ABSTRACT

BROWN, JR., DWAYNE CHRISTIAN. Digital Nudges for Encouraging Developer Behaviors. (Under the direction of Dr. Chris Parnin.)

Decision-making is a vital part of software engineering. Professional software engineers, or developers, are regularly faced with decisions in their work. Moreover, as societal dependence upon technology increases, so does the complexity and impact of the choices programmers make during development processes. Although effective decision-making is critical for developing and maintaining quality software, developers frequently make bad decisions in practice. For example, software engineers often avoid useful *developer behaviors*, or tools and practices designed to help developers complete programming tasks. Despite scientific evidence of their benefit, studies show developers often ignore valuable behaviors such as adopting development tools, secure coding practices, ethical programming guidelines, and other constructive activities.

To increase the adoption of developer behaviors, existing research suggests face-to-face recommendations between colleagues is the most effective method for software engineers to learn about development tools and practices. However, opportunities for these in-person interactions between developers are declining as software engineering teams become more globally distributed and physically isolated via increased remote work among programmers. Bots and automated tools have also been utilized as a means to increase adoption of developer behaviors among software engineers. Yet, despite the advantages of using bots for automating programming tasks, developers often find automated recommendations from these systems to be ineffective and intrusive.

The goal of this research is to fill the gap created by the decline in face-to-face recommendations and the inadequacy of automated approaches to increase the adoption of useful developer behaviors. To achieve this goal, my work introduces ***developer recommendation choice architectures***, a conceptual framework for incorporating behavioral science concepts, specifically *nudge theory*, into automated recommendations for developers. Nudge theory is a framework for improving human behavior that focuses on influencing the environment surrounding decision-making, or *choice architecture*, without 1) providing incentives or 2) banning alternative options. In this work, I aim to use *digital nudges*, or incorporate technology to nudge humans toward better decisions in digital choice environments, to encourage software engineers to adopt better behaviors.

This dissertation advances knowledge in the field by using *developer recommendation choice architectures* to design automated recommendations for improving the decision-making and behavior of software engineers. This novel framework consists of three principles: *actionability*, *feedback*, and *locality*. The thesis of this research argues that *developer recommendation choice architectures* can nudge programmers to adopt better behaviors while developing software, resulting in enhanced code quality and increased productivity for developers. To construct the framework and evaluate this thesis statement, my dissertation consists of a collection of studies exploring recommendations to developers in software engineering:

1. To learn what makes effective recommendations for developers, I conducted studies analyzing two different recommendation techniques:
   (a) First, I analyzed characteristics of *peer interactions* to determine why face-to-face recommendations are effective. We found that tool suggestions from colleagues are beneficial for improving developer behavior because of their ability to foster *receptiveness*, i.e. users are likely to adopt familiar and desirable systems whereas politeness, persuasiveness, and types of tools are less impactful for decisions.
   (b) Next, I introduced the naive *telemarketer design* as a baseline approach for generating recommendations from bots. This approach was implemented in `tool-recommender-bot`, a system for generating tool recommendations. We found our bot was ineffective for developers because it violated *social context* and interrupted *developer workflow*, motivating the need for novel automated recommendation approaches.
2. To devise a new approach to improve automated recommendations, I used these findings and concepts from nudge theory to formulate *developer recommendation choice architectures*. I conducted a formative evaluation to explore the impact of this framework on automated suggestions and show it creates preferable recommendations for software engineers.
3. To evaluate the conceptual framework, I analyzed *developer recommendation choice architectures* within existing recommendation systems by conducting two studies investigating GitHub *suggested changes*, a recent tool that incorporates all of the framework principles:
   (a) The first of these studies explores styles of recommendations by observing developers interacting with different recommender systems. Participants preferred tool recommendations from the suggested changes feature over suggestions from other systems, such as email, because of its clear *communication* and effortless *workflow integration*.
   (b) The second study empirically analyzed the impact of suggested changes on GitHub development practices. We found recommendations from this system are well-accepted by programmers, improve the timing of recommendations between peers, boost coding activity, and increase discussions between developers during code reviews.
4. To further evaluate *developer recommendation choice architectures*, I developed `class-bot`, an automated system that integrates the framework principles to recommend beneficial developer behaviors to students working on programming assignments. We found this system was useful for enhancing the quality of students' code and increasing their productivity.

The contributions of this work are *developer recommendation choice architectures*, a *conceptual framework* for designing automated recommendations to developers, the above *set of experiments* which motivate and provide evidence for the framework, and an *automated recommender system* that incorporates the framework to make recommendations for developer behaviors. This dissertation concludes with broader implications and future directions for using *developer recommendation choice architectures* to improve the productivity, decision-making, and behavior of developers.

Digital Nudges for Encouraging Developer Behaviors

by
Dwayne Christian Brown, Jr.

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2021

APPROVED BY:

Dr. Sarah Heckman	Dr. Kathryn Stolee
Dr. Anne McLaughlin	Dr. Chris Parnin Chair of Advisory Committee

DEDICATION

This work is dedicated to all of my family and friends.

"*Rejoice in hope, be patient in tribulation, be constant in prayer.*"
ROMANS 12:12

BIOGRAPHY

Dwayne Christian Brown, Jr., also known as "Chris", received his high school diploma from Rock Hill High School in Rock Hill, SC. Afterwards, he obtained his Bachelor of Science degree in Computer Science from Duke University in 2013, where he conducted research studying K-12 Computer Science education and graduated with Distinction after completing his honors thesis, "*Integrating Computer Science Into Middle School Mathematics*", under the supervision of Dr. Susan Rodger. Upon the completion of his undergraduate degree, Chris spent two years as a contracted Python developer working for Bank of America in Charlotte, NC.

Motivated by his previous research experiences, Chris decided to return to graduate school to pursue a Ph.D. in Computer Science at North Carolina State University in 2015. He began his graduate career working with Dr. Emerson Murphy-Hill, under whom he obtained a Master of Science in Computer Science in 2017. During his time at NC State, Chris gained further industry experience through internships at Blackbaud (2016) and Red Hat (2017 and 2018). He also added teaching experiences as a TA for undergraduate software engineering (CSC326) and Java programming concepts (CSC216) courses with Dr. Sarah Heckman in addition to teaching the introductory Java programming course (CSC116) during Summer 2020.

Chris' doctoral research, under the advisement of Dr. Chris Parnin, explores improving the behavior of software engineers by integrating concepts from behavioral science into bots and automated systems. His research interests lie in the intersection of human factors, automation and tools, and empirical software engineering. Chris' research philosophy involves characterizing software engineering problems and developing tools and techniques to help solve these issues, with the goal of improving the behavior, productivity, and decision-making of programmers. He will join the Department of Computer Science at Virginia Tech as an Assistant Professor in Fall 2021.

ACKNOWLEDGEMENTS

First, I want to acknowledge my family, without whom this achievement would not have been possible. Thank you to my wife (Bethany Wagner Brown), parents (Dwayne Sr. and Banita), siblings (Kristen, Anita, and Nathaniel), grandfather (Walter White), and other relatives (Clifford, Waltrina, Cliff Jr., Walter, Helen, and Kyle) for your prayers, confidence, and encouragement. Additionally, I want to thank the many friends who provided support and encouragement throughout this process.

Next, I would like to thank the committee members for their time and feedback on my dissertation. To my advisor, Dr. Chris Parnin, for welcoming me to his research group and providing valuable guidance on my work. To Dr. Sarah Heckman, for believing in me as a TA, summer instructor, and providing insight into research and teaching. To Dr. Katie Stolee, for recruiting me for collaborations to expand my research knowledge and experience. And to Dr. Anne McLaughlin, for your collaboration and lending your psychology expertise to this interdisciplinary work.

Additionally, I want to acknowledge Dr. John-Paul Ore who stepped in as an emergency substitute committee member for my oral prelim exam. I am also extremely grateful for my undergraduate research advisors: Dr. Chad Jenkins for providing my first research experience through the CRA Distributed Research Experiences for Undergraduate (DREU) program and Dr. Susan Rodger, whose teaching, advising, and mentorship was invaluable for my career and motivated me to major in Computer Science at Duke University and to further pursue CS research.

I am very grateful to the current and past members of the software engineering research groups at North Carolina State University, including `alt-code`, the now-defunct Developer Liberation Front, RealSearch, RAISE, Dr. Stolee's lab, and all of the other students and researchers that have come through Engineering Building II rooms 3228 and 3229 over the years who have helped make NC State a Top 3 SE research program in the world.[1] I am proud to have been a part of this group and your collaborations, data analysis, feedback, and casual discussions have elevated my research and made this work possible. Additionally, these interactions have been dearly missed during the pandemic.

I would also like to thank other faculty, staff, and students in the department who have supported my throughout the CSC PhD program at NC State, including Dr. Guoliang Jin, Dr. Jamie Jennings, Leslie Rand-Picket and Kayla Bethea, Andrew Sleeth, the CSC Recruitment volunteers (both who recruited me and who I've worked with to recruit others), the Bahlers intramural basketball team, and the accountability writing group. Additionally, thanks to Jamin Quimby at Blackbaud and Og Maciel and Jake Callahan with the Satellite QE team at Red Hat for organizing excellent internship experiences.

There are too many people to thank for their contributions and support of this work in a variety of ways, and I sincerely apologize to anyone I overlooked. This work is primarily based on research supported by the National Science Foundation grant #1714538. Finally, I would like to thank all of the students and professional developers who volunteered their time to participate in the studies included in this dissertation. Without them, this research would have been impossible.

[1] `http://csrankings.org/#/index?soft&world`

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

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.

CHAPTER

2

INTRODUCTION

2.1 Motivation

Decision-Making in Software Engineering

Humans make approximately 35,000 decisions everyday,[1] each with varying outcomes and consequences, good or bad. Similarly, professional software engineers, or developers, are frequently faced with consequential decisions in their work. For example, decision-making is regarded as the "most undervalued" and "most important skill in software development", even moreso than coding skills [Woo19], and a "critical" characteristic of *great* software engineers [Li15]. The importance of these choices grows as technology plays an increasingly vital role in our daily lives.

Tools and guidelines informed by science can encourage humans to make better decisions and adopt beneficial behaviors. For example, the Center for Disease Control suggested wearing masks, social distancing, and avoiding crowds to prevent the spread of coronavirus.[2] In software engineering, researchers have developed and evaluated a wide variety of *developer behaviors*, or tools and practices designed to help developers complete programming tasks more effectively and efficiently, and show these behaviors improve software development processes.

However, like many ignored safety guidelines during the pandemic, developers frequently avoid useful developer behaviors in their work. For example, even though studies show static analysis tools are beneficial for preventing errors [Aye10], decreasing debugging time [Lay07], and reducing developer effort [Sin17], research also shows software engineers rarely use these tools in practice [Joh13].

[1] `https://go.roberts.edu/leadingedge/the-great-choices-of-strategic-leaders`
[2] `https://www.cdc.gov/coronavirus/2019-ncov/prevent-getting-sick/prevention.html`

Research also shows developers make other bad decisions while developing software such as avoiding security tools [Wit15], storing passwords in configuration files [Rah19], failing to upgrade software dependencies [Mir17a], and neglecting ethical programming guidelines [McN18].

This *developer behavior adoption problem* leads to negative consequences that are costly for users and developers. For instance, developers at Zoom failed to secure the video conferencing platform, which led to many "Zoom-bombing" attacks and security vulnerabilities.[3] Additionally, software failures impact billions of users and cost trillions of dollars to repair each year [Tri17]. As society becomes more dependent on technology, it is becoming increasingly important to find ways to improve developer behavior while developing and maintaining software to prevent bad decisions and reduce the impact of their consequences on society.

Developer Recommendations

To help increase adoption of useful behaviors, researchers have explored using automated recommender systems and bots to suggest actions to 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" [4]. Likewise, recommendation systems for software engineering are designed to actively assist developers in seeking information and making decisions while developing software [Rob10]. Spyglass, for instance, is an automated recommender system 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 [Vir10].

However, research also suggests existing approaches for automated recommendation systems and recommender bots are ineffective in their interactions with developers. For example, Viriyakattiyaporn and colleagues found that the inability to deliver suggestions in a timely manner discouraged programmers from adopting tool recommendations from Spyglass [Vir09]. Additionally, studies report developers face many challenges interacting with bots in open source software [Wes18], have negative perceptions of bots to automatically manage software dependencies [Mir17a], and express more frustration in conversations with chatbots compared to humans [Hil15].

While many automated approaches have been developed to help developers make better choices, research shows that face-to-face recommendations between humans are the most effective. For example, Murphy-Hill and colleagues found that *peer interactions*, or the process of learning about tools from coworkers during normal work activities, are the most effective method for software engineering tool discovery compared to other technical approaches such as tool encounters in development environments, social media and websites, tutorials, and discussion threads online [MH11]. Additionally, research shows knowledge sharing and learning from peers are benefits of *pair programming*, or developers working together on the same computer to complete programming tasks [Coc01].

However, even though peer interactions are the most effective method for recommendations to

[3] `https://blog.zoom.us/wordpress/2020/04/01/a-message-to-our-users/`

[4] `https://recsys.acm.org/`, as quoted by [Rob10]

developers, Murphy-Hill also found these recommendations between colleagues occur infrequently in the workplace [MH11]. There are many barriers to peer interactions, such as increased physical isolation due to remote work, developers working in different programming environments, and software engineers being unwilling to learn and share tool knowledge [MH15]. The decline of peer interactions and ineffectiveness of existing automated approaches point to the need for new methods to effectively recommend beneficial behaviors to software engineers in their work.

2.2 Research Overview

To improve the effectiveness of automated recommendations encouraging developers to adopt better practices, my research involves interdisciplinary work analyzing the developer behavior adoption problem through the lens of behavioral science. Specifically, I explore using *nudge theory*, a behavioral science concept for improving human behavior and decision-making, to encourage developers to make better decisions and adopt beneficial behaviors. A *nudge* refers to any factor that impacts human decision-making without providing incentives to individuals or banning alternative options [Tha09]. Additionally, *digital nudges* refer to using technology and user interfaces to influence user behavior in digital choice environments [Wei16]. More details and examples of nudge theory can be found in Chapter 3 of this dissertation.

In nudge theory, *choice architecture* is the idea that the way choices are framed and presented impacts human decisions [Tha13]. To incorporate nudge theory into automated recommendations for developer behavior, my research introduces ***developer recommendation choice architectures***, a conceptual framework to improve the way recommendations are displayed to software engineers in their work. This framework consists of three design principles:

1. *actionability*, or the ease with which developers can adopt the target behavior
2. *feedback*, or the clarity and relevance of the information provided
3. *locality*, or the placement and timing of recommendations.

To construct and evaluate the framework, this work consists of mixed-methods studies collecting and analyzing quantitative and qualitative data to characterize problems with developer recommendations and evaluate tools and techniques to overcome these challenges. The thesis of this dissertation argues we can encourage developers to adopt behaviors to improve the quality of their work and productivity of their development processes by incorporating *developer recommendation choice architectures* into automated recommendations.

2.3 Contributions

This research advances knowledge by making several contributions to defend the thesis statement presented in Chapter 1:

> *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.

To examine "recommendations for software engineers", this research posits:

- a *set of experiments* investigating why *peer interactions* are effective for improving developer behavior and introducing the naive *telemarketer design* baseline automated recommendation approach to evaluate the ineffectiveness of recommendations from bots.

To explore how "we can nudge developers to adopt behaviors", this work presents:

- a *conceptual framework*, ***developer recommendation choice architectures***, to apply concepts from nudge theory to improve the effectiveness of automated recommendations to software engineers.

To defend the claim that "incorporating developer recommendation choice architectures into recommendations for software engineers...can nudge developers to adopt behaviors useful for improving code quality and developer productivity", this research submits:

- a *set of experiments* analyzing GitHub *suggested changes*, an novel recommendation system that incorporates the framework principles to support code recommendations between developers on pull requests.
- `class-bot`, a novel *automated recommender system* that incorporates *developer recommendation choice architectures* to generate digital nudges recommending useful developer behaviors to programmers.

2.4 Outline

The remainder of this dissertation characterizes my research and presents experiment methods and results used to provide evidence to support the thesis statement (Chapter 1) and investigate the research problem and objectives (Chapter 2).

Chapter 3 provides background information for this research, including more details about *developer behavior* and *nudge theory*, two concepts that are critical for the research presented in this dissertation, and describes the existing literature related to this body of work.

Chapter 4 describes preliminary work examining different recommendation approaches to gain insight into what makes an effective recommendation to developers and motivate the need for a novel approach for suggesting developer behaviors.

Chapter 5 introduces *developer recommendation choice architectures* and presents a formative evaluation exploring *actionability*, one of the *developer recommendation choice architectures*, to gather insight on recommendations with this framework from developers.

Chapter 6 presents studies analyzing GitHub *suggested changes*, an existing recommender system that incorporates the *developer recommendation choice architectures* principles, to evaluate the framework by comparing this system to other recommendation styles and empirically analyzing its impact on GitHub development practices.

Chapter 7 introduces a novel recommendation system, `class-bot`, a tool designed to use *developer recommendation choice architectures* in automated notifications. This section presents the results of a study using `class-bot` to improve the development behaviors of students on programming assignments.

Chapter 8 revisits the thesis statement and the contributions of the research presented in this body of work. This dissertation concludes with broader implications and future directions for *developer recommendation choice architectures*, using this framework to continue observing and enhancing developer behavior and motivating the design of future tools for making recommendations to improve the productivity, decision-making, and behavior of developers.

