changes* on Recommendations Between Developers" (In Submission, Fall 2019)

Motivation: GitHub recently introduced a new feature, *suggested changes*,[26] which fosters peer interactions online by allowing developers to recommend code changes to each other during code reviews through this novel system. We chose to analyze this novel feature because it has become very popular on GitHub, with developers "quick to adopt suggested changes" into their workflow and totaled over 100,000 uses within weeks of the initial release, accounting for approximately 4% of pull request review comments and 10% of code reviewers.[27] Additionally, the GitHub suggested changes feature can be considered an example of a digital nudge and adheres to the developer recommendation choice architectures. This research seeks to discover the impact of the recently introduced suggested changes feature by gathering data about the developer usage and effectiveness of these suggestions, collecting feedback from developers about the new feature, and exploring how well the design of this feature generalizes to other types of recommendations. To our knowledge, this is the first study to analyze the suggested changes feature on GitHub. The results from this work provide implications to improve the effectiveness of recommendations to developers and are used to motivate the design of future automated recommender systems.

Suggested Changes: GitHub released suggested changes as a public beta feature in October 2018. This new feature to allow GitHub users to recommend code changes on pull requests to software developers. Figures 5a-c present how the suggested changes feature works. When a reviewer notices a line of code that can be improved, they can click on the plus (+) sign on the line of code in question to write a comment and create a suggestion. Then, the text box in Figure 5a pops up for the users to enter their proposed change. Figure 5b shows a developer typing their suggested code change for the pull request into the text box. Once the reviewer is finished with their suggestion, they can click on the "Start a review" button to submit the suggested change. Finally, the developer who created the pull request can see the suggested change on their code, shown in Figure 5c, and can commit, edit, or ignore the proposed modifications. Clicking "Commit changes" will automatically add the change to the pull request as

[26] `https://help.github.com/articles/incorporating-feedback-in-your-pull-request/#applying-a-suggested-change`

[27] `https://github.blog/2018-11-01-suggested-changes-update/`

a new commit. Suggested changes can be considered a nudge because they are used to encourage developers to improve code in their pull requests without providing incentives for committing suggestions or prohibiting alternative changes to improve the code. Additionally, this feature also implements our developer recommendation choice architectures presented in Section 5: they are *actionable* by allowing developers to immediately apply recommendations by clicking on a button to commit (Figure 5c); provide specific *feedback* to users by providing an improvement to the code with an optional comment (Figure 5b); have high *spatial locality* with recommendations appearing to developers on the exact line of code in their pull request; and have convenient *temporal locality* during code reviews before merging the code.

Research Questions:

RQ1 What suggestions do developers make with suggested changes?
RQ2 How effective is the suggested changes feature on GitHub?
RQ3 How useful is the suggested changes feature for developers?
RQ4 How well does the suggested changes feature generalize to other types of recommendations?

Proposed Methodology: For this research, we plan to conduct a multi-methodology study divided into two phases to gather and analyze data to answer each of our research questions.

Phase 1: *An Empirical Study on GitHub Suggested Changes*

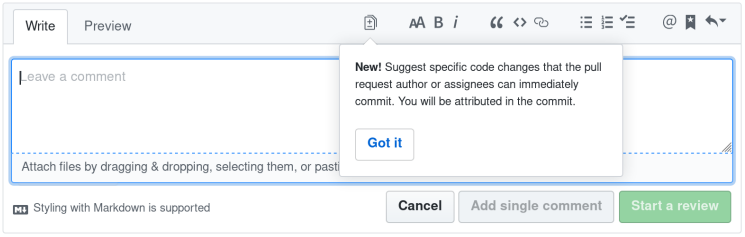
The first phase of this work explores the usage and effectiveness of suggested changes on GitHub to answer the first two research questions.

Data Collection. To collect suggested changes to classify for RQ1, we developed a script to programmatically search for instances of the `suggestions` tag in pull request comments (See Figure 5b). The `suggestions` tag indicates that reviewer used this feature to propose a suggestion on a pull request. To gather projects and pull requests to analyze, we used the GitHub API to sort repositories by the most recently updated pull requests to compile a list of pull request with suggested changes. The script for automatically detecting uses of the suggested changes feature on GitHub pull requests is publicly available online.[28]

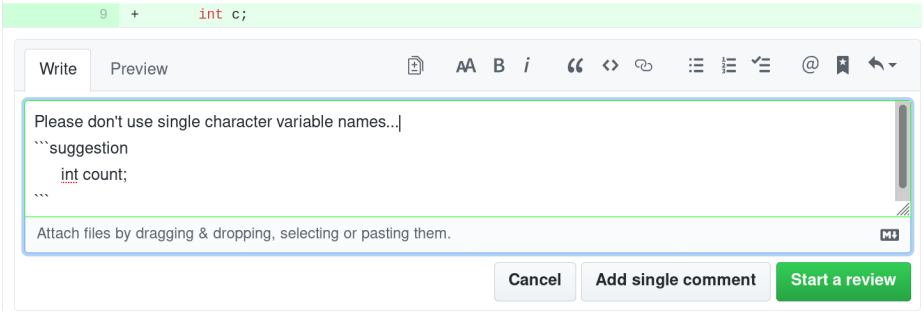
To explore the effectiveness suggested changes, pull requests, and issues for recommendations to developers on GitHub, we mined GitHub repositories to analyze these systems. We collected pull requests and issues on the top-forked projects that have PRs with suggested changes. We analyzed the repositories with the most forks because GitHub recommends forking projects to create pull

[28] `https://github.com/chbrown13/suggestions`

(a) Reviewer adds a pull request comment



(b) Reviewer suggests a code change



(c) Developer applies suggestion from reviewer

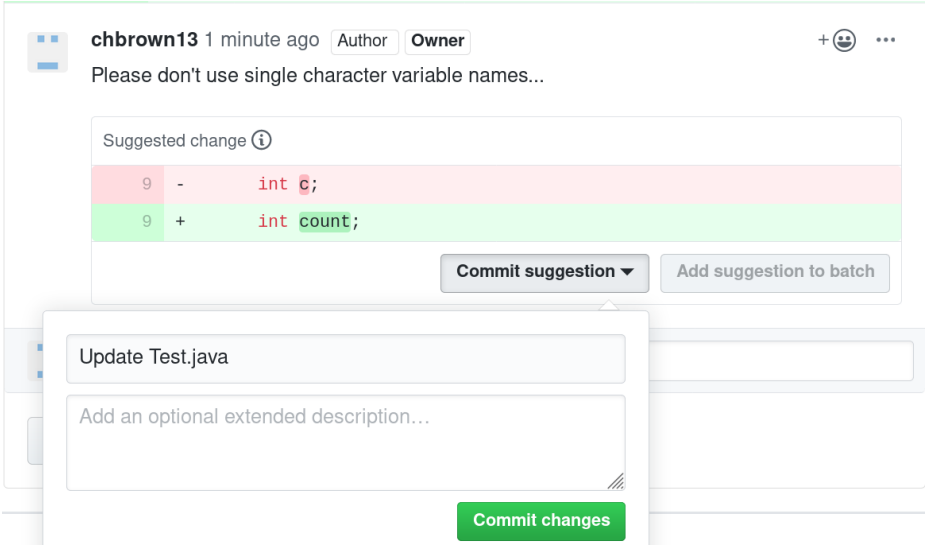


Fig. 5: GitHub Suggested Changes Example

requests,[29] and suggested changes require PRs to make recommendations. Additionally, our we limited our dataset to activity after October 2018 when the suggested changes feature was introduced. To answer RQ2, we analyzed a total of 3683 suggested changes from 11869 pull request review comments, 3882 pull requests, and 3516 issues. We used two metrics to measure the effectiveness of each system: *acceptance* and *timing*. A list of projects used for this evaluation is available in Table 4.