Finally, the appendix includes supplemental information and study materials for the research and experiments presented in this dissertation.

CHAPTER

3

BACKGROUND

"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?'"[1]

3.1 Developer Behavior

Developer behavior refers to the wide array of practices designed to help software engineers complete tasks while developing and maintaining software applications. Software engineering researchers have analyzed countless developer behaviors and show these activities provide many benefits to development teams. An example of a beneficial behavior is utilizing development tools to automatically complete programming tasks. The IEEE Software Engineering Body of Knowledge (SWEBOK), a suite of widely accepted software engineering practices and standards, suggests adopting development tools is a "good practice" that can "enhance the chances of success over a wide range of project" [SWEBOK, p. A-4]. For example, using static analysis tools, or systems that automatically examine code to detect errors without running the program, is a behavior useful for improving code quality [Aye10], preventing errors [Bes10], decreasing debugging time [Lay07], lowering development costs [Le 12], and reducing developer effort [Sin17].

However, despite scientific evidence of their benefit, studies also show software engineers often avoid static analysis tools [Joh13], sparingly fix bugs reported during automated static analysis [Mar19], and seek to ignore alert notifications from these systems [Imt19]. Software engineering researcher Greg Wilson argues this tendency for developers to ignore useful programming tools and practices, or what I refer to as the *developer behavior adoption problem*, is the most important research topic in the field for

[1] `https://twitter.com/gvwilson/status/1142245508464795649?s=20`

the next ten years.[1] The goal of this research is to work towards a solutions to this problem by creating effective systems that make convincing recommendations to encourage working programmers to adopt better behaviors and practices.

3.1.1 Developer Behavior Adoption Problem

Software engineering literature suggests developers frequently ignore beneficial development practices in their work, even though research outlines their advantages and benefits. For example, in addition to ignoring static analysis tools, studies have explored challenges and reasons why developers in industry avoid adopting automated tools for security [Xia14], debugging [Cao10], refactoring [MH08], documentation [For02], build automation [Rah17], and continuous integration [Hil17]. Furthermore, researchers have examined why developers fail to adopt additional useful programming behaviors such as secure coding practices [Men18], adequate software testing [Whi00], agile development methodologies [Ner05], ethical programming guidelines [McN18], and more.

Ultimately, the developer behavior adoption problem can be costly for software developers and users. For example, the National Institute of Standards and Technology reports *debugging*, or the process of finding and removing errors in code, is the most expensive and time-consuming development activity using 50-75% of total costs and 70-80% of programmers' time [NIST02]. Additionally, Tricentis reported software failures impacted 3.7 billion users and lost over $1.7 trillion USD in 2017 [Tri17]. Furthermore, poor developer decisions have increasing effects for society over the next ten years as we become more dependent upon technology. Philosophy professor John K. Davis argues societal problems caused by technology, such as fake news, election interference, and data privacy concerns, will worsen as the power of technology outpaces the wisdom of humans [Dav20]. Thus, my research seeks to explore ways to help developers make wiser decisions and avoid these consequences by increasing the adoption of beneficial developer behaviors.

Disclaimer*

There are many reasons developers make poor choices and avoid beneficial programming behaviors. Although the research presented in this dissertation primarily focuses on programmers, bad decision-making in software engineering is not solely the fault of developers. While *developer inertia*, or the unwillingness of programmers to adjust their existing workflow to learn new tools and practices, prevents the adoption of valuable behaviors [MH11], many other barriers also prevent the adoption of developer behaviors in industry. For example, as reliance on technology increases, so do the number and complexity of decisions developers make in their work. Cognitive issues, including *decision fatigue* and *choice overload*, where decision quality declines as humans make more decisions and are overwhelmed by the amount of choices, can impair the decision-making of developers and lead to poor behaviors, such as copy-and-paste programming [Mak11].

Companies can also prevent the adoption of developer behaviors. For example, Xiao and colleagues found that a company's policies and standards, culture, and structure play a role in developers' adoption of security tools [Xia14]. Similarly, Nerur and colleagues suggest management and organizations

can block development teams from migrating to agile methodologies [Ner05]. Other barriers, such as mandated tools and processes [MH15], time pressure and deadlines [Cos84], and globally distributed development teams [Ebe08], inhibit developers from discovering and adopting beneficial programming tools and practices in industry. Additionally, researchers at Google found that non-technical work factors, such as enthusiasm, support from peers, and performance feedback, play the most important role in the productivity of software developers [MH19b].

Furthermore, researchers and toolsmiths contribute to the developer behavior adoption problem. Tilley and colleagues suggest adoption should be the goal for research-off-the-shelf (ROTS) software [Til03]. However, Norman argues research results and tools often fail to meet the needs of industry developers, leading to a growing *research-practice gap* [Nor10]. For instance, Johnson and colleagues found the main reasons developers avoid static analysis tools are the inability to understand results, difficult customization and integration, and distrust of tool output [Joh13]. Additionally, Wohlin et al. outline challenges integrating empirical software engineering research into industry, citing issues such as lack of trust, differing goals, and poor knowledge exchange and integration [Woh13]. In this work, I explore the developer behavior adoption problem from the perspective of software engineers by using behavioral science concepts to answer the motivating research question: "*How do we get working programmers to actually adopt better practices?*".[1]

3.2 Nudge Theory

To encourage the adoption of useful developer behaviors, I explore incorporating concepts from *nudge theory* to improve automated recommendations to developers. A nudge is defined as any factor "that alters behavior in a predictable way without forbidding alternatives or significantly changing economic incentives" [Tha09, p. 6]. For example, placing healthier foods at the front of a high school cafeteria in a "convenience line" nudged students to increase consumption of fruits and vegetables [Han12]. This example fits the definition of a nudge because students have the option to ignore the target behavior, in this case eating healthy, to select unhealthy foods and they do not receive a reward for deciding to choose healthier options. Nudges are also used to impact human behavior on a much larger scale, for example the United States, the UK, Denmark, and Italy have implemented nudge unit teams to improve the behavior and decision-making of citizens [Cap15]. Using nudge theory to improve the behavior and decision-making of software engineers involves not providing incentives nor forcing developers to adopt useful tools and practices.

3.2.1 Digital Nudges

Digital nudging refers to using technology and user interface design elements to nudge user behaviors in digital choice environments [Wei16]. For example, the FitBit[2] 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 [Wei16]. Prior work has

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

also explored using digital nudges online to improve human behaviors such as enhancing user privacy and security decisions [Acq17], increasing financial savings [Mad01], influencing social media sharing practices [Hua18], and reducing social media usage [Pur20].

Mirsch and colleagues argue implementing digital nudges are "easier, faster and cheaper" and provide more functionality than regular nudges [Mir17b, p. 635]. Furthermore, as more decisions are being made online, Weinmann argues digital nudges are becoming increasingly important because the design of interfaces will "always (either deliberately or accidentally) influences people's choices" [Wei16, p. 433]. While the majority of prior work in digital nudges studies their impact on the decision-making and behavior of software users, there is limited work exploring how they influence software developers, who are frequently faced with decisions in their work in digital choice environments, such as whether or not to adopt developer behaviors. This research aims to use digital nudges to encourage developers to adopt better practices by introducing and evaluating *developer recommendation choice architectures*.

3.2.2 Choice Architecture

Nudges and digital nudges are effective for improving human behavior is because of their ability to influence the context and environment surrounding decision-making, or *choice architecture* [Tha13]. Choice architecture suggests the way decisions are framed and presented impacts the choices humans make. For example, changing the location of fruits and vegetables makes the choice of whether or not to eat healthy easier for humans. Another instance of a choice architecture is the "default rule", which suggests most decision-makers will select the default options when making decisions and has been shown to be effective for improving human behavior. For instance, the Square mobile payment app[3] sets the default behavior for users to tip merchants or actively choose a "no tipping" option. By using the default rule to make tipping the primary action, Square merchants in total earned over $70 million in tips in 2013.[4]

Thaler and Sunstein note "nudges are everywhere" because "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" [Tha09, p. 255]. Johnson and colleagues introduce 11 practical tools for choice architecture to convince people to adopt target behaviors and make better decisions [Joh12]. The research presented in this dissertation introduces and evaluates *developer recommendation choice architectures*, a novel approach that incorporates this concept into automated recommendations to developers. I aim to show that using choice architecture to improve the design and presentation of developer behavior decisions to software engineers within the context of their work can encourage working programmers to adopt better practices.

[3] `https://squareup.com/us/en`

[4] `https://www.fastcompany.com/3022182/how-square-registers-ui-guilts-you-into-leaving-tips`

3.3 Scope of Work

Throughout the rest of this dissertation, the term *nudge* is used to describe automated notifications recommending beneficial developer behaviors to software engineers as digital nudges. The nudges studied and created in this work do not provide incentives for developers choosing to adopt behaviors and allow developers to ignore our recommendations and choose alternate actions. Moreover, while nudges can be applied to many different aspects of software engineering, such as the IDE design, programming languages, and physical workspaces, my research aims to discover if nudges can encourage developers to adopt useful software engineering practices when faced with decisions while completing real-world programming tasks. The primary developer decision-making environment used to examine nudges 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 [Oct].[5] To evaluate nudges and automated recommendations, I developed software robots, or *bots*, to recommend beneficial developer behaviors on public open source repositories to GitHub users.

3.4 Related Work

3.4.1 Making Recommendations to Developers

This work builds upon prior research exploring methods for making recommendations to software engineers. For the purpose of this research, *developer recommendations* refer to any means of making suggestions or conveying information to improve the behavior of software engineers. To increase adoption of useful developer behaviors, prior work has investigated a variety of developer recommendation methods to suggest tools and practices to software engineers.

3.4.1.1 Peer Learning

Learning from peers, or human-to-human recommendations between with developers, has been shown to be an effective method for increasing knowledge. Twidale posits *over-the-shoulder learning*, or informal collaborative learning and help-giving sessions between colleagues, as an effective approach for learning during computer supported work [Twi05]. For example, Murphy-Hill et al. explored how software engineers learn about new development tools and found that *peer interactions* were the most effective method for software engineers to discover development tools compared to other technical methods [MH15]. Likewise, research shows *peer debriefings*, or discussions between developers, are effective for improving code comprehension [Maa14] and *coworker recommendations* are the most popular method for spreading knowledge and increasing security tool adoption [Xia14].

Another form of over-the-shoulder learning in software engineering is *pair programming*, or two developers working together to write code at the same machine. Cockburn and Williams found one of the benefits of this practice is learning from coworkers [Coc01]. Similarly, Begel and Nagappan found

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

developers at Microsoft reported that the spreading of code understanding and learning from partners were some of the main advantages of pair programming [Beg08]. *Peer code reviews*, where programmers critique code contributions before integrating them into source code, provide further opportunities for developers to learn from each other. For example, research shows programmers find peer code reviews beneficial for finding defects and improving code in addition to learning and knowledge transfer between developers [Bac13]. Furthermore, Cohen and colleagues argue *over-the-shoulder reviews* are the most common type of peer code review in practice, and suggest these reviews "lend themselves to learning and sharing between developers" [Coh06, p. 27].

While peer learning is effective for making recommendations and presenting knowledge to developers, studies show opportunities for these human-to-human interactions are declining. For example, Murphy-Hill and colleagues found peer interactions are the most effective mode of tool discovery, they also found these in-person recommendations occur infrequently in practice [MH11]. Additionally, Herbsleb argues the increase of global software engineering and distributed development teams limits changes for peer learning and causes less frequent and less effective communication between developers due to different time zones, cultural differences, and geographic distance [Her07]. Another factor is the increase in remote work among software engineers. Turkle argues that as humans increasingly work and collaborate in isolated environments, technology will replace face-to-face communications [Tur17]. Thus, it is necessary to explore technical approaches to make recommendations for developer behaviors.

3.4.1.2 Online Programming Communities

In-person human-to-human recommendations are in decline, however prior work has also explored the impact of online programming communities, or socio-technical websites dedicated to sharing knowledge and information specifically to programmers. In general, research shows online communities are beneficial for knowledge sharing across geographic locations and positional status [Hwa15]. As opposed to over the shoulder learning through human-to-human recommendations, online programming communities support learning from peers through many-to-many and many-to-one recommendations [Jen17]. Software engineering researchers have explored recommendations and developer learning in online programming communities. For example, Stack Overflow is a popular online question and answer site where developers can receive answers to their questions or respond to queries on many different programming topics. Research suggests comments on Stack Overflow posts are valuable for learning due to their ability to provide improvements and explanations [Sen20].

Additionally, researchers have explored the impact of online programming communities on recommendations to developers through social media. For example, prior work argues that social media has changed the way developers learn and share information GitHub [Beg10]. Specifically, Singer et al. analyzed Twitter and found the popular social media platform is beneficial for increasing awareness and knowledge of development practices [Sin14]. Aniche and colleagues also examined *modern news aggregators* about software development, such as Hacker News and r/programming on Reddit, and found the purpose of most posts on these platforms are geared towards learning. Additionally, they found that receiving knowledge to apply different perspectives to existing work and making recommendations are

the primary motivation for developers to post and respond to posts in these communities [Ani18].

Furthermore, prior work has explored recommendations to developers on code collaboration websites such as GitHub. For instance, research shows that badges on GitHub repositories are beneficial for improving the behavior of developers in terms of the quality of projects [Dab12], GitHub Discussions are valuable for learning between users [Hat21], and pull requests provide opportunities for programmers to learn from looking at other developers' code [Kal15]. While online programming communities are useful for providing information and making recommendations to developers, these platforms are *passive help systems*, or resources that require users to explicitly seek help. Prior work suggests these manual help-seeking systems are ineffective for making recommendations [Fis84].

3.4.1.3 Other Passive Approaches

Previous research has explored a variety of other methods for making recommendations to software engineers. For example, prior work has also proposed continuous social screencasting [MH12a] and live-coding [Bla14] as a mechanisms for developers to learn from peers virtually. Likewise, researchers have explored using *crowdsourcing* to provide recommendations to developers to recommend programming tasks [Mao15] and increase understanding of Java APIs [Sun19]. Prior work has also proposed using *gamification* to encourage developers to adopt better tools and practices through various methods such as *Blaze*, a system that uses points and leaderboards to improve programmer behavior [Sni14], *Free Hugs*, an environment where developers create alter egos that evolve as players adopt better practices [Rya06], and analyzing development tool usage through Serious Game Design Assessment (SGDA) frameworks [Bar16]. Additionally, Murphy-Hill and colleagues posit Testing on the Toilet as a method for recommending development tools to programmers through restroom advertising [MH19a].

However, similar to seeking help in online programming communities, these developer recommendation approaches are examples of passive help systems. Previous research by Fischer and colleagues argues *active help systems* that can automatically make recommendations to users while completing tasks are more effective than manual and static methods for software users [Fis84]. As opportunities for peer learning decline and static recommendation methods remain insufficient, automated approaches are needed to make recommendations and convey information to developers. Thus, the primary focus of my research involves analyzing and developing active help systems to recommend developer behaviors to software engineers. In the next section, I present relevant work studying automated techniques for making developer recommendations.

3.4.2 Recommendation Systems

Recommender systems, or tools that automatically collects inputs and aggregates and presents desired outputs to those seeking recommendations, are useful for assisting users in making choices when faced with insufficient personal experience [Res97]. Prior work suggests these active help systems are more valuable for making recommendations to humans than passive systems. For example, Schafer and colleagues show that Automatic recommendations are more effective than Manual recommendations in e-commerce websites to suggest products to customers [Sch99]. Research has explored the concept

of recommender systems for generating recommendations to software users. For example, automated systems have been used to recommend entertainment content on Netflix [GU16].

Prior work offers tools and algorithms to generate a variety of recommendations to users. For example, Amazon implemented *item-to-item collaborative filtering* to recommend products to customers [Lin03], Yahoo! employs a *contextual-bandit* approach to recommend personalized news articles [Li10], OWL uses *organization-wide learning* to log user commands to recommend tools Microsoft Word [Lin00a], the Lumière Project uses *Bayesian network models* to predict the goals and needs of software users [Hor98], and YouTube uses a *batch-oriented pre-computation* algorithm to recommend videos to users on the world's most popular online video community [Dav10]. The research presented in this dissertation aims to study recommender systems focused on delivering recommendations to improve the behavior and decision-making of software engineers.

3.4.2.1 Recommendation Systems for Software Engineering

Recommendations systems for software engineering (RSSEs) are active help systems designed to assist and guide the actions of developers while completing programming tasks [Rob10]. RSSEs are made up of three components: 1) a data collection mechanism; 2) a recommendation engine to analyze input and generate recommendations; and 3) a user interface to present recommendations. For example, Hipikat observes the current state of source code to recommend software development artifacts, such as Bugzilla issue reports, to developers via an Eclipse IDE plugin [Cub03]. Software engineering researchers and toolsmiths have created and evaluated a variety of RSSEs to help developers complete a wide range of programming tasks. For example, Spyglass [Vir10], Tricorder [Sad15], Prompter [Pon14], and Dhruv [Ank06] are automated systems developed to recommend code navigation techniques, program analysis behaviors, Stack Overflow posts, and bug report artifacts to software engineers. According to Gasparic and Janes, the majority of RSSEs recommend source code changes and coding artifacts to developers and the primary goals of most systems are to support a lack of knowledge among software engineers and overcome insufficient help from existing tools [Gas16].

Similar to general recommender systems, RSSEs can be implemented with a variety of different methods to recommend developer behaviors through differing means. For example, Stench Spyglass [Vir10] and Prompter [Pon14] provide information to developers within their development environment as Eclipse plugins whereas Dhruv presents recommendations in the OpenACS Bugtracker[6] [Ank06] and Tricorder is integrated into project builds at Google [Sad15]. Additionally, previous research shows RSSEs are valuable for providing information to developers throughout the different phases of the software development processes [Pak14]. Prior work has also analyzed various recommendation algorithms in RSSEs, such as sorting by popularity, history-based recommendations, and collaborative filtering, to propose novel techniques [MH12b]. However, studies show developers often ignore recommendations from these systems. For example, prior work by Viriyakattiyaporn and Murphy suggests that the inability to make noticeable and timely recommendations led to ignored suggestions from Spyglass [Vir09]. My research seeks to improve the effectiveness of recommendation systems for software engineering

[6]https://www.project-open.com/en/package-bug-tracker

that refer to tools and systems designed to provide information to developers while developing and maintaining software.

In addition to creating automated recommender systems, research has explored ways to design effective recommendations. For instance, Fogg also outlines design principles for creating and designing persuasive technologies to encourage users to adopt target behaviors [Fog09]. Generally, researchers have motivated the need to focus on user experiences and design in recommendation systems to encourage the adoption of target behaviors. For example, McNee and colleagues argue *user-centric* recommendations focused on experiences and expectations are more important than the accuracy of recommender systems [McN06]. Similarly, Konstan and Riedl suggest evaluating user experiences metrics is more important for automated recommender systems than optimizing recommendation algorithms [Kon12]. For RSSEs, Murphy and Murphy-Hill analyzed recommender systems for software development and found that *trust* was more important than precision for software engineers, and conclude that "trust trumps precision" [Mur10] while Bavota and colleagues posit guidelines for designing systems to recommend code refactoring changes and argue the usability of refactoring recommendation systems is critical [Bav14].

While research suggests most RSSEs present results to users as lists [Gas16], software engineering researchers have evaluated specific design decisions for improving automated developer recommendations. For example, prior work with Smith and Murphy-Hill found *in-situ* design principles for a code navigation tool increased branchless navigation and was preferred by users [Smi17]. Johnson and colleagues propose using developers' experiences to customize programming tools and environments with *bespoke tools* [Joh15]. To study the initiation of system recommendations, Xiao et al. examined *proactive* and *reactive* user interface assistants and found that invocations not requiring user initiation were effective for applying and recalling commands [Xia03]. Furthermore, Robillard and colleagues present categories for design decisions to consider when developing RSSEs, including the context and input, recommendation engine and data source, and the output mode [Rob10]. My research proposes a new approach, *developer recommendation choice architectures*, to improve the design of recommender bots by integrating concepts from nudge theory in systems to improve the decision-making and behavior of developers.

3.4.2.2 Recommender Bots

Research has explored the use of software robots, or bots, to automatically make recommendations to users. For example, prior work has explored the usage of bots on social media platforms such as Twitter [Edw14] and e-commerce websites [Red18]. Prior work posits DevBots, or automated systems to support software development tasks [Erl19]. For example, David-DM[7] and Greenkeeper[8] are bots designed to recommend dependency updates for source code, Repairnator is a system that automatically fixes continuous integration build errors [Url18], ReviewBot automates static analysis and recommends pull request reviewers during code reviews [Bal13], and Mediam seeks to increase the adoption of

[7] `https://david-dm.org/`
[8] `https://greenkeeper.io/`

analyses and techniques evaluated in software engineering research in industry [Bes17]. For the purposes of my research, I refer to DevBots created to automate tasks and convey information to software engineers as bots.

Bots have been highly adopted among software development teams. For example, research shows approximately a third the repositories on GitHub employ a type of bot on their project [Wes18]. However, prior work suggests recommendations from bots are insufficient for improving the behavior of developers. For example, Wessel and colleagues found that bots are useful for automating a variety of development tasks in open source software but also show software engineers reported facing many challenges interacting and comprehending feedback from these automated systems [Wes18]. Similarly, Erlenhov et al. discovered programmers find development bots inconvenient due to their interruption and noise, lack of trust, and poor usability [Erl20] while Mirhosseini and Parnin found that developers were overwhelmed by bots generating automated pull requests on repositories [Mir17a]. Additionally, studies show developers exhibit more frustration through inappropriate language and negative emotions during interactions with chatbots [Hil15], pull requests from bots take significantly longer to review than those submitted by humans [Wyr21], and systems emulating humans with human-presenting profiles are more effective than noticeable bot accounts [Mur16a].

Prior work points to bot-human interactions as a primary factor in the reception of automated notifications. For example, the *Principles of bot design* from Microsoft suggest that "how *smart* the bot is" does not influence its success but delivering "a great user experience" does.[9] Similarly, Storey and Zagalsky note reducing interruptions, supporting context switching, and incorporating situational awareness into bots can improve the productivity of developers [Sto16]. Cerezo et al. also suggest *user-driven communication* can improve the perception and adoption of recommendations from chatbots rather than using single-purpose bot-driven techniques [Cer19]. My research aims to advance work in this field by utilizing interdisciplinary concepts to improve the design of automated recommendations from bots to increase their effectiveness for encouraging developers to adopt useful tools and behaviors.

3.4.3 Interdisciplinary Methods for Improving Behavior

Finally, my research applies concepts from behavioral science to improve the behavior and decision-making of software engineers. Prior work has similarly explored using interdisciplinary techniques to improve developer recommendations and learning. For example, Fleming and colleagues examined *information foraging theory*, the study of how humans search for information, and apply relevant concepts to software engineering and how programmers seek information [Fle13]. Cao and et al. use Minimalist Learning Theory to develop *Idea Garden*, an approach for integrating learning into programming environments and tasks to increase learning [Cao12]. 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 [Sin16]. This work builds upon these studies to explore using behavioral science concepts to influence the behavior and decision-making of developers using nudge theory.

[9] https://docs.microsoft.com/en-us/azure/bot-service/bot-service-design-principles?view=azure-bot-service-4.0

To our knowledge, this was the first work to incorporate nudge theory and digital nudges to influence the behavior of software engineers. Prior research in this area has primarily focused on using digital nudges to influence the behavior of software users. For example, Acquisti and colleagues explored using digital nudges to improve user privacy and security decisions online [Acq17]. Likewise, Huang and colleagues found that digitally nudging social media users impacted social sharing behavior [Hua18]. While these studies show that digital nudges are effective for influencing the behavior of software users, I explore using nudges to improve the behavior of software engineers. However, after introducing nudge theory as a solution to improve software engineer behavior and decision-making, recent work has explored using nudges to accelerate code reviews and encourage developers to review stale pull requests [Mad20]. In my work, I aim to use nudge theory to establish a framework for designing automated recommendations to improve the behavior and decision-making of programmers.

CHAPTER

4

DETERMINING WHAT MAKES AN EFFECTIVE DEVELOPER RECOMMENDATION

This chapter presents studies evaluating two different techniques for recommending developer behaviors, namely *peer interactions* and the naive *telemarketer design*. To explore what makes effective recommendations to software engineers, we analyze in-person tool recommendations between users completing tasks and introduce a baseline automated approach in a simple bot for making recommendations to software engineers to gain insight into what makes a compelling developer recommendation and motivate the need for a new technique. Additional details and study materials for these experiments can be found in Appendix A.

4.1 Peer Interactions

Peer interactions are defined as the process of software engineers learning about development tools and practices from colleagues in-person during normal work activities [MH11]. Murphy-Hill and colleagues examined different modes of development tool discovery among software engineering, including peer interactions and other technical approaches such as random tool encounters in development environments, tutorials, descriptions or mentions online or in publications, discussion threads, social media sites such as Twitter and RSS Feeds, and comments and discussion in online forums, and found that peer interactions are the most effective way developers discover new software engineering tools [MH11].

Furthermore, software engineering literature shows peer interactions are useful for supporting additional developer behaviors such as increasing adoption of security tools to help developers build more secure systems by detecting security vulnerabilities [Xia14], increasing collaborative development practices [Kal15], and improving how developers understand and share knowledge about their code [Maa14]. Little is known about the nature of peer interactions, and to explore what makes these user-to-user recommendations effective I performed a user study to observe and analyze tool recommendations between peers.

4.1.1 Study Rationale

To increase awareness of useful tools and features designed to help users in software, recommender systems can automatically suggest beneficial tools to users. However, despite the large number of automated recommender systems, prior work suggests user-to-user peer interactions are the most effective method for tool discovery [MH11]. There is limited research exploring why face-to-face recommendations between peers are effective increasing tool adoption, and to better understand why users prefer recommendations from colleagues this work administers a user study to analyze different characteristics of the recommendations. The characteristics we analyzed are motivated by existing literature in psychology and persuasion theory, as well as prior software engineering research examining peer interactions. The results of this work provide insights into why in-person tool recommendations between peers are effective and implications for improving automated recommender systems.

4.1.1.1 Research Question

To examine the influence of peer interactions on recommendations between software users, this work sought to answer the following research question:

RQ1 What characteristics of peer interactions make recommendations effective?

To answer this question, we investigated the effectiveness of peer interactions by conducting a user study to observe tool recommendations between 13 pairs of participants completing data analysis tasks. We analyzed the peer interactions by recognizing tool recommendations between partners, observing how tools were suggested based on different recommendation characteristics, and detecting how often suggestions were adopted or ignored by participants. The main contribution of this work is a study to characterize how software users make tool recommendations to peers.

4.1.2 Peer Interaction Characteristics

To analyze the effectiveness of peer interactions, we explored six characteristics of recommendations between partners in our user study: politeness, persuasiveness, receptiveness, time pressure, tool observability, and peer interaction type. These characteristics are motivated from research exploring the delivery of messages to humans in psychology and software engineering. Here, I provide examples for

each characteristic from prior work and present the criteria used to identify instances of each attribute in tool recommendations between participants in this study:

Politeness. Prior work suggests politeness is a key factor for making effective suggestions to humans. For example, Whitworth suggests many existing recommendation techniques, such as pop-up ads and the Microsoft Clippy recommender system, are ineffective and unpopular among users because it was impolite [Whi05]. In software engineering, researchers have explored incorporating politeness into sentiment analysis tools to observe programmers [DNM13], and show this concept can encourage developers to fix issues faster and increase work satisfaction in agile development teams [Ort15]. To measure politeness in peer interactions, we used Leech's six maxims for politeness: Tact, Generosity, Approbation, Modesty, Agreement, and Sympathy [Lee83]. The definition of these criteria and examples from the experiment can be found in Table 4.1.

Persuasiveness. Social psychology posits persuasion theory, or the study of the communication of messages to affect the attitudes and behavior of humans [Gar15]. Research suggests persuasiveness is crucial for making effective recommendations. For instance, O'Keefe argues "human decision-making is shaped by persuasive communication" [O'k02, p. 31]. Fogg also suggests persuasiveness is also necessary to convince users to adopt desired behaviors through software [Fog09]. For example, Faridi and colleagues propose integrating persuasion into the software development lifecycle to help reduce problems developers face while building software products [Far12]. Shen et al. introduce a generic model for developing persuasive messages including three features: Content, Structure, and Style [She12]. We used these criteria defined in Table 4.2 to measure the persuasiveness of recommendations between participants in this study.

Receptiveness. Psychology literature shows receptiveness plays an important role in the adoption of recommendations. For example, Feng argues receptiveness is necessary for humans to receive advice from others during problematic situations [Fen06]. Research also shows receptiveness is beneficial for adopting multicultural experiences and accepting ideas from foreign cultures [Leu10]. Prior work in software engineering has also explored using openness, one of the five personality domains [Dig90], to observe the practices of developers [Smi16]. Fogg argues receptiveness is vital for creating persuasive technology to persuade users to adopt beneficial behaviors online [Fog09]. To define receptiveness, he posits two criteria which are used in this study to measure this characteristic within peer interactions between participants: Demonstrate Desire and Familiarity. Definitions of these criteria and examples derived from our user study are presented in Table 4.3.

Time Pressure. Research from various disciplines shows time pressure impacts recommendations to humans. In behavioral economics, Kocher and colleagues suggest the allotted time to make decisions impacts the quality of choices because "time *is* money" [Koc06]. In marketing, studies show time constraints damage decision-making by stifling creativity and reducing exploratory thinking [And96]. Furthermore, software engineering research asserts time pressure from deadlines negatively impacts development practices [Cos84] and prevents peer interactions [MH15]. To measure time pressure in recommendations between participants in this study, we analyzed sessions to search for discussions about time between partners. Time limits were not strictly enforced in the study, but sessions lasted

one hour and we recommended spending 7–8 minutes on each question. If we determined a statement regarding time was made during a peer interaction, then the recommendation was categorized as being under time pressure. An example of the criteria used to measure time pressure is available in Table 4.4.

Tool Observability. Observability refers to whether or not systems consist of interfaces visible to users, which studies suggest influences the adoption of tools and behaviors. For instance, research shows visual attention, or the perceptual analysis of humans on various sensory features such as size, shape, and color, impacts the brands of products consumers purchase [Pie99]. Furthermore, Nielsen submits "Visibility of system status" as a usability heuristic for designing usable user interfaces [Nie93]. Murphy-Hill and colleagues suggest systems should have noticeable causes and effects to improve tool discoverability for software engineers [MH15]. To analyze this, we examined the observability of tools recommended between participants in the study. To evaluate the impact of the perception of tools on the outcome of peer interactions, we analyzed tools suggested between participants in our study to categorize them as Observable or Non-observable. Table 4.5 presents the definitions of these criteria and examples from the study.

Peer Interaction Type. Murphy-Hill and colleagues found peer interactions are the most effective mode of tool discovery among software engineers. They introduce two types of peer interactions, *peer observations* and *peer recommendations* [MH11], that differ in how suggestions are instigated between colleagues. Peer observation refers to when a developer views a colleague using an unfamiliar tool, while peer recommendations occur when colleagues notice coworkers completing tasks inefficiently and recommend a tool. Peer interactions were categorized as Peer Observations or Peer Recommendations based on the analysis of recommendations between participants.

4.1.3 Methodology

To observe these characteristics in peer interactions and investigate their effectiveness as a recommendation approach, I implemented a mixed methods approach to analyze using grounded theory and statistical data analysis techniques to examine the impact of each characteristic on observed tool recommendations between peers.

4.1.3.1 Data Collection

Participants

To evaluate peer interactions, we observed pairs of participants working together to complete data analysis tasks. Participants were students from various disciplines at North Carolina State University and professional analysts from the NC State Laboratory for Analytic Sciences[1] (LAS). For the remainder of this section, student participants are delineated with the S- prefix and LAS participants are presented with the L- prefix. More details about the user study participants can be found in Appendix A.1.7. Overall, 13 pairs completed the study, seven pairs of students and six pairs of professional analysts.

[1] `https://ncsu-las.org/`

Table 4.1 Definition and examples of the politeness peer interaction characteristic

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)

Table 4.2 Definition and examples of the persuasiveness peer interaction characteristic

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	*"Control-Shift-End"* (S1)
	Unpersuasive	*"I guess we're going to have to use some math calculations here, or a pivot table."* (L9)

Table 4.3 Definition and examples of the receptiveness peer interaction characteristic

Receptiveness Criteria		
Demonstrate Desire	Definition	User showed interest in discovering, using, or learning more information about the suggested 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	User 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)

Table 4.4 Definition and examples of the time pressure peer interaction characteristic

Time Pressure Criteria		
Time Pressure	Definition	Participant makes statement regarding time after a recommendation
	Yes	"*Yeah, that would work if we had time.*" (L5)
	No	No comments about time

Table 4.5 Definition and examples of the tool observability peer interaction characteristic

Tool Observability Criteria		
Observability	Definition	The ability to view the recommended tool through a graphical user interface
	Observable	"*Let's deploy a histogram...*[In Menu] *Insert, Recommended Charts...*" (S7)
	Non-Observable	"*Control-Shift-End*" (S1)

Tasks

Participants were asked to complete data analysis tasks based on the Kaggle Titanic data science competition.[2] The specific study tasks given to participants are available in Appendix A.1.1. The dataset for the tasks consisted of two comma-separated values files, TRAIN.CSV and TEST.CSV. The training tasks required participants to analyze the TRAIN.CSV file containing information about Titanic passengers to observe patterns and correlations in the data based on the survival of individuals. The testing task required participants to analyze the TEST.CSV files, which contained a different set of passengers and no details about their survival, and use their analysis from the preliminary tasks to predict whether individuals survived the shipwreck. While we provided solutions to the final tasks, the intent of this study was not to scrutinize the correctness or efficiency of solutions but to investigate how participants recommended tools to each other to solve the problems.

Study Design

Pairs of subjects were provided with one machine to work together to complete the tasks. The experiment machine was a Windows 10 laptop with several data analysis programs installed including Microsoft Excel 2016,[3] JMP Pro 12,[4] MySQL Workbench 6.3,[5] Python 2.7,[6] PyCharm,[7] R (command line and GUI),[8] and RStudio.[9] Participants were permitted to request and download additional free and publicly available software applications to analyze the data, however they were prohibited from using the Internet to complete the tasks to prevent looking up information about the problem and requiring participants to only rely on their own knowledge of tools to complete the tasks. Screen and voice recordings of participants completing the task were captured to further observe and analyze characteristics of tool recommendations between pairs.