Classifying types of suggested changes. To categorize the types of changes developers suggest with this feature, we randomly sampled 100 recently updated pull requests with an instance of the `suggestions` tag in the comments. A random sample was used to avoid bias from classifying suggested changes from the same GitHub users and projects. To identify categories of suggested changes, two researchers performed an *open* coding by analyzing pull requests review comments with the `suggestions` tag and code changes recommended by developers with this feature (inter-rater agreement = 71%, Cohen's $\kappa = 0.5942$). The two coders then came together to discuss their results come to an agreement. Below we define and provide examples of the four identified categories for suggested changes:

Corrective: The corrective category refers to using suggested changes to fix issues found in the code. Software engineering research shows corrective changes are important for improving code quality. For example, Bacchelli and colleagues found that finding defects is the primary motivation for software engineers to conduct code reviews [7]. For example, Figure 6a presents a corrective suggested change from a reviewer on a developer's pull request. The suggestee referred to a variable as a global variable instead of a class variable, and the suggester proposes a fix by adding the `self` keyword.[30]

Improvement: Improvement suggested changes refer to when reviewers recommend code changes to refactor or optimize a contributor's code. Developers at Microsoft reported that code improvements are the primary benefit of code reviews.[31] Additionally, further analysis by Bacchelli and colleagues revealed that, while developers reported correcting defects as the primary motivation for code reviews, code improvements were the most frequently mentioned motivation [7]. Figure 6b presents an instance of an improvement suggested change, where the suggester proposes improving the readability of the suggestee's code by renaming a variable from `x` to `manifest`.[32]

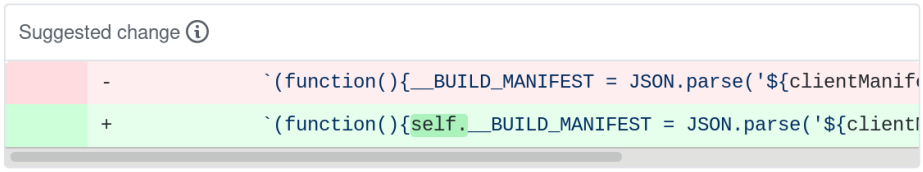
Formatting: The formatting category refers to refactoring code changes that impact the style and presentation of the code. Fixing formatting issues is also an important change to improve code. Bacchelli and colleagues reported developers also found code reviews to be useful for ensuring code styles and standards are consistent between programmers on development teams [7]. An example of a for-

[29] `https://help.github.com/en/articles/fork-a-repo`
[30] `https://github.com/zeit/next.js/pull/7696#discussion_r302333269`
[31] `https://www.michaelagreiler.com/code-reviews-at-microsoft-how-to-code-review-at-a-large-software-company/`
[32] `https://github.com/gatsbyjs/gatsby/pull/13471#discussion_r277948539`

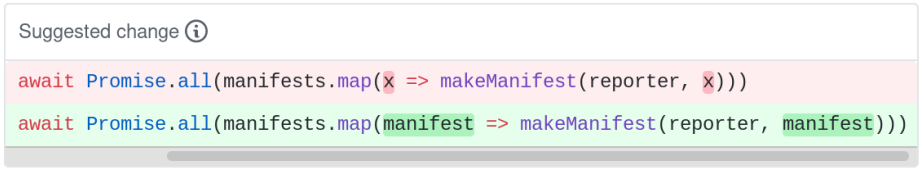
matting suggestion is presented in Figure 6c.[33] where the suggester recommends changes to fix spacing issues in the suggester's code that violate the Python PEP8 whitespace standards.[34]

Non-Functional: Non-functional suggested changes refer to modifications reviewers recommend outside of the code. This includes suggestions to fix spelling and grammar issues or reword phrases in code documentation and comments. Non-functional changes are prevalent in software engineering, for example Beller and colleagues found that documentation changes are the most frequent type of fixes applied during code reviews for open source software [17]. Figure 6d presents an example of a non-functional suggested change. In this case, the suggester discovers a typo where the suggestee misspelled *deserialize* in a documentation files and uses the suggested changes feature to recommend a fix for the error.[35]

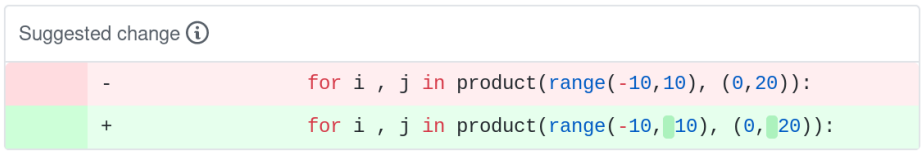
(a) Corrective:



(b) Improvement:



(c) Formatting:



(d) Non-Functional:

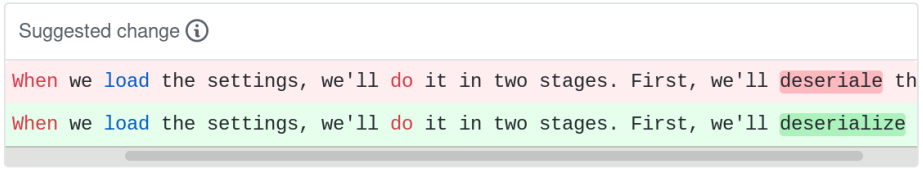


Fig. 6: Suggested Changes Categories

[33] https://github.com/numba/numba/pull/4204#discussion_r310598073
[34] https://www.python.org/dev/peps/pep-0008/#whitespace-in-expressions-and-statements
[35] https://github.com/microsoft/terminal/pull/1258#discussion_r293932790

Determining acceptance of recommendations. We define acceptance as the process of users welcoming suggested changes recommended by another developer. Accepting these suggestions indicates an effective recommendation because the author trusts changes and believes it will be beneficial to the code. Research shows this concept is important in software development. For example, to emphasize the importance of acceptance in software engineering, Middleton and colleagues argue receiving code contributions from outside developers is essential for the maintenance and evolution of open source software [76].

Since the suggested changes feature is currently not supported by the GitHub API,[36] we extended the suggested changes detection script to examine commits made on pull requests with suggestions. To determine whether suggestions were accepted or rejected, our script found instances of pull requests with suggested changes. Then, it parsed review comments on the PR to extract the recommended code modifications between the `suggestions` tag and the ending `'''`. Finally, it checked whether the suggested line of code was present in another commit after the comment was made to the pull requests on the same file that received the suggested change. Suggested changes with lines of code that were found to be integrated in a subsequent commit on a pull request were considered accepted, otherwise they were regarded as rejected.

To determine the acceptance of pull requests, we simply used the existing GitHub status. In the pull-based software development model, accepted changes must be merged into the source code [49]. In this case, a Merged PR is considered accepted because the recommended code changes were reviewed and approved by a maintainer to be integrated into the project. For issues, the only possible statuses are Open or Closed. Thus, we were not able to automatically detect if recommendations from this system were closed to be accepted into repositories or closed to be ignored. To analyze only accepted issues with recommendations, we filtered out issues with GitHub labels `bug` and `duplicate` to avoid bug reports and multiple instances of the same issue.[37] After automatically filtering these labels, a researcher manually examined the title, description, discussion, and status of issues to code issues based on two criteria: 1) if they contain a *recommendation* or *no recommendation* and 2) if the recommendation was *accepted* or *not accepted* into the project. Accepted issues are those with recommendations that are integrated with a pull request or commit to the repository.

Determining timing of recommendations. The second metric used to determine the effectiveness of recommendations was the amount of time developers took to accept a suggestion, or acceptance time, for each system. In behavioral economics, Kocher and colleagues suggest the time to make decisions is important to the quality of choices because "time *is* money" [63]. In software engineering, research also shows that time can impact the cost and effort required to fix bugs in code [65]. To measure timing for each system, we calculated the amount of

[36] `https://github.community/t5/GitHub-API-Development-and/Accessing-the-new-quot-GitHub-Suggestions-quot-via-API-public/td-p/13922`

[37] `https://help.github.com/en/articles/about-labels`

time between the creation of the recommendation and its acceptance into the repository.

For suggested changes, we measured the acceptance time as the amount of time between a reviewer commenting on a pull request with the `suggestions` tag until the time a subsequent commit adding the suggested line of code was created on the pull request. To measure the acceptance time for pull requests, we used the GitHub API to calculate the difference between the time from when a GitHub developer creates a pull request and when the pull request is merged into the repository by a project maintainer. After manually inspecting issues to determine they have an accepted recommendation, we used the GitHub API to determine the acceptance time for these issues by calculating the difference between the time the issue was created and the time it was closed.