Additionally, participants weer asked to provide qualitative data after sessions to further analyze peer interactions observed during studies. To debrief participants, we emailed a survey (students) and

[2] https://www.kaggle.com/c/titanic
[3] https://products.office.com/en-us/excel
[4] http://www.jmp.com/en_us/home.html
[5] http://www.mysql.com/products/workbench/
[6] https://www.python.org/
[7] https://www.jetbrains.com/pycharm/
[8] https://www.r-project.org/
[9] https://www.rstudio.com/

conducted a semi-structured interview (LAS) to ask subjects about tool recommendations that occurred during the study. The guiding questions for the semi-structured interview are available in Appendix A.1.3. Additionally, participants were asked to complete a demographic survey (Appendix A.1.6) to collect information about our study population.

4.1.3.2 Determining the effectiveness of peer interactions

Two independent researchers viewed the recordings from each study session to note instances of tool recommendations, categorize the recommendation based on our peer interaction characteristics, and determine the outcome of the recommendation. We iteratively define coding criteria for identifying and characterizing instances of peer interactions. Each recording was analyzed to determine:

- when a tool recommendation took place
- if the recommendation was effective
- if the recommendation was polite, persuasive, receptive, under time pressure, or made about observable tools

More details about the specific information collected for each peer interaction observed in this evaluation are provided in Appendix A.1.4. Below, I describe our process for identifying peer interactions, determining the existence of each characteristic, and evaluating the effectiveness of recommendations.

Identifying peer interactions

To identify peer interactions between participants, I developed a model to define tool recommendations based on the GOMS (Goals, Operators, Methods, and Selection rules) model in Human-Computer Interaction [Dia03]. This model, shown in Figure 4.1, outlines how we recognized peer interactions between two participants. Each node indicates a step required to denote an instance of a peer interaction. To describe this model, I use terms for developers in a pair programming environment where the user actively operating the keyboard and mouse is the driver and the peer observing is the navigator [Coc01].

During *task analysis*, both users consider the problem and develop a strategy to complete the task. *Task execution* refers to users discovering a mismatch between their task solution strategies. For peer observations, the navigator observes the driver completing the task using an unfamiliar tool. Peer recommendations occurs when the navigator notices deficiencies in the driver's approach and desires to suggest a better strategy. Finally, in the *dialogue* step a tool is recommended between the users. We transcribed recommendations to analyze the dialogue and determine the type of recommendation. For example, during peer observations the navigator may inquire about the driver's unknown tool. For peer recommendations, the navigator would suggest a tool to complete the task more efficiently or the may driver seek help. Two researchers independently analyzed the recordings of participant pairs completing the tasks and used this model to determine instances of peer interactions between partners.

Figure 4.1 Model to identify peer interactions between participants

Characterizing peer interactions

After identifying tool recommendations between participants, the two coders analyzed each peer interaction instance to determine if each of our peer interaction characteristics, politeness, persuasiveness, receptiveness, time pressure, tool observability, and interaction type, play a role in the effectiveness of recommendations. We used a valence scale to calculate scores for categorizing recommendations based on politeness (*polite, neutral, impolite*), persuasiveness (*persuasive, unpersuasive*), and receptiveness (*receptive, neutral, unreceptive*) according to our criteria defined for each trait (see Tables 4.1-4.3):

+1 Participant obeyed a specific characteristic criteria
0 Participant neither obeyed nor violated a specific characteristic criteria
-1 Participant violated a specific characteristic criteria

This scale was used by the coders to classify peer interactions individually, then we came together to discuss and resolve disagreements. A positive result indicates the existence of a characteristic, while a negative sum signifies the characteristic was not present. The definitions of the scoring criteria implemented to identify these characteristics, which we arrived at iteratively, is available in Appendix A.1.5. We used Cohen's Kappa to calculate the inter-rater agreement for politeness ($\kappa = 0.50$), persuasiveness ($\kappa = 0.28$), and receptiveness ($\kappa = 0.51$). A binary scale was implemented to measure time pressure (*time pressure, no time pressure*) and tool observability (*observable, non-observable*) based on the criteria in Tables 4.4 and 4.5 as well as the type of peer interaction.

Determining effectiveness

After identifying instances of software tool recommendations between participants, the two coders analyzed each interaction to categorize them as *effective, ineffective*, and *unknown*. Effective recommendations indicate the recommendee, or participant receiving the tool suggestion, used a tool after it was suggested by their partner for a majority of the relevant tasks. For ineffective recommendations, the recommendee mostly ignored the tool suggested by their partner when given the opportunity to apply it. Finally, since the study consisted of two participants working together on the same computer, there were cases of unknown recommendations where there was no opportunity for the recommendee to use the suggested tool for the rest of the study session. To resolve disagreements between the two coders, we watched the recording clip of the peer interaction instance in question together, each explained our reasoning behind our individual coding, discussed the rationale for each code, and came to an agreement.

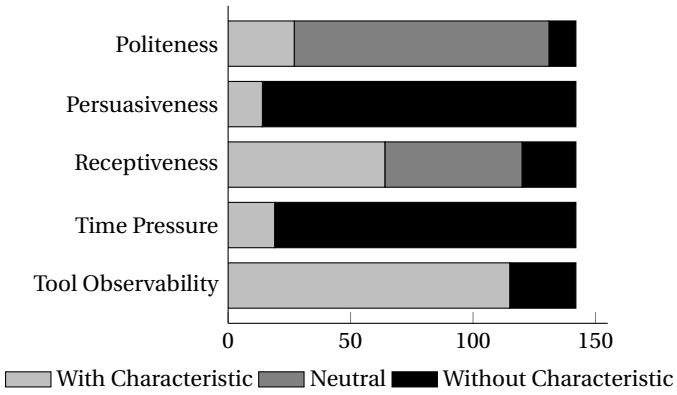


Figure 4.2 Peer Interaction Characteristic Results

4.1.4 Results

We identified a total of 142 peer interactions from our user study, categorizing 71 as effective, 35 as ineffective, and 36 as unknown. Study pairs averaged approximately 11 tools recommended between participants within each session. To quantitatively analyze the data collected from the experiment, the Mann-Whitney-Wilcoxon (W) test was used to evaluate ordinal data (*politeness*, *persuasiveness*, and *receptiveness*) and Pearson's chi-squared (χ^2) test was used to evaluate categorical data (*time pressure* and *tool observability*). All statistical tests were calculated with an alpha level of $\alpha = .05$ and odds ratios (OR) were used to measure effect size.

4.1.4.1 Characteristics

Each of the peer interactions observed between participants were categorized based on the peer interaction characteristics and their overall effectiveness. Figure 4.2 displays the number of recommendations that meet each of the characteristics for peer interactions we observed, and Table 4.6 displays the effectiveness of recommendations based on the peer interaction characteristics.

Politeness

Our analysis shows the majority of participants did not make polite recommendations according to our politeness criteria, classifying 27 tool recommendations between peers as polite, 11 as impolite, and 104 as neutral. Overall, while polite recommendations were more likely to be adopted than impolite ones ($OR = 0.6786$), politeness did not significantly impact the outcome of peer interactions (W, $p = 0.4936$).

Persuasiveness

Additionally, we discovered participants were rarely persuasive during peer interactions; there were only 14 persuasive recommendations in total while 128 were unpersuasive according to our study criteria. While prior work suggests persuasiveness is an important characteristic for convincing users

Table 4.6 Peer Interaction Effectiveness 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%

to adopt desired behaviors, this characteristic did not significantly influence the effectiveness of tool recommendations between participants (W, $p = 0.4556$, $OR = 1.4722$).

Receptiveness

We found most of the peer interactions in our study incorporated receptiveness, categorizing 64 as receptive, 56 as neutral, and 22 as unreceptive. Overall this characteristic had a high rate of effectiveness, with 61% of receptive peer interactions leading to the adoption of the recommended tool. Furthermore, we found receptiveness significantly impacts the outcome of tool recommendations between peers (W, $p = 0.0002$, $OR = 0.2840$). We expand on this finding in the Summary.

Time Pressure

Only 19 peer interactions observed between participants were categorized as being under time pressure. Similar to prior work, we found tool recommendations between peers without time pressure were more effective (52%) and more than twice as likely to be accepted by participants compared to those where time pressure was present ($OR = 2.2857$). However, this characteristic did not play a significant role in the outcome of peer interactions (χ^2, $p = 0.1470$).

Tool Observability

Observable tools were far more recommended during peer interactions in our study, with 115 recommendations compared to 27 non-observable tools. However, we found non-observable tool recommendations were slightly more effective (52%). Examples of observable tools recommended in our study include applications such as R and software features like Sort and pivot tables in Excel. Non-observable tools were primarily keyboard shortcuts. We found that the observability of tools did not significantly impact the effectiveness of recommendations (χ^2, $p = 0.4928$, $OR = 2.4060$).

Peer Interaction Type

In our analysis, we found that peer recommendations ($n = 89$) occurred more often than peer observations ($n = 53$). This indicates software users are more likely to make suggestions for tools and features as opposed to observing a tool and seeking information. Although tools recommended through peer recommendations are more likely to be adopted by users than peer observations (62%), this difference was not statistically significant (χ^2, $p = 0.3163$, $OR = 0.5729$).

4.1.4.2 Summary

Our results were unable to show that politeness, persuasiveness, time pressure, observability, and interaction type influence tool recommendations between peers. However, we discovered that receptiveness was the only characteristic to significantly impact the outcome of peer interactions. Thus, we conclude no matter how polite, persuasive, time-constrained, or visible a recommendation for a system is, users

will not adopt a tool unless they are receptive to using it. The receptiveness characteristic is also the most difficult to implement, since it solely depends on how recommendees receive and respond to suggestions, which recommenders cannot control. Our findings suggest peer interactions are effective because of their ability to foster user receptivity. To define receptiveness, we used criteria from prior work [Fog09] that suggests users must demonstrate ***desire*** and ***familiarity*** to be receptive to recommendations. We further expound on these criteria in the Discussion.

4.2 Naive *Telemarketer Design*

The naive *telemarketer design* is a basic approach for making automated recommendations to software engineers. This technique is referred to as a telemarketer design because behaves similar to a telemarketer that "calls" users to deliver static messages, never deviates from the script, and lacks the social context necessary to adjust messages, customize recommendations, or respond to questions and feedback. I developed this technique to define a baseline approach for recommending useful developer behaviors to programmers, such as static analysis tool adoption. With the naive *telemarketer design*, a system sends developers a generic message with information about a tool, provides a random example featuring a code snippet incorporating a common programming error irrelevant to the program, and provides the sample output from the tool given this vague error. To evaluate this naive design, I developed a simple bot to identify a baseline for making automated recommendations to software engineers and to better understand how developers respond to recommendations from automated systems.

4.2.1 Study Rationale

While peer interactions are the most effective method of tool discovery, Murphy-Hill and colleagues also discovered they occur infrequently in the workplace [MH11]. Furthermore, Turkle argues technology has become a substitute for face-to-face communications between humans [Tur17]. Thus, as peer interactions are in decline, it is becoming increasingly important to develop automated systems to recommend developer behaviors. Software engineering research shows bots are useful for automating a variety of programming tasks to improve developer productivity [Sto16]. However, studies also show bots can also be inconvenient and frustrating during interactions with humans [Hil15; Sta86]. To better understand the impact of bots on recommendations to developers, we evaluated the naive *telemarketer design* baseline approach in `tool-recommender-bot`, an automated system for making development tool recommendations to software engineers on GitHub. In this study, we examined the effectiveness of recommendations from `tool-recommender-bot` and gathered feedback from developers who received suggestions from thus system to better understand the impact of automated recommender bots and set the groundwork for designing better solutions in future approaches.

4.2.1.1 Research Questions

In this work, we explored the effectiveness of automated recommendations from bots by using the naive *telemarketer design* to discover:

RQ1 How well do bots encourage developers to adopt useful software engineering practices?
RQ2 How do developers respond to receiving recommendations from bots?

In this evaluation, the goal was to initiate and then identify reactions from developers to evaluate the naive *telemarketer design* recommendations in `tool-recommender-bot`. The results suggest bots with limited technical knowledge and generic recommendations are ineffective for influencing programmer behavior, and responses from developers provide insight into `tool-recommender-bot` was inadequate and implications for improving future systems. This work contributes the naive *telemarketer design*, a simple method that provides a baseline for designing automated recommendations, `tool-recommender-bot`, a bot that incorporates the naive *telemarketer design* to recommend static analysis tools to developers, to motivate the need for new automated recommendation approaches.

4.2.2 `tool-recommender-bot`: Implementing the naive *telemarketer design*

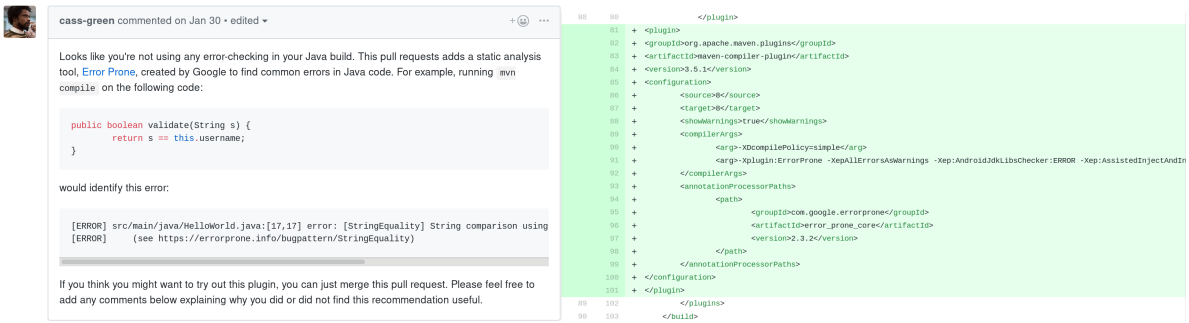
To evaluate the naive *telemarketer design*, I developed `tool-recommender-bot` to generate automated tool recommendations to developers. This bot integrates the naive *telemarketer design* by automatically making generic recommendations and adding static analysis tools on repositories using automated pull requests on GitHub. Pull requests are the preferred method to propose changes to GitHub repositories [Gitb], and prior work suggests automated pull requests are useful for upgrading out-of-date dependencies [Mir17a] and fixing static analysis tool violations [Car20]. The initial implementation of `tool-recommender-bot` in this evaluation naively recommends ERROR PRONE, an open source Java static analysis tool,[10] to developers on GitHub (see Figure 4.3a). To automatically integrate ERROR PRONE into projects, `tool-recommender-bot` adds the ERROR PRONE plugin to repositories that utilize the Maven automation and dependency management tool for Java applications by updating the Project Object Model (*pom.xml*) build configuration file to run the tool when the code compiles (Figure 4.3b).

Figure 4.4 provides a closer look at the automated pull request recommendation text from this system to show how the naive *telemarketer design* is implemented in `tool-recommender-bot`. First, the recommendation provides generic information about the ERROR PRONE Java static analysis tool (Fig. 4.4.A). Then, the bot also presents a simple example of a Java coding error, using the "==" operator to evaluate string equality instead of the `String.equals()` function (Fig. 4.4.B1), and provides the corresponding output from ERROR PRONE based on the given `StringEquality` error[11] (Fig. 4.4.B2). Ineptly, this simple error may not be present in the program and is irrelevant to the code base of project. An example pull request from our system using the naive *telemarketer design* on our repository can be found here[12] and is available in Appendix A.2.1.

4.2.3 Methodology

To observe the naive *telemarketer design* as a baseline for automated recommendations, I designed a mixed methods study to analyze the effectiveness of `tool-recommender-bot` recommendations

[10] `http://errorprone.info/`
[11] `http://errorprone.info/bugpattern/StringEquality`
[12] `https://github.com/CSC-326/JSPDemo/pull/2`



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

Figure 4.3 Example automated pull request from `tool-recommender-bot`

and collect feedback from developers to further evaluate this simple approach and gain insight into improving future automated recommender systems.

4.2.3.1 Data Collection

Projects

To evaluate the baseline naive *telemarketer design* approach, `tool-recommender-bot` sent automated recommendations to developers working on real-world software applications. The projects used for the evaluation were public open source software repositories on GitHub randomly sampled from the evaluation for Repairnator [Url18], an automated program repair bot.[13] To be eligible for this experiment, projects selected for the 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

Due to the fact that ERROR PRONE analyzes Java code, our evaluation was limited to projects written in that programming language. To determine projects that build with Maven, we checked to ensure repositories contained a *pom.xml* file in the highest-level directory and confirmed the project could be validated and compiled before adding the tool plugin. We also verified that projects did not already utilize ERROR PRONE by analyzing *pom.xml* files to make sure the ERROR PRONE plugin was not present to avoid making recommendations to projects that already use the tool and target developers less likely to know about it. Overall, we identified 52 projects that met these criteria to use for this study. The list of GitHub repositories used in the evaluation of the naive *telemarketer design* in `tool-recommender-bot` is available in Appendix A.2.2. These selected projects that received automated pull request recommendations from our bot varied in functionality, programming language, contributions, and size.

[13] `https://github.com/Spirals-Team/repairnator/blob/master/resources/data/results-buildtool.csv`

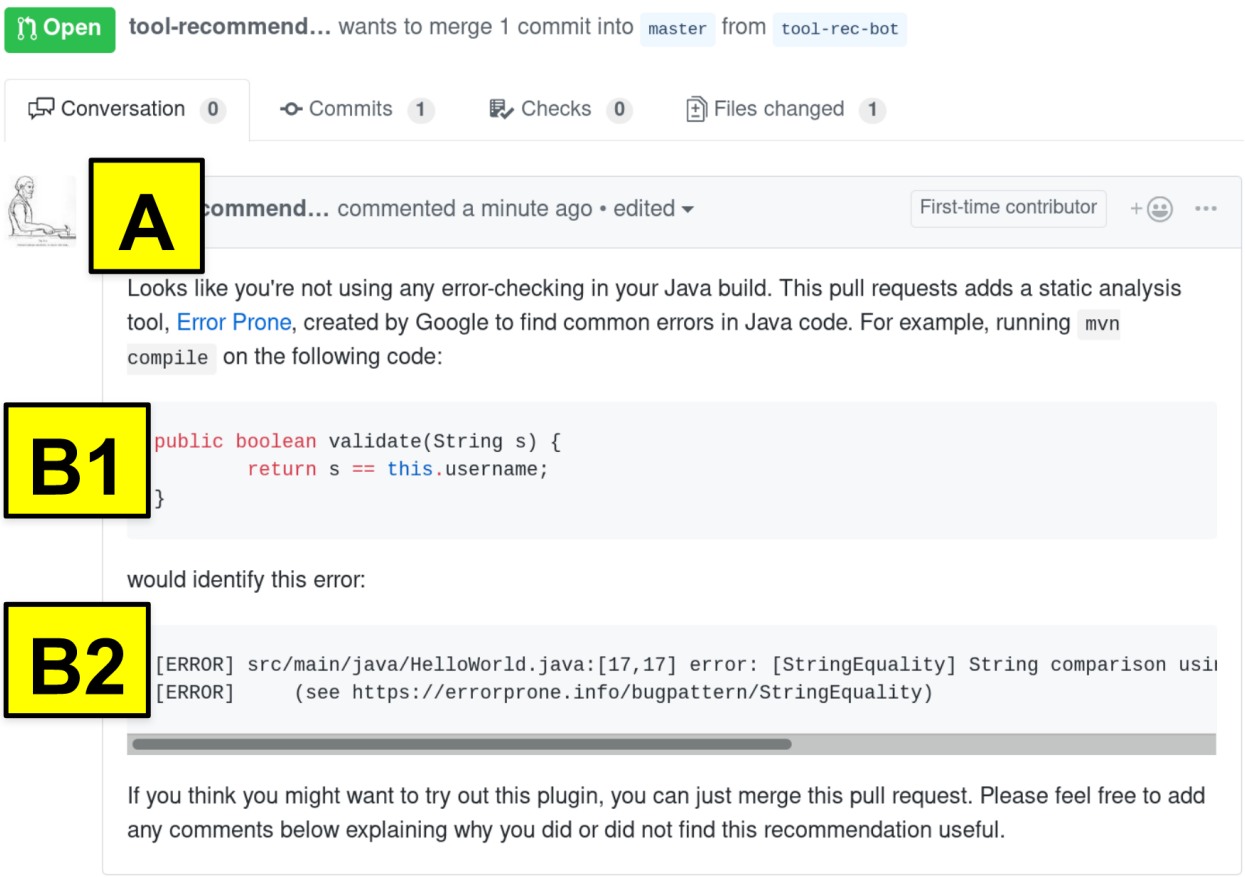


Figure 4.4 Naive *Telemarketer Design* recommendation

4.2.3.2 Determining the effectiveness of naive *telemarketer design* recommendations

To evaluate the naive *telemarketer design*, we categorized recommendations as *effective* and *ineffective* based on the status of automated pull requests from `tool-recommender-bot`. On GitHub, developers have the option to merge pull requests and incorporate them into the repository[14] or close pull requests without merging them.[15]. Additionally, developers can also ignore pull requests by leaving them open. For our evaluation, merged automated pull requests indicated an effective recommendation because developers showed a willingness to try ERROR PRONE and adopt the recommended tool into their repository by merging the changes from `tool-recommender-bot` into their code base. For example, in the pull-based software development model, merged pull requests indicate contributions from external developers are approved to be integrated into the source code for a repository [Gou14b]. Alternatively, closed or ignored pull requests implied the recommendation was ineffective. The naive *telemarketer design* pull request recommendations were monitored for one week to categorize the recommendations.

To assess the baseline naive *telemarketer design* automated recommendation approach, we cal-

[14] `https://docs.github.com/en/enterprise-server@2.22/github/collaborating-with-issues-and-pull-requests/merging-a-pull-request`

[15] `https://docs.github.com/en/enterprise-server@2.22/github/collaborating-with-issues-and-pull-requests/closing-a-pull-request`

culated the rate of effectiveness was calculated by measuring the percentage of merged pull requests out of the 52 `tool-recommender-bot` recommendations sent. Additionally, we aggregated comments from GitHub developers on pull requests to analyze how programmers reacted to receiving a naive *telemarketer design* recommendation on their repository. In `tool-recommender-bot` automated pull requests, our bot encouraged developers to provide feedback on recommendations by asking developers to "Please feel free to add any comments below explaining why you did or did not find this recommendation useful". This qualitative data was compiled and analyzed to determine how developers reacted to receiving naive *telemarketer design* recommendations from a bot and collect feedback on this simple approach.

4.2.4 Results

The `tool-recommender-bot` system sent 52 automated pull requests recommending ERROR PRONE to developers on GitHub using the naive *telemarketer design*. On these recommendations, we received a total of 24 comments from developers or other automated systems. To analyze the data collected, we calculated the merge rate of pull request recommendations from `tool-recommender-bot` and examined feedback from programmers to determine effectiveness.

Table 4.7 `tool-recommender-bot` Effectiveness Results

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

4.2.4.1 Recommendation Effectiveness

Our findings show that the naive *telemarketer design* is not effective for influencing developer behavior (see Table 4.7). In this evaluation, `tool-recommender-bot` was only able to make two successful recommendations out of 52 total notifications (4%). The remaining automated pull requests, 10 closed and 40 receiving no response from developers, resulted in an overwhelming 96% of `tool-recommender-bot` recommendations categorized as ineffective. Furthermore, while two recommendations from our system were merged, in one case a GitHub issue was created to report problems with the project build based on the changes by `tool-recommender-bot` and the pull request was reverted in a later pull request to remove ERROR PRONE from the project. Even though the tool was eventually removed, we still categorize this instance as an effective recommendation because the developers accepted the pull request and tried the tool before removing it.

4.2.4.2 Feedback

Overall, we observed 24 total pull request comments on 17 unique projects that received recommendations from `tool-recommender-bot`. Of the 24 comments, six were made by automated systems on the pull requests to provide information for first-time contributors, request Contributing License Agreement (CLA) signatures, or present code coverage updates. Thus, we received 18 responses from 15 individual developers on 14 recommendations, most of which was negative feedback. Additionally, of the recommendations that received feedback from developers through pull request comments ($n = 14$), 86% were ineffective and immediately rejected and closed by users ($n = 11$) or left open and never revisited by a developer ($n = 1$).

While we received some positive reactions from developers on `tool-recommender-bot` recommendations, even on those that were not merged (i.e. "*lgtm, Good Contribution*" (P9), "*Thanks for sharing it*"(P13)), the majority of feedback was poor (i.e. "Please stop using automated tools with a lack of understanding against random repos on github." (P6), "*Automated advertising is spam!*" (P8)). Overall, the analysis of developer comments posits the main criticisms from developers were about breaking builds ($n = 8$) and messing up the *pom.xml* formatting ($n = 5$). We further investigate this feedback to provide themes describing why the naive *telemarketer design* in `tool-recommender-bot` was ineffective.

4.2.5 Summary

Our results suggest simple bots are ineffective for influencing the behavior of developers. Most naive *telemarketer design* recommendations were ineffective, ignored or rejected by developers. This motivates the need for new design approaches and improvements to recommender bots to enhance recommendations and encourage developers to adopt better behaviors. Based on feedback from developers who received naive recommendations from `tool-recommender-bot`, we discovered the main drawbacks of the naive *telemarketer design* are a lack of ***social context*** and interrupting ***developer workflow***. These concepts are explained and further outlined in the Discussion section.

4.3 Discussion: *Developer Recommendation Preconditions*

Based on the results from these preliminary studies, we uncovered four concepts useful for designing automated behavioral recommendations to developers based on the efficacy of peer interactions and inadequacy of the naive *telemarketer design*: ***desire***, ***familiarity***, ***social context***, and ***developer workflow***. At a minimum, automated recommender systems must incorporate these *developer recommendation preconditions* in order to make convincing recommendations for software engineers. Below we explore each concept, providing definitions, examples from the completed evaluations, and illustrations from existing software engineering literature.

4.3.1 Desire

Demonstrating desire, or users expressing eagerness to adopt recommended tools and practices, led to effective recommendations between participants in the peer interactions user study. For example, in peer interaction observed during the study participant L12 recommended using the multi-level sorting functionality in Excel. Their partner, L11, demonstrated a desire to use this feature by responding "*Oh! Add level! Yes, awesome!*", and the multi-level sorting tool was adopted for completing the rest of the study tasks. Meanwhile, in another session one participant recommended using R for analyzing data to complete a task, but their partner responds "*No, no, no...*" (S14). This suggests recommending desirable tools and behaviors can increase adoption among developers, while a lack of desire can negatively impact the outcome of a recommendation.

Software engineering research also suggests desire impacts the adoption of activities and behavior by developers. For instance, Senyard and colleagues suggest desire is important for motivating programmers to contribute to and maintain successful open source software projects [Sen04]. Furthermore, Murphy-Hill and colleagues found that one barrier to the adoption of useful development tools and practices is *developer inertia*, which refers to when programmers are unwilling to switch from their current workflow and 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" [MH15]. Prior work proposes using history-based recommender systems to track user behavior and recommend desirable tools based on their activity [MH12b, p. 16]. To effectively recommend developer behaviors useful for completing programming tasks, recommendations must include desirable and advantageous tools and practices to encourage adoption by developers.

4.3.2 Familiarity

Another takeaway from the peer interactions user study is that users are more likely to adopt recommendations for well-known and recognizable tools and concepts. We observed comments from participants explicitly expressing familiarity and knowledge about recommended tools or their functionality. For instance, in one interaction when L8 recommended using the `COUNTIF` function in Excel, L7 was familiar with the feature and navigated to the menu to adopt the tool replying "*Yeah...here we go*". On the other hand, we found unfamiliar tools negatively impacted recommendations. For example, when a S10 proposed using R to complete a task, their partner responded "*I don't know R*" (S9), and the partner's unfamiliarity with R led to an ineffective recommendation. According to our results, recommending familiar tools can increase the effectiveness of recommendations for developer behavior. This points to a need to incorporate familiar concepts to developers when recommending beneficial development tools and practices.

There are several ways familiarity can impact behavior adoption. For example, familiarity can also lead to *developer inertia*, where programmers prefer familiar tools and processes over of adopting new systems. Prior work also suggests increasing employee knowledge about their workplace and environment, or *work familiarity*, improves their performance [Goo92]. In software engineering, research

shows familiarity impacts the completion of tasks and coordination among distributed development teams [Esp02], increasing code comprehension, or familiarity with code, improves development practices [Ko06], and unfamiliarity in the Eclipse[16] development environment led to disorientation and decreased productivity [DA06]. To incorporate familiarity, prior work propose using history-based systems ranking commands based on similarity using collaborative filtering [MH12b]. Additionally, existing recommender systems posit organization-wide learning [Lin00b] and collecting user history from networked workstations [Mal95] to suggest tools used by colleagues in similar circumstances.

4.3.3 Social Context

The results from the naive *telemarketer design* study show this approach was ineffective because of its lack of social context. This refers to the standard practices and community activities necessary to participate in software engineering by interacting with developers and contributing to projects. Examples of these activities include adhering to formatting and style guidelines, participating in code review discussions, and agreeing to CLAs. The most common complaint we received from GitHub users on `tool-recommender-bot` recommendations related to social context was that this system did not follow project style guidelines and disconfigured the whitespace of *pom.xml* files when automatically adding the ERROR PRONE plugin (see Figure 4.3b). For example, developers replied "*The automated tool you use messed up the pom.xml formatting to an extent that I could not see it*" (P5) and "*This change removes quite a lot if important things from the POM file*" (P7), even though `tool-recommender-bot` only added the ERROR PRONE plugin and nothing was removed from configuration files. This suggests our bot's inability to conform to the social context surrounding software development discouraged developers from adopting recommendations.

Social aspects of computing play a major role in making recommendations to developers. For example, prior work argues software engineering is a social activity [Ahm08], peer interactions between colleagues are the most effective mode of tool discovery [MH11], social media has changed the way developers learn about and share new information [Beg10; Sin14], and social factors influence the adoption of security development tools [Xia14]. Furthermore, research shows integrating into social context impacts the effectiveness of automated recommendations. For example, Wessel and colleagues evaluated the usage of bots in open source software and found that problems integrating into social context, specifically limited decision-making abilities and poor communication and feedback, were the biggest challenges developers face during interactions with bots [Wes18]. Prior work also found that systems emulating humans receive better responses from developers and are more effective than recognizable bot accounts [Mur16b]. Thus, effectively integrating recommendations into the social context of software engineering can improve the adoption of development practices and tools.

[16] https://www.eclipse.org/ide/

4.3.4 Developer Workflow

The second theme derived from user feedback on `tool-recommender-bot` was that the naive *telemarketer design* approach disrupted developer workflow, or the existing processes used by programmers to complete development tasks and deliver software. The most notable example of this was the fact that automated pull requests recommendations for ERROR PRONE often broke continuous integration builds for repositories (see Figure 4.5). However, adding a new static analysis tool to projects often introduced errors and caused the existing infrastructure to fail. 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`, saying the pull requests "*has introduced erroneous behavior to the build.*" (P10), "*Thanks for the contribution, but given the number of errors, I think it would cause more harm than good ;)*" (P11), and "*Your change itself looks good, but it seems CI's failing somehow*" (P12). Furthermore, P5 and P7 were both concerned about the impact on the overall build time. This inability to smoothly integrate into the workflow of developers prevented users from adopting naive *telemarketer design* recommendations from `tool-recommender-bot`.

Software engineering literature also notes the importance of integration into the workflow of developers. For example, research shows that developers at Google and Facebook primarily ignore static analysis tool warnings that are not integrated into their development workflow [Sad18; Dis19]. Additionally, Johnson and colleagues report the primary reasons software engineers avoid static analysis tools are due to their lack of customizability and poor integration into their existing processes [Joh13] while Rahman et al. show that considering the existing workflow of development teams influences software engineers' adoption of continuous integration and build automation tools [Rah17]. Furthermore, Tonder suggest successful integration of bots with human workflows is important for improving the effectiveness of program repair bots [Ton19]. To improve recommendations for developer behaviors, systems should suggest tools without breaking existing development mechanisms and easily integrate into the workflow of developers.

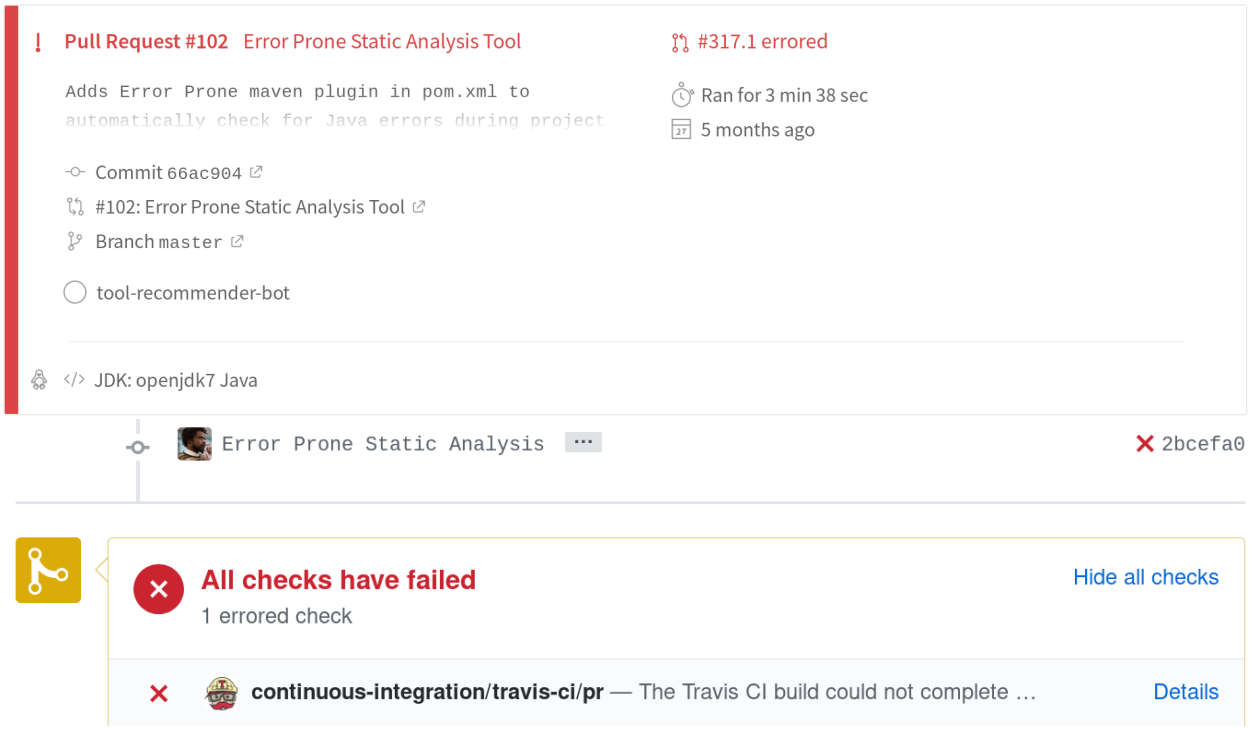


Figure 4.5 Example of `tool-recommender-bot` causing project builds to fail

CHAPTER

5

DEVELOPING THE CONCEPTUAL FRAMEWORK

The preliminary evaluations investigate what makes effective developer recommendations by exploring the effectiveness of peer interactions and the failures of the naive *telemarketer design*. Altogether, these studies posit four *developer recommendation preconditions*, or aspects necessary to make effective recommendations to programmers, **desire**, **familiarity**, **social context**, and **developer workflow**. However, as opportunities for peer interactions decline and bots produce inadequate suggestions, how can these prerequisites be incorporated into automated recommendations to encourage developer behaviors? This chapter introduces ***developer recommendation choice architectures***, a state-of-the-art approach to design automated systems by presenting desirable and informative recommendations to developers within their development environment and workflow.