Project	**Language**	**Forks**	**Suggestions**	**PRs**	**Issues**
qmk/qmk_firmware	C	8723	3627	1997	290
h5bp/Front-end-Developer-Interview-Questions	HTML	8325	1	35	5
Azure/azure-quickstart-templates	PowerShell	7743	2	921	147
firebase/quickstart-android	Java	5603	2	91	124
mavlink/qgroundcontrol	C++	1584	4	402	267
qgis/QGIS	C++	1516	47	436	2683

Table 4: RQ2 Study Projects

Phase 2: ***Developer Feedback on Suggested Changes*** The second phase consisted of a survey and user study to answer the last two research questions on the usefulness and generalizability of GitHub suggested changes.

Data Collection. To determine the usefulness of suggested changes, we surveyed developers who interacted with the feature on GitHub. Surveys were emailed to users with publicly available email addresses who either received or made a suggestion on a pull request within the last six months. Our survey asked users how useful they found the this feature using a 5-point Likert scale as well as free response questions to provide details on what specifically they find useful or not useful about the system and how it is integrated it into their project.

To answer RQ4, we conducted a user study to examine applying the suggested changes feature to tool recommendations. We recruited 14 professional software developers, presented in Table 5, to participate in this study. The participants averaged 5 years of industry experience in various roles such as Software Engineer, Software Developer, Quality Engineer, Consultant, Data Migration Consultant, Support Specialist, User Researcher, and Technical Test Lead. Additionally, all participants were at least somewhat familiar with GitHub. We conducted a think aloud study with a semi-structured interview and audio and screen recorded all sessions to collect feedback from developers on how well the suggested changes feature translates to software engineering tool recommendations.

Determining usefulness of suggestions. We emailed surveys to a total of 570 GitHub users who interacted with suggested changes and received 39 responses (7% response rate). Throughout the remainder of this paper, we use the C- prefix to describe a *suggestee*, or a contributor who received a suggested change on their pull request, and the R- prefix to indicate a *suggester*, or a reviewer who made a comment with the suggested changes feature on a pull request. We aggregated the Likert scores to examine the overall usefulness, then two experts *open* coded the open-ended responses from developers on the useful (72%, κ = 0.6828) and unuseful (77%, κ = 0.7125) aspects of suggested changes. The researchers discussed derived categories and came to a consensus on themes found in comments. To resolve disagreements in coding statements, the first author acted as the tie-breaker. The primary inconsistencies in coding came from determining the existence of multiple themes since responses from developers could span more than one category.

Determining the generalizability of recommendations. To determine the impact of suggested changes on tool recommendations, we asked participants to interact with sample recommendations from a suggested change, pull request, issue, and email. Participants were asked to provide a Likert-scale ranking on how likely they would adopt the tool and to discuss what they like and dislike about each system as well as provide insight into what makes tool recommendations effective in general. Figure 7 presents an example of the prototype suggested change recommendation system, using the feature to suggest a fix for a bug reported by static analysis output and recommend a tool for developers to find and prevent errors in the future. Staged recommendations for pull requests, issues, and emails contained similar text suggesting tools to participants from each system. To analyze feedback on this design, we aggregated the Likert scores and open-ended feedback from participants. For the rest of this paper, user study participants are indicated with a P-prefix. We transcribed and analyzed recordings of sessions to present feedback from developers on receiving tool recommendations with suggested changes.

Expected Results: From this evaluation, I expect to gain insight into the GitHub suggested changes feature. For RQ1, we hypothesize that suggested changes are useful for recommending a wide variety of code changes. For RQ2, we hypothesize suggested changes are as effective as other systems for recommendations between users on GitHub. For RQ3, we expect users find this new feature very useful for suggesting and receiving code change recommendations. Finally, for RQ4 we believe the design of this system can generalize to other types of recommendations and is preferred by software developers for recommending static analysis tools. Overall, we expect the results from this study to motivate the use of digital nudges for making effective recommendations to developers. The results of this evaluation were submitted for publication to the 2020 International Conference on Software Engineering (ICSE 2020).[38]

[38] `https://conf.researchr.org/home/icse-2020`

ID	Experience (yrs)	GitHub Familiarity	OSS Contributions	Tool Usage
P1	30	Very Familiar	Occasionally	Very Frequently
P2	Less than 1	Moderately Familiar	Never	Never
P3	Less than 1	Very Familiar	Rarely	Moderately Frequent
P4	8	Very Familiar	Very Frequently	Very Frequently
P5	10	Familiar	Rarely	Moderately Frequent
P6	5	Moderately Familiar	Occasionally	Very Frequently
P7	6	Familiar	Frequently	Very Frequently
P8	6	Familiar	Very Frequently	Very Frequently
P9	Less than 1	Moderately Familiar	Occasionally	Very Frequently
P10	1	Moderately Familiar	Occasionally	Very Frequently
P11	3	Familiar	Very Frequently	Very Frequently
P12	3	Familiar	Rarely	Very Frequently
P13	1	Moderately Familiar	Never	Never
P14	1	Moderately Familiar	Never	Frequently

Table 5: RQ4 User Study Participants

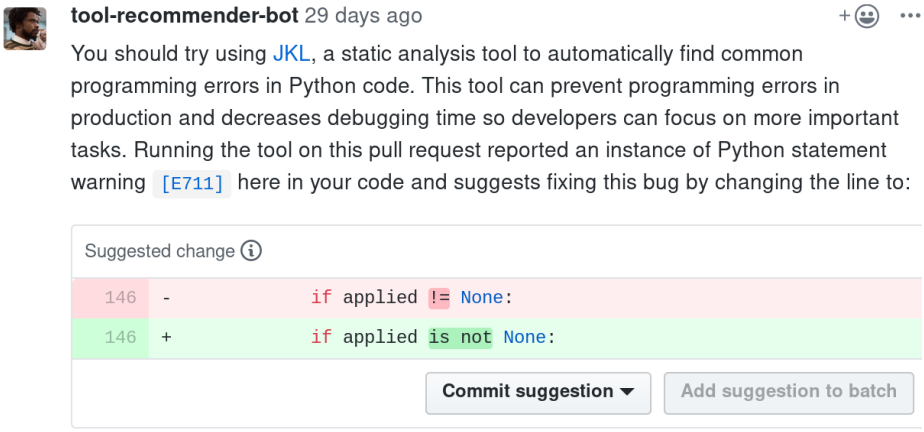


Fig. 7: Mock Recommender system with suggested changes

6.2 [nudge-bot] "Nudging Students Toward Better Software Engineering Behaviors" (Proposed, Spring 2020)

Motivation: To integrate user receptiveness [26] and improve on the effectiveness of a naive bot [27] for making suggestions to improve developer behavior, we propose implementing a new bot to nudge software engineers to adopt developer behaviors. To understand the impact of implementing automated recommendations as digital nudges, we plan to implement and evaluate a novel recommendation approach in a new system: *nudge-bot*. While *tool-recommender-bot* and the naive *telemarketer design* approach failed to effectively recommend static analysis tools to developers, we aim to enhance *nudge-bot* by incorporating our conceptual framework for designing effective developer recommendations in addition to design implications from the results of the `suggestions` study examining an existing system that can be considered a digital nudge. To evaluate this system, we plan to examine how recommendations from *nudge-bot* improve the decision-making and behavior of students in a college-level software engineering course. While research shows that automation is useful to help with grading and teaching in introductory computer science courses [101] and providing feedback to students [55], there is little to no work exploring the use of automated recommendations to improve student behavior. We aim to show that this new system can improve student behavior in software engineering education and the results from this evaluation can provide implications for making effective recommendations to improve the behavior of software engineers in industry.

Software Engineering Education: Research suggests improvements are needed for software engineering education and the future of the software industry. The ACM notes that there is a "crisis" in Computer Science Education that will result in necessary computing-related jobs going unfilled [118]. Furthermore, even though the computer science major is one of the most popular fields of study at universities, it also has a very high attrition and failure rate [14]. For software engineering education, a subset of computer science, researchers have explored problems with effectively preparing students for industry. Jazayeri outlines challenges of teaching software engineering and provides ideas for a new curriculum and skills to educate successful software engineers [57]. Furthermore, Devadiga argues that topics taught in school are often unrelated to industry practice and suggests merging with startups to improve SE education [36]. Additionally, Heckman and colleagues present "the good, the bad, and the ugly" of teaching software engineering to students using an open source software system [52]. Several issues they found include delayed feedback, group dynamics, high workload, and aggressive scheduling. We aim to improve on these problems in software engineering education by using nudges to encourage students to adopt useful developer behaviors.