5.1 Developer Recommendation Choice Architectures

Choice architecture refers to the organization of the context in which humans make decisions [Tha13]. To improve the decision-making of humans, Johnson and colleagues posit 11 practical tools for choice architects, or "anyone who present(s) people with choices", valuable for structuring decisions and describing options to encourage the adoption of target behaviors [Joh12]. In this work, I view software engineering researchers and toolsmiths are also *choice architects*, creating tools and practices requiring developers to make decisions while developing and maintaining software applications. Thus, I argue the presentation and organization of these decisions to developers impacts the choices they make in their work.

To further improve software engineering bots, I introduce ***developer recommendation choice architectures***, a conceptual framework to design automated recommendations from bots. This approach is motivated by the findings from the preliminary studies (Chapter 4), software engineering literature, and prior work in nudge theory. To devise this framework, I analyzed the tools for choice architecture and apply these concepts in a development context. This mapping, presented in Table 6.1, derived three principles for designing developer recommendations: ***actionability***, ***feedback***, and ***locality***. By incorporating these concepts into automated notifications, we can improve the way decisions are presented to developers and encourage adoption of useful tools and practices. For the remainder of this chapter, we provide definitions, motivation, and example for each principle and present a formative evaluation of this framework exploring actionable recommendations.

Table 5.1 Developer Recommendation Choice Architectures

	Choice Architecture Tool [Joh12]	**Definition**
Actionability	Technology and decision aids	Introducing technology to aid decision makers in choice tasks
	Use defaults	The way decision makers initially encounter choice tasks
Feedback	Reduce number of alternatives	Limiting the number of choice options presented to decision makers
	Focus on satisficing	Helping users consider outcomes that lead to higher choice satisfaction
	Attribute parsimony and labeling	Limiting the number of characteristics presented with options
	Translate and rescale for better evaluability	Presenting attributes to increase impact and clarity
	Customized information	Personalization to account for individual differences between decision-makers
	Focus on experience	Considering the background and knowledge of decision-makers
Locality	Limited time windows	Providing time restrictions for users to make decisions
	Partitioning of options	Groups or categories of options or attributes
	Decision staging	Dividing decisions into multiple stages

5.1.1 Actionability

Actionability refers to the ease with which developers can adopt behaviors presented in recommendations. Nudge theory research suggests actionability is a key concept for encouraging humans to make better decisions. For example, Thaler and Sunstein suggest 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" [Tha09, p. 85]. Similarly, Johnson suggests incorporating technology aids and using defaults are actionable ways to influence human behavior. For example, Madrian and Shea implemented the default rule to nudge employees to enroll in retirement plans. By having users automatically opt-in to 401k plans instead of requiring manual signing up, they discovered increased enrollment, with 98% of new employees selecting a plan within 36 months, and improved money-saving behaviors [Mad01].

In the naive *telemarketer design* evaluation, we found developers disapproved of recommendations from `tool-recommender-bot` because of their deficiencies integrating into development workflows and making more work for developers. For example, P3 commented "This introduces a bunch of errors, can you check whether they are worth fixing or configure the plugin so as to ignore the false positives?". Software engineering research also shows actionability is important to developers for adopting development tools and practices. For instance, 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 [Hec11]. Additionally, Evans and colleagues show that by automatically turning on security analyses in the SPLIT static analysis tool[1] increased the amount of security vulnerabilities fixed [Eva02]. We propose actionable development tool and behavior recommendations can increase adoption from developers.

5.1.2 Feedback

The feedback principle refers to providing clear and relevant information to developers. Sunstein and Thaler note "the best way to help Humans improve their performance is to provide feedback" and "choices can be improved with better and simpler information" [Tha09, p. 92, 204]. Johnson suggests practical techniques for improving feedback to decision-makers during choices such as limiting the number of options, presenting desirable outcomes, adding labels, reducing the attributes of choices, providing comprehensible content for choosers to evaluate, customizing information and messages, and relating to knowledge and experiences [Joh12]. Behavioral science research shows enhanced feedback on decisions improves human behavior. For instance, 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 customized recommendations for daily caloric intake encouraged consumers to purchase unfamiliar and healthier meals [Wis10].

The results from the peer interactions study suggest providing information about desirable outcomes and targeting familiar concepts of tools and behaviors can incorporate receptiveness into recommendations. Similarly, the naive *telemarketer design* study found generic and irrelevant recommendations

[1] `https://splint.org/`

from `tool-recommender-bot` were unproductive, violating social context, and respondents longed for details that "*would actually help*" and "*attach[ing] a report with actual findings in our code instead of just some generic example*" (P7). Software engineering research also suggests feedback to developers factors in influencing their behavior. For instance, Barik and colleagues examined the structure of compiler error messages on how developers resolved problems [Bar18]. Furthermore, Cerezo and colleagues suggest that *user-driven communication* can improve the perception of chatbots compared to bot-driven techniques [Cer19]. 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.

5.1.3 Locality

Locality refers to the setting of recommendations in the context of developers completing programming tasks. Johnson presents several tools for incorporating choice architecture into the setting of choices, including restricting the amount of time for users to make decisions, organizing options into groups, and dividing decisions into multiple states [Joh12]. Prior work studying RSSEs also suggests *when* and *what* to recommend are challenges for automated recommender systems [Hap08]. To describe locality in recommendations for developers, we divide this concept into two subcategories: *spatial* and *temporal* locality.

5.1.3.1 Spatial:

Spatial locality refers to the location where developer receive recommendations. Behavioral science research suggests the location of options matters when encouraging humans to adopt beneficial behaviors. For example, Hanks found that by changing the location of vegetables and fruits in a high school cafeteria, they found an increase in the amount of healthier foods purchased and consumed by students [Han12]. The preliminary studies also suggest location matters in recommendations. For example, in the peer interactions study a participant recommended the Find and Replace functionality in Excel and their partner responds "*Where's the find and replace?*" (S12), displaying their unfamiliarity with the feature.

Software engineering research shows the placement of decisions impacts the behavior of programmers, as developers prefer notifications located in convenient locations. For example, in a collaboration with Smith et al. we developed FLOWER, an Eclipse code navigation plugin created to help developers avoid disorientation by incorporating *in situ* design principles to prevent users switching between views during code search. We found the location of suggestions within the coding editor of the IDE led to increased efficiency with branchless navigation for developers to find security vulnerabilities and received positive feedback from participants [Smi17]. Thus, I propose automated recommendation systems for encouraging developer behaviors should situate suggestions in convenient and detectable locations to improve the decision-making of software engineers.

5.1.3.2 Temporal:

Temporal locality refers to the timing of recommendations made to developers. Nudge theory suggests timing of decisions influences human decision-making. For example, an effective nudge for farmers in Kenya was to change the time of year for fertilizer discounts, and this time-limited window encouraged them to make purchases earlier and improve the harvest of crops [Duf11]. In the naive *telemarketer design* study, developers mentioned the potential impact of adopting `tool-recommender-bot` recommendations on the timing of their project builds led to ineffective recommendations. For example, P7 desired information about "*the over head in terms of build time*". These naive recommendations also came at inconvenient times for developers who did not have the bandwidth to fix additional issues and breaking builds, such as one respondent who commented, "*Can you fix the errors reported by your tool in the build so that I can see the proposed changes?*" (P17).

Software engineering research also shows that the timing of recommendation within programmers' workflow is important for increasing adoption of developer behaviors. For example, Distefano examined configuring static analysis tools to run at *diff time*, or on code contributions submitted by developers during code review before being merged into the code base, and found that this rescheduling 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 overnight [Dis19]. Alternatively, untimely recommendations led to programmers ignoring code navigation strategies from Spyglass [Vir09]. To improve the effectiveness of automated recommendations, systems should make timely suggestions to programmers within their workflow to increase their desire to adopt useful software engineering behaviors and practices.

5.2 Preliminary Evaluation

To explore the impact of *developer recommendation choice architectures* on the behavior and decision-making of software engineers, I conducted a preliminary evaluation to provide an overview of actionable recommendations. While this work focuses solely on *actionability*, the subsequent studies and future work aim to study all of the conceptual framework principles.

5.2.1 Methodology

To evaluate actionability in automated recommendations, this study incorporates multiple methods of analyzing survey responses to quantitatively evaluate developer preferences and qualitatively collect feedback on actionable recommendations.

5.2.1.1 Data Collection

Participants

Professional software engineers were recruited to participate in this study. Overall, we received responses from 15 software developers and participants had an average of 7.3 years of industry programming experience.

```
9  if status is True:
10 -   print 'passed'
11 +   print('passed')
```

Listing 1 Example of a PEP 3105 static analysis violation and fix

Study Recommendation

For this evaluation, we presented participants with recommendations to fix a PEP 3105 Python 3 style warnings. This warning indicates Python `print` statements are are now `print()` functions in the latest version of the programming language.[2] For instance, line 9 of Listing 1 contains an example PEP 3105 violation. This simple recommendation also has larger implications for improving the behavior of developers because, as of January 1, 2020, Python 2 is officially no longer supported. The programming language announced there will be "no new bug reports, fixes, or changes" to the older version and encouraged developers to upgrade to Python 3.[3] Our sample recommendations proposed fixing PEP 3105 warnings and upgrading the code base to Python 3.

Survey

To investigate the impact of actionability in automated suggestions, we presented participants with sample automated recommendations to fix PEP 3105 warnings and upgrade to the latest version of Python. The survey presented participants with screenshots of two suggestions, one actionable and one static, and asked participants to select which recommendation they preferred and provide reasoning behind their choice. Additionally, we asked participants to input their years of professional development experience and to provide general feedback on designing automated recommendations from bots. The survey distributed to developers is available to view in Appendix B.1.1.

5.2.1.2 Determining the effectiveness of actionable recommendations

In this formative evaluation, we surveyed developers to investigate actionability in automated recommendations. The survey included actionable and static recommendations to fix PEP 3105 errors and a message encouraging users to upgrade from Python 2 to Python 3. The static recommendation presented to participants is displayed in Figure 5.1, while Figure 5.2 presents the actionable recommendation. Based on the *developer recommendation choice architectures* design principles, both of these recommendations incorporate the same *feedback* (information to promote repairing the PEP 3105 error, proposing a fix, and encouraging users to upgrade to Python 3), *spatial locality* (placed on Line 10 of a sample code snippet containing a PEP 3105 error), and *temporal locality* (located on an open pull request before the code is merged).

[2] `https://www.python.org/dev/peps/pep-3105/`
[3] `https://www.python.org/doc/sunset-python-2/`

Figure 5.1 Static recommendation to fix a PEP 3105 error

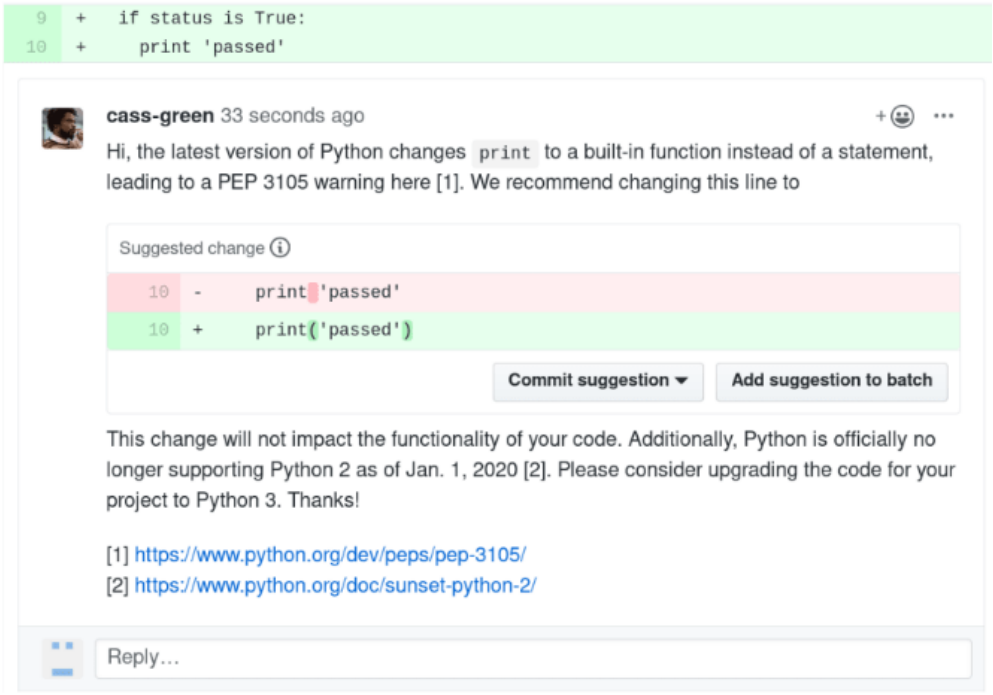


Figure 5.2 Actionable recommendation to fix a PEP 3105 error

However, these sample notifications differ on the *actionability* of recommendations. The static recommendation would require developers to re-submit a pull request to make the proposed change (`print('passed')`). On the other hand, the actionable recommendation incorporates a "Commit suggestion" button which allows developers to automatically commit the suggested fix for the `PEP 3105` violation to their code. This technology aid makes the decision of whether or not to fix the error simpler for developers by providing a solution and incorporating the ability to easily integrate the suggested code changes into their workflow and code. We aim to discover how this design decision impacts developers' preferences for adopting behaviors from automated recommendations.

5.2.2 Results

Our survey responses, presented in Table 5.2, reveal 100% of developers ($n = 15$) preferred the actionable recommendation over the static approach. This indicates developers are much more likely to adopt recommendations that make it easier to adopt suggestions. Developers also provided feedback praising the actionable recommendation, reporting it "*lets you automatically merge it*" (P8), "*appl[ies] the change automatically*" (P3), "*provide[s] an actionable short cut*" (P2), and "*can directly commit the change instead of having to do a manual commit*" (P10). Thus, we conclude that actionability is an effective approach for encouraging developers to improve their behavior.

Table 5.2 Survey Results on the Actionability of Recommendations

	n	**Percent**
static	0	0%
actionable	15	100%

5.3 Discussion

To improve developer decision-making, I present *developer recommendation choice architectures*, a conceptual framework which incorporates concepts from nudge theory to design actionable, informative, and convenient automated recommendations. This approach, motivated by the preliminary studies exploring effective developer recommendations as well as prior work in behavioral science and software engineering, posits *actionability*, *feedback*, and spatial and temporal *locality* as key factors influencing the adoption of developer behaviors. As choice architects, software engineering researchers and bot developers can enhance the way decisions are presented to programmers by integrating this framework into automated recommendation systems. To evaluate this approach, I conducted a formative evaluation investigating the impact of actionability on developers' perception of automated recommendations, and found participants significantly prefer actionable notifications in contrast to static ones. By incorporating this framework into automated recommendations, the thesis of this dissertation argues that systems can encourage the adoption of behaviors useful for improving code quality and productivity of

developers (Chapter 1). The next two chapters present further evaluations of *developer recommendation choice architectures* by analyzing existing recommender systems (Chapter 6) and introducing new tools incorporating this framework (Chapter 7) to analyze its impact on the decision-making and behavior of developers.

CHAPTER

6

ANALYZING EXISTING RECOMMENDATION SYSTEMS

Developer recommendation choice architectures is a framework to create automated recommendations improving how developers perceive and respond to suggestions by incorporating ***actionability***, ***feedback***, and ***locality***. The preliminary evaluation of this framework shows developers prefer actionable recommendations, however each principle factors into recommendations to developers and their overall impact on developer behavior remains unknown. To evaluate this approach, I first analyze the framework within existing recommender systems. This chapter explains how GitHub *suggested changes* adheres to the *developer recommendation choice architectures* principles and presents two studies analyzing *suggested changes* to explore their impact on the style and impact of recommendations. Additional study materials for these evaluations are available in Appendix C.

6.1 GitHub *Suggested Changes*

GitHub, a popular online code hosting site with millions of developers and billions of code contributions each year [Oct], introduced the *suggested changes* feature as a public beta release in October 2018. Since the announcement, the GitHub blog reports users have been "quick to adopt suggested changes" into their code review processes with over 100,000 uses within weeks of the initial public beta release, accounting for 4% of pull request comments and 10% of code reviewers during that time [Git18b]. This system, illustrated in Figure 6.1a-c, allows developers to make recommendations for code improvements to peers on GitHub during pull request reviews. The work presented in this chapter is the first research, to my knowledge, to study the GitHub *suggested changes* feature.

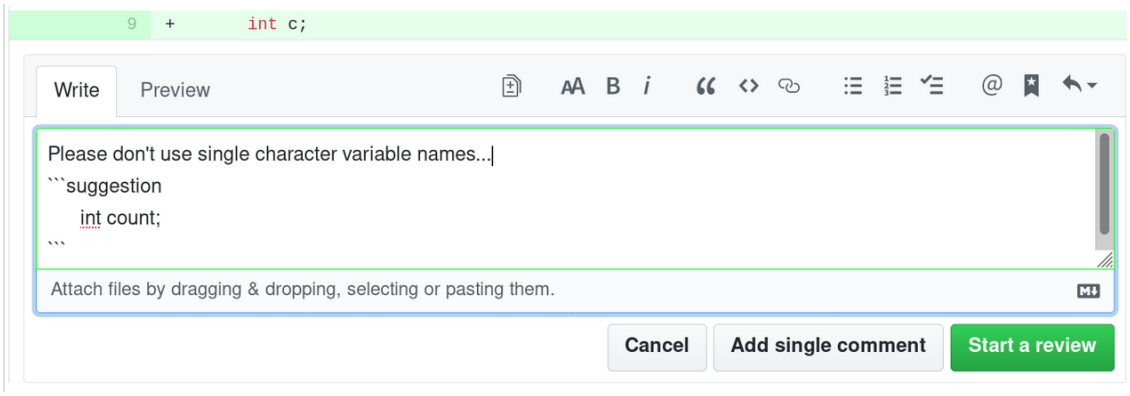
To use the GitHub *suggested changes* feature, a reviewer observes a deficient line of code on a pull request, they can click on the plus (+) sign on the line of code in question, in this case Line 9, to generate a pull request review comment (Figure 6.1a). Then, a text box is displayed for reviewers to enter a comment and they can click the GitHub *suggested changes* icon ( ) to propose changes to the line of code. Figure 6.1b presents an example recommendation, where the reviewer encourages the developer not to use a single character variable name, which is discouraged in the Java programming language for non-temporary variables,[1], and recommends changing the variable name from `int c` to a more descriptive identifier `int count`. Once the reviewer finishes their suggestion, they can click on the "Start a review" button to submit their recommendation. Finally, the developer who submitted the pull request can see the suggested change on their code, shown in Figure 6.1c, and has the ability to commit, edit, or ignore the proposed modification. Clicking on "Commit changes" provides the ability to automatically incorporate the proposed change into the pull request as a new commit.

GitHub *suggested changes* can also be considered a nudge, encouraging developers to improve their code without providing incentives to apply reviewer suggestions (i.e. money) and allowing alternative changes to improve the code (i.e. `int compute`). Additionally, this system incorporates all of the developer recommendation choice architectures presented in Chapter 5: it is *actionable* by providing the ability for developers to automatically apply recommendations from peers by clicking on a button to commit suggestions (Figure 6.1c); provides informative *feedback* to users by providing a specific improvement to the code with an optional comment (Figure 6.1b); has convenient *locality* with recommendations appearing to developers on the exact line of code within their pull request and during code reviews before contributions are merged into the code (Figure 6.1a). By evaluating the design of this novel feature, I aim to explore the impact of *developer recommendation choice architectures* on recommendations to developers within this system and provide implications for designing of future recommender bots.

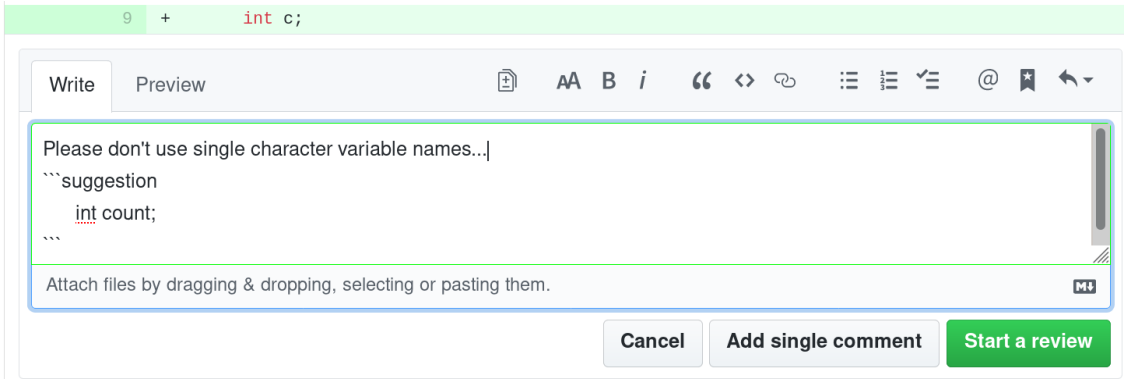
6.2 Recommendation Styles

Recommendation styles refers to techniques utilized by automated approaches for conveying developer behavior recommendations to programmers. Prior work suggests styles of suggestions can impact the decision-making and behavior of users. For example, Fischer argues active help systems that automatically provide help to users are more effective than passive approaches [Fis84]. Additionally, software engineering researchers have proposed a wide variety of recommendation tools and techniques to encourage developers to adopt useful practices with diverse manners of presenting information to users, such as badges [Tro18], Twitter [Sin14], software documentation [For02], live-coding [Bla14], crowdsourcing [Gor15], gamification [Bar16], idea gardening [Cao12], and Testing on the Toilet [MH19a].

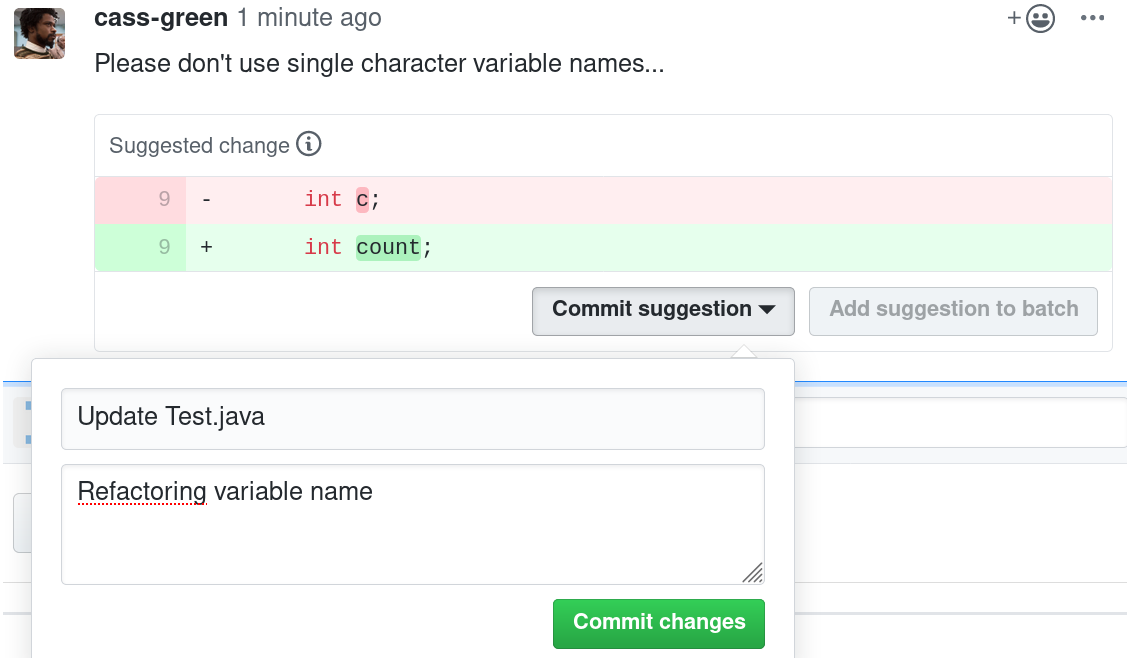
[1] `https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html`



(a) New suggested changes feature on GitHub pull requests



(b) Reviewer adds comment and suggested change to modified line of code



(c) Developer can apply suggested change and commit to PR

Figure 6.1 Example of the GitHub *suggested changes* feature

6.2.1 Study Rationale

To overcome decreasing opportunities for human-to-human recommendations and the increase of distributed development teams, researchers have explored creating recommender systems for software engineering to support programmer decision-making and improve the behavior of developers [Rob10]. However, studies such as [Hil15], and [Vir09] show that developers often find automated recommendations from systems ineffective. Additionally, the results of the naive *telemarketer design* study found that the simple recommendation approach in `tool-recommender-bot` failed because of its lack of social context and intrusiveness into development workflows. This work seeks to evaluate several system-to-human recommendation approaches, including the popular GitHub *suggested changes* feature, to discover their impact on the presentation of suggestions to programmers and gain insights into improving future automated recommendation approaches.

6.2.1.1 Research Question

To discover the impact of GitHub *suggested changes* as a recommendation system for automated recommendations, we sought to answer the following research question:

RQ How well does the suggested changes feature generalize to different styles of recommendations?

To answer this research question, I devised a user study that consisted of professional software engineers evaluating static analysis tool recommendations from four different systems: GitHub *suggested changes*, emails, GitHub issues, and GitHub pull requests. We analyzed these systems to evaluate the design of these features for making recommendations to software engineers. The results show that programmers preferred tool recommendations using *suggested changes* due to its content and design. The goal of this study is to discover the impact of this feature, and hence the *developer recommendation choice architectures* framework, on sharing information to developers and to provide implications for improving recommendations to software engineers. This study contributes a user study exploring different recommendation styles and a mock-up design for a future automated recommender system incorporating GitHub *suggested changes*.

6.2.2 Styles

To analyze the recommendation style of GitHub *suggested changes*, this work compares mock automated static analysis tool recommendations from this system to similar notifications given via emails, issues, and pull requests. Little is known about *suggested changes* and their impact on recommendations to developers, however the comparative systems examined in this evaluation were selected based on prior work exploring them as a mechanism for recommendations and their influence in software engineering. Additionally, I provide a breakdown of how each recommendation system analyzed during this study fit within *developer recommendation choice architectures* in Table 6.1 (to see how GitHub *suggested changes* support the framework, see Section 6.1).

Email. Email is one of the most popular forms of communication today, with approximately 4 billion users sending and receiving over 293 billion emails daily [Rij20]. Additionally, Sterne and colleagues suggest emails are the most powerful tool to reach audiences and spread information in marketing [Ste00]. Emails are also prevalent in software engineering, where prior work shows communication with emails in distributed agile development teams differs from face-to-face and instant messaging communication [Nii11] and proposes using emails to deliver security tool recommendations to developers [Jor14]. Furthermore, many systems, such as the Coverity static analysis tool[2] and GitHub security scans,[3] alert developers and provide reports via email.

However, emails generally do not fit into the *developer recommendation choice architectures* framework for designing recommendations for software engineers. For example, email systems usually have poor actionability and lack the ability for users to automatically apply suggestions made in recommendations. Additionally, this system has very poor locality, with email recommendations often presented in a separate location in an application outside of developers' programming environments and they can be received at any time during the software development process. These problems can lead to a variety of problems for workers, including email overload [Dab06] and reduced productivity from interruptions [Jac01]. Despite the shortcomings of email for actionability and locality, this system is able to provide clear and comprehensible feedback in suggestions to users depending on the content of the message text.

Issues. The GitHub issue tracker is a useful system for tracking a variety of information for repositories on GitHub [Gita]. For example, more that 20 million issues were closed by developers in 2019 [Oct]. Additionally, issues are another method for developers to make and receive recommendations on GitHub. Bissyandé and colleagues found that the majority of issues are labeled as "bugs", but argue those labeled as "feature" or "enhancement" that recommend improvements and new functionality to projects are "equally important for issue reporters" [Bis13]. Prior work has also observed issues to find correlations between issues and enhancements added to projects [Kri18] and analyzed issue labels to visualize activity in open source software repositories [Izq15].

GitHub issues have several characteristics that comply with *developer recommendation choice architectures* principles. For example, this system can provide understandable recommendations to developers in the title, description, or comments of issues. It can also provide further feedback with features such as labels to categorize the issue, milestones to group issues within the context of the project, and assignees to specify developers to complete the task.[4] For spatial locality, while issues do not occur within the code itself but they are present on the same project repository in a separate section. However, like emails, GitHub issues are not actionable, due to the fact developers cannot automatically implement recommendations from this system, and have poor temporal locality in that issues can be submitted to GitHub projects at any time during the development process. Research also shows GitHub issues can frequently go unnoticed or ignored by developers [Sor19].

[2] `https://scan.coverity.com/`

[3] `https://docs.github.com/en/github/administering-a-repository/managing-security-and-analysis-settings-for-your-repository`

[4] `https://guides.github.com/features/issues/`

Pull Requests. Pull requests provide a mechanism for developers to propose changes to projects on GitHub [Gitb]. We examined pull requests because they are the most popular method to recommend code changes to repositories. For example, in 2019 there were over 200 million pull requests submitted and 87 million merged into code repositories across the platform [Oct]. Research suggests pull requests are also useful for making suggestions to developers on GitHub. For example, Padhye and colleagues show recommending enhancements to projects is the most common purpose for pull requests submitted and merged into repositories [Pad14]. Additionally, prior work has explored generating automated pull requests to encourage developers to update package dependencies [Mir17a], fix static analysis errors [Car20], and recommend static analysis tools [Bro19].

GitHub pull requests closely adhere to the *developer recommendation choice architectures* framework principles. Developers are able to automatically merge proposed changes from pull requests into their source code, making it an actionable system. However, recommendations made to developers through review comments are not actionable and require users to manually apply suggestions. Pull requests can incorporate feedback by providing coherent suggestions to developers through the description of pull requests and the repository changes proposed to projects. Recommendations from this system also have high spatial locality, being submitted on the same GitHub repository with the ability to be integrated directly into the code base. Additionally, development teams usually have code review processes that provide a workflow for developers to inspect changes proposed in pull requests.[5] However, similar to issues and emails the timing of pull requests cannot be controlled by project maintainers, which means they can occur on repositories at any time during the development process. This may also factor in to pull request evaluation latency, or the amount of time for developers to address pull requests on repositories [Yu15], and contribute to reviewers' difficulty prioritizing contributions [Gou15].

Table 6.1 Mapping recommendation styles to *developer recommendation choice architectures*

	Actionability	***Feedback***	***Spatial Locality***	***Temporal Locality***
Emails	○	●	○	○
Issues	○	●	◑	○
Pull Requests	◑	●	●	◑
Suggested Changes	●	●	●	●

●Incorporates principle ◑Somewhat incorporates principle ○Does not incorporate principle

6.2.3 Methodology

To compare recommendation styles, I developed a user study using a mixed methods approach to collect quantitative and qualitative data from developers participating in an interactive think aloud study examining static analysis tool recommendations.

[5] https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-request-reviews

Table 6.2 Recommendation Styles Study Participants

Participant	Experience (years)	GitHub Familiarity	OSS Contribution Frequency	Tool Usage Frequency
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

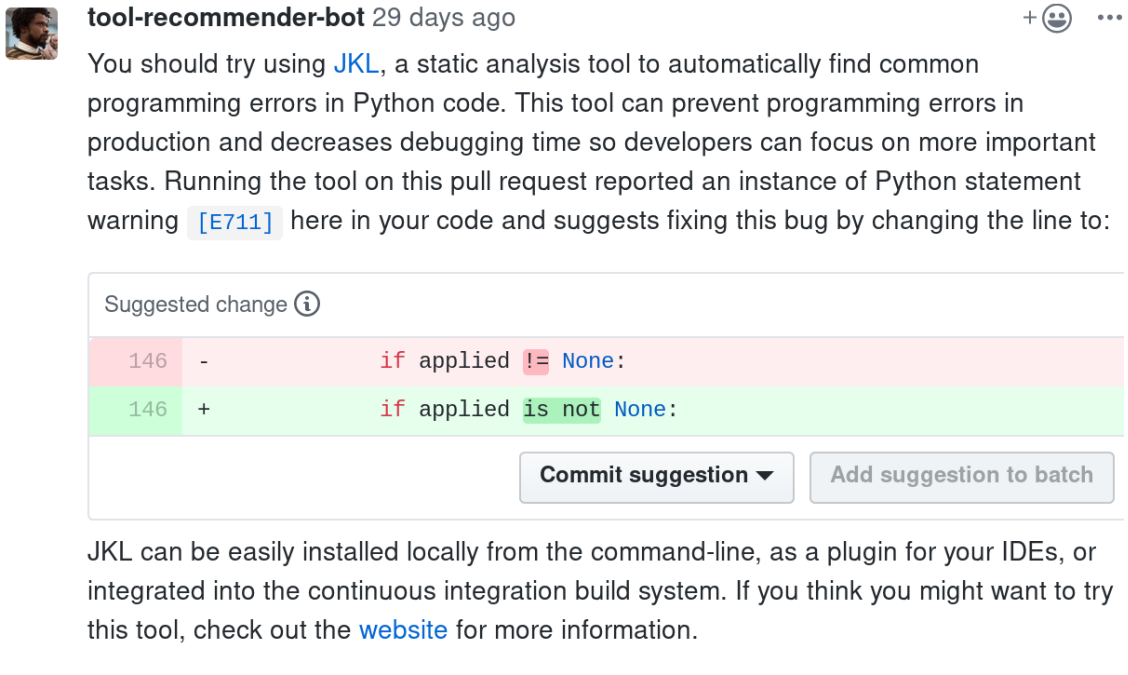


Figure 6.2 Example of the GitHub *suggested changes* recommendation style

6.2.3.1 Data Collection

Participants

We recruited 14 professional software developers, presented in Table 6.2, to participate in this study. Participants averaged 5 years of industry experience and consisted of workers from various companies spanning many different positions such as Software Engineer, Quality Engineer, Consultant, Data Migration Consultant, Support Specialist, User Researcher, and Technical Test Lead. All of participants were at least moderately familiar with GitHub and most had experience contributing to open source projects and incorporating development tools into their work.