Research Questions:

RQ1 How do nudges influence software engineering student productivity?
RQ2 How do nudges impact the quality of software engineering student projects?

Proposed Methodology: To evaluate integrating nudge theory into developer recommendations, we plan to conduct a mixed-methods research study.

Data Collection. The data for this evaluation will be collected from the undergraduate (CSC 326) Software Engineering course at North Carolina State University. The final project for the course will require students to maintain and add new features to iTrust[39], an electronic health records system. First, we will analyze data by mining team GitHub repos for final projects from previous semesters of CSC 326. This will help us determine which project milestones to nudge for as well as when to nudge. During this stage, we will collect information from iTrust repositories and past teams such as overall project grade, submission times, total number of commits, amount of time to complete various milestones, and code contributions by each group member. While the project requirements change for the team project each semester, each iteration of the class will have similar milestones to complete every year. Additionally, we plan to conduct to pilot and test *nudge-bot* this semester on a project for the graduate software engineering course (CSC 510).

After examining projects from prior semesters to determine the best behaviors and timing for nudges, we plan to mine repositories for the CSC 326 final project in Spring 2020 to answer our research questions. Participants will be volunteer software engineering groups from the class who opt-in to receiving notifications from a bot and consent to using the data collected for this research. We plan to gather quantitative data based on student behaviors and performance while developing their iTrust project based on recommendations from *nudge-bot*. Additionally, we plan collect qualitative data by surveying students to gather feedback and learn about their perception of nudges from our automated system.

Implementing nudge-bot. To build our new *nudge-bot* recommender system, we utilize the GitHub API to mine repositories and track behaviors of students. This system will implement several different types of interventions to encourage students to adopt better behaviors based on monitored activity. We plan to design our bot to incorporate results from the prior work to address our conceptual framework for making effective recommendations to developers. In our evaluation, *nudge-bot* will send notifications to students encouraging them to adopt better software development behaviors for their project. The input to the system will be a YAML configuration file mapping project milestones with deadlines and containing other information such as project team members and contact information. *nudge-bot* will be deployed on Jenkins servers used for the CSC 326 course. Our goal is for software engineering educators to be able to customize *nudge-bot* and integrate it into their SE courses and projects, and for software engineering researchers to be able to extend this system to recommend useful behaviors to professional developers contributing to software applications. The code for *nudge-bot* will also be open sourced and publicly available online.[40]

[39] `https://github.ncsu.edu/engr-csc326-staff/iTrust2-v5`
[40] `https://github.com/chbrown13/nudge-bot`

Defining student behaviors. To answer RQ1, we will observe the impact of nudges on influence software engineering student project management behaviors. We aim to show that nudging students to adopt useful project management behaviors can improve the quality of their project and development productivity.

Project Management: Poor project management is a problem that negatively impacts software engineering students Beaubouef and colleagues note that poor project management is a hindrance for computer science students and leads to poor performance in programming classes and high attrition among CS majors [14]. Additionally, this is a problem that also applies to professional software engineers in industry. Charrette argues that billions of dollars are wasted every year due to failing software projects, and suggests these project fail because of poor project management factors such as inadequate project goals, inaccurate estimates, ill-defined requirements, poor communication, and sloppy development practices [31]. The increase of distributed development teams in the software engineering industry has also led to additional project management challenges in global software development, including a lack of cultural understanding, communication, managing time differences, knowledge management and transfer in teams, trust, geographical distances, and more [88]. By implementing *nudge-bot* to influence project management behaviors in students, we aim to show that our results can translate to the software engineering industry and improving the behavior of professional developers.

For the CSC326 software engineering final project, students are asked to complete a variety of requirements for the iTrust system. For example, development process requirements for the assignment include adding a project wiki page, using GitHub issues, maintaining a passing build on Jenkins, and using a `development` branch.[41]. Additionally, the project requires various functional requirements to add new functionality to iTrust[42]. For project milestones, we plan to nudge students who have not started on specific tasks with an approaching deadline to encourage them to begin. To measure project management, we will compile a list of the functional requirements implemented and process requirements met by teams by observing GitHub repositories. We hypothesize creating nudges to encourage good project management behaviors for students and will in turn help them complete more requirements as well as adopt good practices to improve the development quality and productivity for their final project.

Nudging student behaviors. Nudges will incorporate the basic content for developer recommendations (*desire*, *familiarity*, *social context*, and *developer workflow*) as well as our developer recommendation choice architectures (*actionability*, *feedback*, and *spatial* and *temporal locality*) to automatically recommend better project management behaviors to students for their final team project. To explore the impact of actionability in developer recommendations, we plan to implement *nudge-bot* to create *active* and *passive* actionable nudges to students.

[41] `https://pages.github.ncsu.edu/engr-csc326-staff/326-course-page/team-project/tp-process`

[42] `https://pages.github.ncsu.edu/engr-csc326-staff/326-course-page/team-project/problem-stmt#required-elements`

Active: Active nudges are actionable automated interventions that partially complete programming tasks for students. We plan to determine if these recommendations will encourage users to adopt better behaviors and finish development tasks. For example, the GitHub Projects[43] is a useful feature for users to organize and track progress of their work using a project board. Tasks are created as cards and moved between columns on the board to indicate the status and track progress for each card. To encourage projects with GitHub Project boards to be active in completing tasks, an example active nudge is to automatically create cards with requirements for students to complete for their team project on their repositories. By doing this, we aim to increase awareness for students about this feature and encourage them to use project boards and cards to improve their project management activities. Overall, we hypothesize active nudges will effectively encourage software engineering students to adopt behaviors to improve project management and collaboration.

Passive: Passive nudges will be automated notifications that encourage students to adopt improved behaviors without impacting the project. These will be actionable visual indicators to recommend developer behaviors that would be useful to improve their project without automatically completing tasks for students. For instance, a passive nudge to encourage students to move tasks on a GitHub Projects board is to create badges to indicate their usage on a repository. Figure 8 presents an example of this, displaying a red badge to show developers are not active on their GitHub Project board (i.e. cards remain in the same column) while a green badge indicates they are active in tracking tasks on their board (i.e. cards exist and are moved between columns consistently). Research shows that badges on have been widely adopted on GitHub and repositories with this feature have a higher quality of code for the signals they indicate [108]. Furthermore, badges fit into the nudge theory framework by not providing incentives or preventing options for developers. Additionally, we plan to create a badge for the overall project health. This metric will indicate the likelihood a project will pass based on various factors, such as the amount of requirements completed and the recency of activities such as commits and PRs to prevent team procrastination. We hypothesize passive nudges such as these will be able to encourage students to adopt useful developer behaviors for managing project tasks and increasing team collaboration.

Fig. 8: Example of Project Health badges to nudge developer behaviors

[43] https://help.github.com/en/github/managing-your-work-on-github/about-project-boards

Table 6 presents several examples of student behaviors with potential nudges that would be implemented in our new system. The complete list of developer behavior nudges for *nudge-bot* will be finalized after collecting data from projects from previous semesters of CSC 326, discussions with the instructors of the course for next semester, and the requirements for the final project in Spring 2020 are officially set. More details and information about the course and milestones to complete for the team project this semester are available online.[44]

Behavior	**Nudge**
Manage development tasks	Badge to indicate project health (*passive*)
Implement feature (i.e. diabetes)	Automated PR with empty class (i.e. Diabetes.java) (*active*)

Table 6: Examples of nudges for software engineering students

Defining student productivity. To answer RQ1, we will measure student productivity using two metrics: the total amount of *time* for teams to complete assignments and the amount of *functional requirements* met for the team project.

Time: We will examine time to discover if nudging developers impacts improves productivity and prevents procrastination on the final project. To measure time, we plan to calculate the overall amount of time needed to complete the project as well as time submitting milestones ahead of deadlines. Professional software engineers are often poor at estimating time to complete projects and fall behind schedule [31]. In SE education, Beaubouef and colleagues note that CS students often procrastinate and delay in completing tasks for their class programming assignments, which leads to lower grades and increased frustration [14]. However, Willman and colleagues also found that higher performing students start and end activities earlier than their peers [116]. To encourage students to start and submit deliverables for the final CSC 326 project, we aim to nudge students to adopt better project management behaviors helping them to be more productive and finish tasks sooner.

Functional Requirements: Another metric for measuring productivity is to observe the number of functional processes, or technical deliverables and functionality added to iTrust for the final project. Prior work explores using function points, or the amount of functionality a system provides to users, as a metric to measure productivity [22] and predict developer effort [74]. Examples of functional requirements for the CSC 326 project this semester include adding new features such as an Oral Glucose Tolerance Test, diabetes diagnoses, and blood sugar data entry.[45] Additionally, we will observe GitHub activity for projects

[44] `https://pages.github.ncsu.edu/engr-csc326-staff/326-course-page/team-project/`

[45] `https://pages.github.ncsu.edu/engr-csc326-staff/326-course-page/team-project/problem-stmt`

including commits, issues, and pull requests to measure productivity. We hypothesize that interventions from *nudge-bot* will encourage students to complete more functional requirements and be more productive on their final projects.

Defining project code quality. To answer RQ2, we will examine code quality by observing two metrics: *grades* and *process requirements*.

Grade: To determine if nudges impact code quality, the first metric we plan to examine is the final project grade. Figas and colleagues note that extrinsic motivators such as incentives and grades are key in motivating software engineering students to perform well on group projects [41]. Additionally, Wilson and Shrock found that performance in early computer science courses can predict success in future CS classes [117]. In the undergraduate software engineering course, the grading rubric for the final project consists of real-world software engineering quality metrics and process requirements, in addition to assessing the quality of the functional requirements implemented by teams on their project.[46] We hypothesize that recommendations from *nudge-bot* will improve overall student grades on their final team project.

Process Requirements: Process requirements refer to tasks that deal with the code and development processes. Karunasekera suggests using team processes, process adherence, and quality to assess student projects to prepare software engineering students for industry careers [61]. Additionally, software engineering industry research shows that individual programming, development team, and organizational processes are vital to the success of software engineering products [33]. Similarly, the CSC 326 final project requires students to follow development processes that impact code quality such as adhering to the NCSU department Java style guide,[47] passing unit tests, at least 70% test coverage, documenting code, conducting code reviews, adding a project wiki page and README, and maintaining passing project builds. We aim to show that both active and passive nudges to software engineering students can increase the amount of technical and team processes accomplished by development teams on their final projects.

Expected Results: We hypothesize that active and passive nudges to software engineering students will improve their behavior while working on teams to complete a development project. Specifically, we believe that sending automated notifications to CSC 326 students from *nudge-bot* will: 1) increase the amount of functional and process requirements implemented for development teams on the iTrust healthcare system, 2) reduce the amount of time students procrastinate and complete work on their projects, and 3) improve the overall software quality and student grades for the team project.

[46] `https://pages.github.ncsu.edu/engr-csc326-staff/326-course-page/team-project/#grade-categories`

[47] `https://pages.github.ncsu.edu/engr-csc116-staff/CSC116-Materials/course-resources/style-guidelines/`

7 Related Work

My research is based on prior work that explores making recommendations to developers and technical approaches to improving software engineer behavior.

7.1 Developer Recommendations

Prior work has investigated different methods for making effective recommendations to software developers. One such method is in-person interactions between with colleagues, which research suggests is the most effective way to make suggestions to software engineers. Murphy-Hill explored seven methods software engineers learn about new development tools and found that *peer interactions* were the most effective [85]. Additionally, Cockburn and Williams suggest that collaboration occurring pair programming is useful for saving time and money in addition to improving developer work satisfaction, design quality, code reviews, problem solving abilities, learning, team building, communication, and project management within an organization [32]. Likewise, Maaleej also analyzed peer debriefings, or discussions between developers, and found that these peer interactions are effective for improving code comprehension [71]. While these studies show recommendations between developers is effective, my work aims to use nudge theory to improve automated developer recommendations.

Researchers have also explored the impact of passive help systems on developer learning. These static systems include Stack Overflow [13], Twitter [99], Hacker News [11], GitHub [34], software documentation [45], and social media in general [15]. Furthermore, prior work has proposed and evaluated methods to improve developer recommendations and help solve the software engineering adoption problem. Examples of these methods include idea gardening [28], automated pull requests [77], continuous screencasting [83], live-coding [21], crowdsourcing [48], explorative and exploitative searching [60], logging activities [73], organization-wide learning [70], shared knowledge bases [112], and gamification [12]. Additionally, software engineering researchers have also suggested using theories from other fields to improve developer recommendations. For example, Fleming and colleagues examined applying *information foraging theory*, the study of how humans search for information, to software engineering and how programmers seek information [43]. Furthermore, Singer explored integrating concepts from *diffusion of innovations*, a sociology theory for explaining how knowledge and ideas spread, to increase tool adoption among software developers [98]. To our knowledge, our work is the first research to integrate nudge theory into developer recommendations to improve the decision-making and behavior of software engineers.

7.2 Recommendation Systems for Software Engineering

Software engineering researchers and toolsmithshave developed and evaluated many active help systems to assist programmers. Robillard and colleagues define and provide examples of RSSEs in the context of software development [92].

Prior work has introduced a variety of tools to make recommendations and help developers complete a wide range of programming tasks. For example, Spyglass improves code navigation [112], ToolBox recommends Unix commands [73], Tricorder suggests static analysis bugs to fix [94], Coronado recommends queries to improve code searching [46], Dhruv suggests developers and artifacts to resolve bug reports [5], and many more. Besides these active help systems, research has also explored the use of software robots, or bots, to automatically make recommendations to developers. For example, David-DM[48] and Greenkeeper[49] are bots designed to recommend dependency updates for developers [78]. Beschastnikh and colleagues also implemented analysis bots to increase the adoption of software engineering research in industry [18].

In addition to developing automated recommender systems, research has also explored ways to improve the overall effectiveness of these systems. Fogg also outlines design principles for creating and designing persuasive technologies to encourage users to adopt target behaviors [44]. Furthermore, McNee and colleagues argue that the accuracy of recommender systems is not sufficient for increasing adoption and suggest developers of recommender systems must implement *user-centric* recommendations focused on user experiences and expectations [75]. Similarly, Konstan and colleagues posit that evaluating user experiences metrics is more important for automated recommender systems than optimizing recommendation algorithms [64]. For RSSEs, Murphy and Murphy-Hill explored the concept of trust in recommender systems for software development and found that trust was more important than precision for software engineers [81]. Our work also seeks to improve the effectiveness of recommender bots by focusing on integrating concepts from nudge theory in recommendations to improve developer decision-making and behavior.

7.3 Nudge Theory in Software

To our knowledge, this is the first work exploring the impact of nudge theory on software developers. However, prior work has examined using digital nudges to influence the behavior of software users. For example, Weinmann and colleagues argue that user interface design can impact user behavior and decision-making in digital choice environments [113]. Acquisti and colleagues explored using digital nudges to improve user privacy and security decisions online [1]. Likewise, Huang and colleagues found that digitally nudging social media users impacted social sharing behavior [56]. Additionally, Gupta and colleagues studied using digital nudge interventions to improve distributed team performance on the Test of Collective Intelligence, an online evaluation to measure the ability of team to collaborate and complete a series of tasks [50]. While these studies show that digital nudges are effective for impacting the behavior of software users, we aim to discover if the nudge theory framework can also improve behavior and decision-making of software developers.

[48] `https://david-dm.org/`
[49] `https://greenkeeper.io/`

8 Research Plan

This section presents the research plan for the completion of the work presented in this proposal for my dissertation. All studies are referred to by their short names given in Sections 3 and 6.

8.1 Completed Projects

I have completed and published the following projects for this research proposal:

`interactions`: VL/HCC 2017 [26]
`sorry`: BotSE 2019 [27]

The preliminary research outlined in this proposal was also presented and published at the Doctoral Symposium for the International Conference on Software Engineering (ICSE 2019) in Montreal, Canada [23]. Additionally, this work has been presented at various industry conferences including QECampX 2018 [24],[50] a Red Hat quality engineering conference, and DevConf 2019 [25],[51] an open source conference for software developers and technologists.

8.2 Upcoming Projects

The following projects are currently under submission or in progress for publication and plan to complete in Spring 2020:

`suggestions`: ICSE 2020*
Dissertation: Summer/Fall 2020*
`nudge-bot`: ICSE SEET 2021*

The GitHub suggested changes study is currently under submission to ICSE 2020. We plan to complete the evaluation for *nudge-bot* next semester and submit our results to the Software Engineering Education and Training track at ICSE 2021. Additionally, I plan to complete the majority of the drafting for my dissertation in Spring 2020.

8.3 Proposed Timeline

Table 7 presents a detailed timeline outlining the completion of the remaining research proposed in this document. This plan does not include completed projects and preliminary work.

[50] `https://www.qecamp.com/`, requires Red Hat employee login
[51] `https://devconf.info/us/2019`

Table 7: Research Timeline

	Sept	**Oct**	**Nov**	**Dec**	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**June**	**July**	**Aug**	**Sept**	**Oct**
`suggestions`														
Data Analysis	■	■	■	■										
Rebuttal			■											
`nudge-bot`														
Development	■	■	■	■	■	■								
Pilot			■	■	■	■								
Data Collection							■	■	■					
Data Analysis								■	■					
Writing				■	■	■	■	■	■	■	■	■	■	■
Submission														■
`Dissertation`														
Writing					■	■	■	■	■	■	■	■	■	
Defense									■	■	■	■	■	

* These milestones depend on acceptance and publication into top software engineering conferences. In case of paper rejection or other schedule delays, additional submission venues include ASE, MSR, VL/HCC, CSCW, RecSys, TSE, and TOSEM, along with other peer-reviewed software engineering conferences and journals.

9 Thesis Contract

My dissertation will consist of the following deliverables for the committee upon the completion of this research:

- ☐ Dissertation chapter on the `interactions` study.
- ☐ Dissertation chapter on the `sorry` study.
- ☐ Dissertation chapter on the `suggestions` study.
- ☐ Dissertation chapter on the *`nudge-bot`* study.

10 Acknowledgements

This material is based on research supported by the National Science Foundation under Grant No. 1714538. I would like to thank my advisor Dr. Chris Parnin (CSC) and committee members Dr. Sarah Heckman (CSC), Dr. Kathryn Stolee, and Dr. Anne McLaughlin (PSY). Additionally, I would like to thank everyone who contributed, participated, and provided feedback on all of the research presented in this proposal.

References

1. Acquisti, A., Adjerid, I., Balebako, R., Brandimarte, L., Cranor, L.F., Komanduri, S., Leon, P.G., Sadeh, N., Schaub, F., Sleeper, M., et al.: Nudges for privacy and security: Understanding and assisting users choices online. ACM Computing Surveys (CSUR) **50**(3), 44 (2017)
2. Ahmadi, N., Jazayeri, M., Lelli, F., Nesic, S.: A survey of social software engineering. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops. pp. 1–12. IEEE (2008)
3. Andreessen, M.: Why software is eating the world. The Wall Street Journal **20**(2011), C2 (2011)
4. Andrews, J., Smith, D.C.: In search of the marketing imagination: Factors affecting the creativity of marketing programs for mature products. Journal of Marketing Research pp. 174–187 (1996)
5. Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., Welty, C.: Supporting online problem-solving communities with the semantic web. In: Proceedings of the 15th international conference on World Wide Web. pp. 575–584. ACM (2006)
6. Ayewah, N., Pugh, W.: The google findbugs fixit. In: Proceedings of the 19th international symposium on Software testing and analysis. pp. 241–252. ACM (2010)
7. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 international conference on software engineering. pp. 712–721. IEEE Press (2013)
8. Balachandran, V.: Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 931–940. IEEE Press (2013)
9. del balzo, G.: Nudging in the UK, in the USA, in Denmark, in Italy: an international comparison of Behavioural Insights Teams. Ph.D. thesis (07 2015)
10. Barik, T., Ford, D., Murphy-Hill, E., Parnin, C.: How should compilers explain problems to developers? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 633–643. ACM (2018)
11. Barik, T., Johnson, B., Murphy-Hill, E.: I heart hacker news: expanding qualitative research findings by analyzing social news websites. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 882–885. ACM (2015)
12. Barik, T., Murphy-Hill, E., Zimmermann, T.: A perspective on blending programming environments and games: Beyond points, badges, and leaderboards. In: Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on. pp. 134–142. IEEE (2016)
13. Barua, A., Thomas, S.W., Hassan, A.E.: What are developers talking about? an analysis of topics and trends in stack overflow. Empirical Software Engineering **19**(3), 619–654 (2014)
14. Beaubouef, T., Mason, J.: Why the high attrition rate for computer science students: some thoughts and observations. ACM SIGCSE Bulletin **37**(2), 103–106 (2005)
15. Begel, A., DeLine, R., Zimmermann, T.: Social media for software engineering. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. pp. 33–38. ACM (2010)

16. Begel, A., Nagappan, N.: Usage and perceptions of agile software development in an industrial context: An exploratory study. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). pp. 255–264. IEEE (2007)
17. Beller, M., Bacchelli, A., Zaidman, A., Juergens, E.: Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the 11th working conference on mining software repositories. pp. 202–211. ACM (2014)
18. Beschastnikh, I., Lungu, M.F., Zhuang, Y.: Accelerating software engineering research adoption with analysis bots. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER). pp. 35–38. IEEE (2017)
19. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM **53**(2), 66–75 (2010)
20. Bettinger, E.P., Long, B.T., Oreopoulos, P.: The fafsa project: Results from the h&r block fafsa experiment and next steps (2013)
21. Blackwell, A., McLean, A., Noble, J., Rohrhuber, J.: Collaboration and learning through live coding (dagstuhl seminar 13382). In: Dagstuhl Reports. vol. 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2014)
22. Bok, H.S., Raman, K.S.: Software engineering productivity measurement using function points: a case study. Journal of Information Technology **15**(1), 79–90 (2000)
23. Brown, C.: Digital nudges for encouraging developer actions. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. pp. 202–205. IEEE Press (2019)
24. Brown, C.: How to improve software tool discovery and usage (2019), red Hat
25. Brown, C.: Sorry to bother you! effective oss recommendations (2019), `https://devconfus2019.sched.com/event/RFDu`, devConf.US
26. Brown, C., Middleton, J., Sharma, E., Murphy-Hill, E.: How software users recommend tools to each other. In: Visual Languages and Human-Centric Computing (2017), `http://people.engr.ncsu.edu/ermurph3/papers/vlhcc17-discovery.pdf`
27. Brown, C., Parnin, C.: Sorry to bother you: designing bots for effective recommendations. In: Proceedings of the 1st International Workshop on Bots in Software Engineering. pp. 54–58. IEEE Press (2019)
28. Cao, J., Kwan, I., Bahmani, F., Burnett, M., Fleming, S.D., Jordahl, J., Horvath, A., Yang, S.: End-user programmers in trouble: Can the idea garden help them to help themselves? In: 2013 IEEE Symposium on Visual Languages and Human Centric Computing. pp. 151–158 (Sept 2013). https://doi.org/10.1109/VLHCC.2013.6645260
29. Cao, J., Rector, K., Park, T.H., Fleming, S.D., Burnett, M., Wiedenbeck, S.: A debugging perspective on end-user mashup programming. In: Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on. pp. 149–156. IEEE (2010)
30. Cerezo, J., Kubelka, J., Robbes, R., Bergel, A.: Building an expert recommender chatbot. In: Proceedings of the 1st International Workshop on Bots in Software Engineering. pp. 59–63. IEEE Press (2019)
31. Charette, R.N.: Why software fails [software failure]. IEEE spectrum **42**(9), 42–49 (2005)

32. Cockburn, A., Williams, L.: Extreme programming examined. chap. The Costs and Benefits of Pair Programming, pp. 223–243. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001), http://dl.acm.org/citation.cfm?id=377517.377531
33. Crowston, K., Annabi, H., Howison, J., Masango, C.: Effective work practices for software engineering: free/libre open source software development. In: Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research. pp. 18–26. ACM (2004)
34. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in github: transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 conference on computer supported cooperative work. pp. 1277–1286. ACM (2012)
35. De Alwis, B., Murphy, G.C.: Using visual momentum to explain disorientation in the eclipse ide. In: Visual Languages and Human-Centric Computing (VL/HCC'06). pp. 51–54. IEEE (2006)
36. Devadiga, N.M.: Software engineering education: Converging with the startup industry. In: 2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T). pp. 192–196. IEEE (2017)
37. Diaper, D., Stanton, N.: The handbook of task analysis for human-computer interaction. CRC Press (2003)
38. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Commun. ACM **62**(8), 62–70 (Jul 2019). https://doi.org/10.1145/3338112, http://doi.acm.org/10.1145/3338112
39. Duflo, E., Kremer, M., Robinson, J.: Nudging farmers to use fertilizer: Theory and experimental evidence from kenya. American economic review **101**(6), 2350–90 (2011)
40. Espinosa, A., Kraut, R., Slaughter, S., Lerch, J., Herbsleb, J., Mockus, A.: Shared mental models, familiarity, and coordination: A multi-method study of distributed software teams. ICIS 2002 Proceedings p. 39 (2002)
41. Figas, P., Hagel, G., Bartel, A.: The furtherance of motivation in the context of teaching software engineering. In: 2013 IEEE Global Engineering Education Conference (EDUCON). pp. 1299–1304. IEEE (2013)
42. Fischer, G., Lemke, A., Schwab, T.: Active help systems, pp. 115–131. Springer Berlin Heidelberg, Berlin, Heidelberg (1984). https://doi.org/10.1007/3-540-13394-1_10, http://dx.doi.org/10.1007/3-540-13394-1_10
43. Fleming, S.D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., Kwan, I.: An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. ACM Transactions on Software Engineering and Methodology (TOSEM) **22**(2), 14 (2013)
44. Fogg, B.: Creating persuasive technologies: An eight-step design process. In: Proceedings of the 4th International Conference on Persuasive Technology. pp. 44:1–44:6. Persuasive '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1541948.1542005, http://doi.acm.org/10.1145/1541948.1542005
45. Forward, A., Lethbridge, T.C.: The relevance of software documentation, tools and technologies: a survey. In: Proceedings of the 2002 ACM symposium on Document engineering. pp. 26–33. ACM (2002)
46. Ge, X., Shepherd, D., Damevski, K., Murphy-Hill, E.: How developers use multi-recommendation system in local code search. In: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 69–76. IEEE (2014)

47. Goodman, P.S., Shah, S.: Familiarity and work group outcomes. Group process and productivity pp. 276–298 (1992)
48. Gordon, M., Guo, P.J.: Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. In: Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on. pp. 13–21. IEEE (2015)
49. Gousios, G., Pinzger, M., Deursen, A.v.: An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering. pp. 345–355. ACM (2014)
50. Gupta, P., Kim, Y.J., Glikson, E., Woolley, A.W.: Digitally nudging team processes to enhance collective intelligence. In: Proceedings of the 7th ACM Collective Intelligence (CI) Conference. ACM (2019)
51. Hanks, A.S., Just, D.R., Smith, L.E., Wansink, B.: Healthy convenience: nudging students toward healthier choices in the lunchroom. Journal of Public Health **34**(3), 370–376 (01 2012). https://doi.org/10.1093/pubmed/fds003, `https://doi.org/10.1093/pubmed/fds003`
52. Heckman, S., Stolee, K.T., Parnin, C.: 10+ years of teaching software engineering with itrust: The good, the bad, and the ugly. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training. pp. 1–4. ICSE-SEET '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183377.3183393, `http://doi.acm.org/10.1145/3183377.3183393`
53. Heckman, S., Williams, L.: A systematic literature review of actionable alert identification techniques for automated static code analysis. Information and Software Technology **53**(4), 363 – 387 (2011). https://doi.org/https://doi.org/10.1016/j.infsof.2010.12.007, `http://www.sciencedirect.com/science/article/pii/S0950584910002235`, special section: Software Engineering track of the 24th Annual Symposium on Applied Computing
54. Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D.: Trade-offs in continuous integration: assurance, security, and flexibility. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 197–207. ACM (2017)
55. Hu, Z., Gehringer, E.: Use bots to improve github pull-request feedback. pp. 1262–1263 (02 2019). https://doi.org/10.1145/3287324.3293787
56. Huang, N., Chen, P., Hong, Y., Wu, S.: Digital nudging for online social sharing: Evidence from a randomized field experiment. In: Proceedings of the 51st Hawaii International Conference on System Sciences (2018)
57. Jazayeri, M.: The education of a software engineer. In: Proceedings of the 19th IEEE international conference on Automated software engineering. pp. 18–xxvii. IEEE Computer Society (2004)
58. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In: Proceedings of the 2013 International Conference on Software Engineering (ICSE). pp. 672–681. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013). https://doi.org/10.1109/ICSE.2013.6606613, `http://people.engr.ncsu.edu/ermurph3/papers/icse13b.pdf`
59. Kaggle: Titanic: Machine learning from disaster, `https://www.kaggle.com/c/titanic`
60. Karim, M.R., Yang, Y., Messinger, D., Ruhe, G.: Learn or earn?-intelligent task recommendation for competitive crowdsourced software development (2018)
61. Karunasekera, S., Bedse, K.: Preparing software engineering graduates for an industry career. In: 20th Conference on Software Engineering Education & Training (CSEET'07). pp. 97–106. IEEE (2007)

62. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on software engineering (12), 971–987 (2006)
63. Kocher, M.G., Sutter, M.: Time is moneytime pressure, incentives, and the quality of decision-making. Journal of Economic Behavior & Organization **61**(3), 375–392 (2006)
64. Konstan, J.A., Riedl, J.: Recommender systems: from algorithms to user experience. User modeling and user-adapted interaction **22**(1-2), 101–123 (2012)
65. Layman, L., Williams, L., Amant, R.S.: Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In: Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on. pp. 176–185. IEEE (2007)
66. Leech, G.: Principles of Pragmatics. Longman linguistics library ; title no. 30, Longman (1983), https://books.google.com/books?id=TI1rAAAAIAAJ
67. Leventhal, H., Singer, R., Jones, S.: Effects of fear and specificity of recommendation upon attitudes and behavior. Journal of personality and social psychology **2**(1), 20 (1965)
68. Li, P.L., Ko, A.J., Zhu, J.: What makes a great software engineer? In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 700–710. IEEE Press (2015)
69. Linkenbach, J.W., Perkins, H.: Most of us are tobacco free: An eight-month social norms campaign reducing youth initiation of smoking in montana. (2003)
70. Linton, F., Joy, D., peter Schaefer, H., Charron, A.: Owl: A recommender system for organization-wide learning (2000)
71. Maalej, W., Tiarks, R., Roehm, T., Koschke, R.: On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. **23**(4), 31:1–31:37 (Sep 2014). https://doi.org/10.1145/2622669, http://doi.acm.org/10.1145/2622669
72. Madrian, B.C., Shea, D.F.: The power of suggestion: Inertia in 401 (k) participation and savings behavior. The Quarterly journal of economics **116**(4), 1149–1187 (2001)
73. Maltzahn, C.: Community help: Discovering tools and locating experts in a dynamic environment. In: Conference Companion on Human Factors in Computing Systems. pp. 260–261. CHI '95, ACM, New York, NY, USA (1995). https://doi.org/10.1145/223355.223664, http://doi.acm.org/10.1145/223355.223664
74. Matson, J.E., Barrett, B.E., Mellichamp, J.M.: Software development cost estimation using function points. IEEE Transactions on Software Engineering **20**(4), 275–287 (1994)
75. McNee, S.M., Riedl, J., Konstan, J.A.: Being accurate is not enough: How accuracy metrics have hurt recommender systems. In: CHI '06 Extended Abstracts on Human Factors in Computing Systems. pp. 1097–1101. CHI EA '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1125451.1125659, http://doi.acm.org/10.1145/1125451.1125659
76. Middleton, J., Murphy-Hill, E., Green, D., Meade, A., Mayer, R., White, D., McDonald, S.: Which contributions predict whether developers are accepted into github teams. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 403–413. MSR '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3196398.3196429, http://doi.acm.org/10.1145/3196398.3196429

77. Mirhosseini, S., Parnin, C.: Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 84–94. IEEE Press (2017)
78. Mirhosseini, S., Parnin, C.: Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 84–94. IEEE Press (2017)
79. Mirsch, T., Lehrer, C., Jung, R.: Digital nudging: Altering user behavior in digital environments. Proceedings der 13. Internationalen Tagung Wirtschaftsinformatik (WI 2017) pp. 634–648 (2017)
80. Murgia, A., Janssens, D., Demeyer, S., Vasilescu, B.: Among the machines: Human-bot interaction on social q&a websites. In: Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems. pp. 1272–1279. ACM (2016)
81. Murphy, G.C., Murphy-Hill, E.: What is trust in a recommender for software development? In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering. pp. 57–58. ACM (2010)
82. Murphy-Hill, E., Black, A.: Breaking the barriers to successful refactoring. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 421–430 (May 2008). https://doi.org/10.1145/1368088.1368146
83. Murphy-Hill, E.: Continuous social screencasting to facilitate software tool discovery. In: Proceedings of the 34th International Conference on Software Engineering. pp. 1317–1320. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012), http://dl.acm.org/citation.cfm?id=2337223.2337406
84. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. p. 42. ACM (2012)
85. Murphy-Hill, E., Lee, D.Y., Murphy, G.C., McGrenere, J.: How do users discover new tools in software development and beyond? Computer Supported Cooperative Work (CSCW) **24**(5), 389–422 (2015). https://doi.org/10.1007/s10606-015-9230-9, http://dx.doi.org/10.1007/s10606-015-9230-9
86. Murphy-Hill, E., Murphy, G.C.: Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In: Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work. pp. 405–414. CSCW '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1958824.1958888, http://doi.acm.org/10.1145/1958824.1958888
87. Nerur, S., Mahapatra, R., Mangalaraj, G.: Challenges of migrating to agile methodologies. Communications of the ACM **48**(5), 72–78 (2005)
88. Niazi, M., Mahmood, S., Alshayeb, M., Riaz, M.R., Faisal, K., Cerpa, N., Khan, S.U., Richardson, I.: Challenges of project management in global software development: A client-vendor analysis. Information and Software Technology **80**, 1–19 (2016)
89. O'Grady, S.: The New Kingmakers: How Developers Conquered the World. " O'Reilly Media, Inc." (2013)
90. O'keefe, D.J.: Persuasion. The International Encyclopedia of Communication (2002)
91. Rahman, A., Partho, A., Meder, D., Williams, L.: Which factors influence practitioners' usage of build automation tools? In: Proceedings of the 3rd International

Workshop on Rapid Continuous Software Engineering. pp. 20–26. IEEE Press (2017)
92. Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE software **27**(4), 80–86 (2010)
93. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google (2018)
94. Sadowski, C., Van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 598–608. IEEE Press (2015)
95. Schultz, P.W., Nolan, J.M., Cialdini, R.B., Goldstein, N.J., Griskevicius, V.: The constructive, destructive, and reconstructive power of social norms. Psychological science **18**(5), 429–434 (2007)
96. Senyard, A., Michlmayr, M.: How to have a successful free software project. In: 11th Asia-Pacific Software Engineering Conference. pp. 84–91. IEEE (2004)
97. Shen, L., Bigsby, E.: The effects of message features: content, structure and style. The SAGE handbook of persuasion developments in theory and practice (2012)
98. Singer, L.: On the diffusion of innovations: How new ideas spread (Dec 2016), `https://leif.me/2016/12/on-the-diffusion-of-innovations-how-new-ideas-spread/`
99. Singer, L., Figueira Filho, F., Storey, M.A.: Software engineering at the speed of light: how developers stay current using twitter. In: Proceedings of the 36th International Conference on Software Engineering. pp. 211–221. ACM (2014)
100. Singh, D., Sekar, V.R., Stolee, K.T., Johnson, B.: Evaluating how static analysis tools can reduce code review effort. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 101–105. IEEE (2017)
101. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. Acm Sigplan Notices **48**(6), 15–26 (2013)
102. Smith, J., Brown, C., Murphy-Hill, E.: Flower: Navigating program flow in the ide. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 19–23 (Oct 2017)
103. Society, I.C., Bourque, P., Fairley, R.E.: Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edn. (2014)
104. Storey, M.A., Zagalsky, A.: Disrupting developer productivity one bot at a time. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 928–931. ACM (2016)
105. Thaler, R.H., Sunstein, C.R.: Nudge: Improving decisions about health, wealth, and happiness. Penguin (2009)
106. Thaler, R.H., Sunstein, C.R., Balz, J.P.: Choice architecture (2014)
107. Tilley, S., Huang, S., Payne, T.: On the challenges of adopting rots software. In: Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering. pp. 3–6 (2003)
108. Trockman, A., Zhou, S., Kästner, C., Vasilescu, B.: Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In: Proceedings of the 40th International Conference on Software Engineering. pp. 511–522. ACM (2018)
109. Turkle, S.: Alone together: Why we expect more from technology and less from each other. Hachette UK (2017)
110. Urli, S., Yu, Z., Seinturier, L., Monperrus, M.: How to design a program repair bot?: insights from the repairnator project. In: Proceedings of the 40th Interna-

tional Conference on Software Engineering: Software Engineering in Practice. pp. 95–104. ACM (2018)
111. Viriyakattiyaporn, P., Murphy, G.C.: Challenges in the user interface design of an ide tool recommender. In: 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering. pp. 104–107. IEEE (2009)
112. Viriyakattiyaporn, P., Murphy, G.C.: Improving program navigation with an active help system. In: Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research. pp. 27–41. CASCON '10, IBM Corp., Riverton, NJ, USA (2010). https://doi.org/10.1145/1923947.1923951, http://dx.doi.org/10.1145/1923947.1923951
113. Weinmann, M., Schneider, C., vom Brocke, J.: Digital nudging. Business & Information Systems Engineering **58**(6), 433–436 (2016)
114. Wessel, M., de Souza, B.M., Steinmacher, I., Wiese, I.S., Polato, I., Chaves, A.P., Gerosa, M.A.: The power of bots: Characterizing and understanding bots in oss projects. Proceedings of the ACM on Human-Computer Interaction **2**(CSCW), 182 (2018)
115. Whitworth, B.: Polite computing. Behaviour & Information Technology **24**(5), 353–363 (2005). https://doi.org/10.1080/01449290512331333700, http://dx.doi.org/10.1080/01449290512331333700
116. Willman, S., Lindén, R., Kaila, E., Rajala, T., Laakso, M.J., Salakoski, T.: On study habits on an introductory course on programming. Computer Science Education **25**(3), 276–291 (2015)
117. Wilson, B.C., Shrock, S.: Contributing to success in an introductory computer science course: a study of twelve factors. In: ACM SIGCSE Bulletin. vol. 33, pp. 184–188. ACM (2001)
118. Wilson, C., Sudol, L.A., Stephenson, C., Stehlik, M.: Running on empty: The failure to teach k-12 computer science in the digital age (2010)
119. Wisdom, J., Downs, J.S., Loewenstein, G.: Promoting healthy choices: Information versus convenience. American Economic Journal: Applied Economics **2**(2), 164–78 (April 2010). https://doi.org/10.1257/app.2.2.164, http://www.aeaweb.org/articles?id=10.1257/app.2.2.164
120. Wohlin, C.: Empirical software engineering research with industry: Top 10 challenges. In: Proceedings of the 1st International Workshop on Conducting Empirical Studies in Industry. pp. 43–46. IEEE Press (2013)
121. Xiao, S., Witschey, J., Murphy-Hill, E.: Social influences on secure development tool adoption: Why security tools spread. In: Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing. pp. 1095–1106. CSCW '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2531602.2531722, http://doi.acm.org/10.1145/2531602.2531722