Study Design

To collect data to answer the research question, we conducted a user study examining GitHub *suggested changes*, issues, pull requests, and emails as systems for static analysis tool recommendations. The study tasks involved these types of recommendations because static analysis tool adoption is a developer behavior that is beneficial for software development teams, however prior work shows developers often avoid these systems in practice [Joh13]. During the study, each participant evaluated recommendations from all four systems simulated in an experimental GitHub repository they were not familiar with. Participants were asked to interact with each recommendation system, as if they received it as an automated notification for their own project, and to use think-aloud methods [Jas04] to verbalize their thoughts as they explored each system. Then, to conclude the study we conducted semi-structured interviews to gather feedback from participants on each recommendation style and general opinions about tool recommendations.

6.2.3.2 Determining the impact of recommendation styles

To explore the impact of recommendation styles on developer behavior, participants interacted with tool recommendations from emails, issues, pull requests, and *suggested changes*. A sample GitHub *suggested changes* recommendation used for this study is available in Figure 6.2, while examples from the other systems are available in Appendix C.1.1. Recommendations from each system contained similar text recommending a static analysis tool that finds and prevents programming errors to participants, however suggestions differed in the presentation of recommendations according to each system. For example, GitHub *suggested changes* incorporates all of the *developer recommendation choice architectures* principles (Section 6.1) while emails have poor *actionability* to adopt recommendations and unfavorable *locality* outside of the development environment and workflow. To avoid bias, tools were recommended with made-up names (`ABC`, `DEF`, `GHI`, and `JKL`) and varying programming languages (JavaScript, Java, and Python) to prevent participants from relying on their previous experience with existing tools and programming languages. The order participants interacted with each system was also randomized to prevent order bias in our results.

Each study session was screen and audio recorded, and the semi-structured interview responses were transcribed to analyze developers' interactions with the systems. During the interview, the moderator asked participants to provide a five point Likert-scale rating on how likely they would adopt the tool recommended from each system. A score of four or five indicated participants were likely to accept the suggestion while a one or two indicated they were unlikely to adopt the tool and would reject the recommendation. Additionally, we asked subjects to discuss what they liked and disliked about each approach and to provide general observations on automated recommendations. Two researchers performed an *open card sort* [Beg14] to extract themes from open-ended responses and provide insight for improving recommendation systems to developers. The raters individually grouped statements based on responses concerning effective and ineffective recommendations, then came together to analyze and discuss the derived themes and sort the data accordingly.

6.2.4 Results

To analyze the data collected from our user study, we averaged Likert scale ratings and used the Kruskal-Wallis statistical test to calculate the likelihood of adoption and measure developer preferences. We also present the themes derived from the open card sort providing feedback on each system and general comments on developer recommendations.

6.2.4.1 Likelihood of Adoption

The findings from the user study show that GitHub *suggested changes* were the preferred tool recommendation system by participants. Table 6.3 presents the average and median Likert scores representing participants' reported likelihood they would adopt the tool recommendation from each method. The *suggested changes* feature had the highest score averaging a 4 in the 5-point Likert scale ranking for all participants. Additionally, this system had the highest rate of participants likely to adopt the tool presented in the recommendation (85%), and it was the only method that did not receive a 1 score. We also found that participants' preference for tool recommendations with GitHub *suggested changes* was statistically significant ($H = 16.7527$, $p = .00079$, $\alpha = .05$) compared to the other systems. This indicates the style of *suggested changes* is preferred by software engineers for receiving recommendations, and this tool has the style of an effective system for improving developer behaviors, such as increasing static analysis tool adoption.

Table 6.3 Survey Results on the Likelihood of Recommendation Style Adoption

	Average Score	***Median***
Suggestions	4	4
Pull Requests	3.71	4
Issues	2.86	3
Email	2.36	2

6.2.4.2 Qualitative Feedback

During the user study, we collected qualitative data through think aloud and a semi-structured interview to learn what developers liked and disliked about each system and gain insight into designing automated recommendation systems. Here, we present comments from participants on each of the different suggestion mechanisms and provide the themes derived from the open card sort on general recommendations.

Emails

The majority of participants were unlikely to adopt static analysis tool recommendations via email, ranking this system as a 1 or 2 ($n = 11$). Most developers also provided unfavorable feedback on receiving email recommendations, such as "*I hate emails*" (P3), "*if this came across unsolicited I would feel sort of intruded upon*" (P4), "*would honestly be pretty suspicious when I get any email asking to install software on my computer*" (P12), "*if I see an email about something it actually gives me less of a view of it*" (P6), and "*I'd immediately delete it...I wouldn't even give it a look. I'd actually probably not like that tool even more just because their sending out spam emails*". However, while most participants disliked tool recommendations sent by email, three participants, P2, P6, and P11, responded they were likely to adopt tools recommended through this system noting "*it feels personal*" (P2) and "*I like email more*" (P11).

Issues

Participants were noncommittal on their likelihood to adopt static analysis tool recommended through GitHub issues, with most participants ($n = 9$) scoring them as a 3. The primary feedback provided from developers on this system was the amount of effort required to learn more information and integrate the tool. For example, P1 noted "*I'd be much less likely to integrate it [the tool]...that's a lot of work*", P4 stated "*I see this as a big time sync to go through and evaluate how many of those actually are issues and how many are false positive things*", and P14 desired more specific information adding "*it reads a little more spammy without a code example...It seems like you could just post this message on any project. Why is this useful for my project specifically? I have no idea*". Additionally, P13 complained about the amount of text within the issue recommendation commenting "*this one is a ton of words*".

Pull Requests

Developers reported being likely to adopt static analysis recommendations from pull requests on GitHub ($n = 9$). Participants appreciated the timing and location of recommendations from this system on repositories. For example, P7 stated "*I'd be significantly more likely to try it if I already have a pull request that has all the changes I need to get the tool or something going in the project*" and P4 added "*getting pretty quickly an explanation, the actual issue in the code, and you know a basically free way to incorporate that into the process as well as you know the tool itself*". Alternatively, participants criticized the lack of information provided about tools provided in pull request suggestions, saying "*it doesn't really outline the steps*" (P13). P8 and P14 also criticized using pull requests on GitHub as a recommendation system, outside the intended purpose of this feature, stating pull requests should "*solve a problem rather than tell people they should try to use this thing*" and "*it's not that they really want to do a pull request. It's not that their going to be adding to the project...I don't know why it's a pull request specifically*".

Suggested Changes

Our results show participants were significantly more likely to adopt static analysis tool recommendations from *suggested changes* than from emails, issues, or pull requests. Overall 12 developers responded

they were likely to accept the recommendation, providing a variety of reasons why they favored *suggested changes* recommendations. Participants praised the location of the recommendation ("*having it comment right here, 'This fixes your bug!' That's nice*" (P8)), the visual aids of the feature ("*this one is definitely better because it's more visual*" (P13)), and the content of the information provided ("*it has a little more detail*" (P9) and "*this has very good detail, very detailed change of the code and so it's clear*" (P10)). Only one participant (P6) was unlikely to adopt the tool recommended with this feature, stating "*putting a tool recommendation in a comment of a pull request, it's kind of to me out of place*".

General Feedback

In addition to collecting feedback from developers on each recommendation system, we asked participants to provide general insight into designing effective recommendation systems. The open card sort derived five themes for what developers seek in recommendations. Below we present the categories uncovered in our qualitative analysis and provide representative quotes from participants:

Examples: Developers in our study noted examples are important to include in recommendations to support decisions on the adoption of tools. For example, participants specifically mentioned wanting to observe tool usage and output, view demos, and test systems in a local environment. Additionally, respondents desired the ability to easily find and access examples of tools in the documentation and/or on it's website.

> *"In general I think showing an example of the type of error that it would find, cause that immediately shows some value, I think that helps a lot"* (P4)

Integration: Participants indicated information about how well tools integrate into their existing processes should be incorporated into recommendations. Workflow details such as how easy it is to install the system, how well it works with other tools (i.e. continuous integration build systems), the impact on resources (i.e. memory, GPU, lagging, etc.), how relevant it is to their project and development needs, and if it adheres to company policies are important details for developers to consider when deciding to adopt tools from automated recommendations.

> *"I want something that I can install it and use it as quickly as possible with as minimal fussing with it and setup as possible."* (P5)

Marketing: Several participants remarked recommendations with text similar to advertising or marketing notifications would deter them from adopting tools from automated suggestions. For instance, developers in our study were suspicious and distrustful of static analysis tool suggestions from emails and unsolicited messages, but were more open to recommendations with human-like communication.

> *"I'm not sure how the bot would generate the text to do the recommendation but try to make it seem a little more human? Rather than it was written by someone in an advertising agency or something."* (P7)

Popularity: Participants reported providing information about the popularity of recommended systems into suggestions is valuable for helping programmers to decide whether or not to adopt development tools. Specifically, developers mentioned seeking information about the usage and reputation of tools from online reviews, meetup groups, colleagues, conference talks, social media, and other resources before incorporating systems into their workflow.

> *"Try to highlight the popularity, popularity is so crucial...I care about the adoption. The current adoption, that's a testimony of the strength"* (P13)

Reliability: Developers noted information about the reliability of tools and their sources is useful for determining whether or not to adopt them. For instance, study participants mentioned reliable tool output, little to no false positives, predictable behavior, up-to-date documentation, and ongoing development on the tool itself impact their decision-making.

> *"[It] definitely needs to work reliably. Any time a tool starts doing things like it occasionally has problems that's something that makes me want to stop using it...I want the tool to be more reliable than that."* (P8)

6.2.5 Summary

Our results show that GitHub *suggested changes* are an effective system improving the developer behavior. We found 14 professional software engineers were significantly more likely to adopt automated static analysis tool recommendations from this feature over suggestions from emails, GitHub issues, and pull requests. Further analysis of responses from developers derived themes for improving automated recommendations to improve the behavior software engineers. To provide implications for improving future recommender systems, we combine these themes into two overarching categories for designing effective automated suggestions: recommendation ***content*** (*Integration*, *Popularity*, and *Reliability*) and ***design*** (*Examples* and *Marketing*). In the discussion, I present how these categories align with *developer recommendation choice architectures* and contribute to the preference for the GitHub *suggested changes* recommendations.

6.3 Developer Impact

While the recommendation styles study shows users prefer to receive recommendations from systems incorporating *developer recommendation choice architectures*, their *developer impact*, or influence on programming practices, is still unexplored. For example, prior work suggests bots are useful for automating programming tasks and supporting software development activities such as completing continuous integration tasks, running tests, and decreasing code review time and effort in open source software projects [Wes18]. However, our prior work shows that naive *telemarketer design* recommendations from `tool-recommender-bot` negatively impacted development practices by breaking continuous integration project builds, aimlessly introducing static analysis errors to projects, and violating project

style guidelines [Bro19]. To evaluate the impact of the conceptual framework on development practices, I conducted an empirical evaluation of the *suggested changes* feature to analyze its influence on programming activities and its usefulness for developers on GitHub.

6.3.1 Study Rationale

The GitHub platform has a variety of methods for developers to make recommendations to each other, such as issues which are useful for recommending new functionality and enhancements to projects [Bis13]. GitHub recently introduced the *suggested changes* feature, a novel recommendation system on the platform that allows developers to provide specific and actionable feedback to peers during reviews (see Figures 6.1a-c) [Git18a]. While this feature is increasing in popularity and usage on GitHub [Git18b], little is known about how developers use *suggested changes* to make recommendations to each other and its impact on development practices. Researchers have performed in-depth analyses on a variety of GitHub features to explore their impact on developer behavior and software engineering practices, including pull requests [Gou15], issues [Bis13], issue tracker labels [Cab15], links between issues and pull requests [Li18], repository forks [Jia17], commit comments [Guz14], README files [Pra19], stars [Bor18], and badges [Tro18]. This study adds to this work by offering an empirical analysis of GitHub *suggested changes* to examine their impact on development processes and provide implications for improving future recommender systems.

6.3.1.1 Research Questions

To explore the impact of *suggested changes* on development practices, we investigated the following research questions:

RQ1 What types of recommendations do developers make with suggested changes?
RQ2 How effective are recommendation systems on pull requests?
RQ3 What impact do suggested changes have on pull requests?
RQ4 How useful are suggested changes for recommendations between developers?

To answer these questions, I contrived a multi-methodological study divided into two phases. The first phase seeks to answer the first three research questions by mining GitHub repositories and empirically analyzing the *suggested changes* feature to categorize types of suggestions, measure the effectiveness of this system, and observe its impact on the pull-based software development model. The second phase explores the final research question by collecting qualitative data from developers to provide feedback on their experiences with the GitHub *suggested changes* system. The results show *suggested changes* are effective for recommending a variety of changes to programmers and supports developers in the code review process by helping them make suggestions and decisions on proposed changes quicker. Users also find this feature useful because it allows clear communication and easy integration. The main contribution of this work is the first study to empirically analyze the GitHub *suggested changes* feature.

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

The first phase of this evaluation mines public GitHub repositories to explore the usage, effectiveness, and impact of *suggested changes*.

6.3.2.1 Data Collection

Identifying suggested changes

To automatically detect GitHub *suggested changes* on pull requests, I developed a script to programmatically parsed pull request review comments to find instances of the ``` `suggestion` ``` tag in a review (see Figure 6.1b) since this feature is currently not supported by the GitHub API.[6]. This markdown snippet indicates a recommendation was made from a review to a developer on a pull request using this system. The script utilized the PyGithub API[7] to collect repositories for identifying *suggested changes* in on updated pull requests and parsed comments to collect instances of the GitHub *suggested changes* feature. To limit our data collected, we only observed activity on pull requests after October 2018 when the *suggested changes* feature was first introduced on GitHub. For categorizing *suggested changes*, we randomly sampled the most recently updated repositories to compile a list of current pull request comments incorporating this feature. To determine the effectiveness of *suggested changes* and their impact on development practices, we analyzed the top-forked repositories on GitHub with pull requests using this feature. For the first research question, we randomly sampled 100 *suggested changes* on the most recently updated pull requests. To answer RQ2 and RQ3, we analyzed a total of 51,250 pull requests, 17,712 *suggested changes*, and over 152,030 pull request review comments on the most forked repositories.

6.3.2.2 Categorizing suggested changes

To categorize GitHub *suggested changes*, two researchers performed an *open* coding qualitative analysis on a sample of suggestions. We analyzed instances of this feature on the most recently updated repositories to characterize types of changes developers recommend. The raters independently analyzed *suggested changes* to inspect the code change recommended, the review comment text, and the entire discussion on pull requests to identify categories of changes proposed to peers by developers with this feature. Then, we came together to discuss our derived categories and come to an agreement. Finally, we used our defined categories to classify 100 randomly sampled instances of pull request comments containing the ``` `suggestion` ``` tag (inter-rater agreement = 71%, Cohen's $\kappa = 0.5942$). The list of sampled *suggested changes*, derived from projects varying in size, programming language, number of contributions, and other repository metrics, are available in Appendix C.2.1. Our main source of discrepancies arose from determining the functionality of suggested changes and deciding if suggestions were correcting or improving lines of code.

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

[7] `https://pygithub.readthedocs.io/en/latest/`

6.3.2.3 Determining the effectiveness of suggested changes

To investigate the effectiveness of suggestions, we analyzed GitHub *suggested changes* and pull request review comments, two mechanisms for providing recommendations to developers on code contributions. Pull request review comments differ from general pull request and issue discussion comments because they are situated on exact lines of code during the review process. Furthermore, reviewers can suggest specific changes to developers in pull request review comments by using code fences (```) to start and end the proposed block of code. For example, Figure 6.3 presents an instance of a review comment providing the same recommendation to change `int c` to `int count` as the *suggested change* in Figure 6.1c.

GitHub *suggested changes* are a special instance of pull request review comment, allowing reviewers to provide feedback on contributions from developers. The primary difference between *suggested changes* and review comments is the user interface, which involves the "Commit suggestion" button to automatically apply recommended changes to pull requests and the highlighted *diff* format to display suggested code modifications in the newer system. To evaluate the effectiveness of code recommendations on pull requests, we compare and contrast both of these features to analyze how they impact recommendations between developers during reviews. To measure effectiveness, the evaluation observed the *acceptance* and *timing* of recommendations from these systems. We further divide these criteria into subcategories: *contribution acceptance* and *recommendation acceptance*, and *contribution time*, *recommendation time*, and *recommendation acceptance time*.

Acceptance

Acceptance refers to users incorporating proposed changes from another developer. *Contribution acceptance* involves how frequently proposed changes from developers are incorporated into projects. Prior work suggests accepted contributions from developers are a valuable metric for measuring the success and performance of software on GitHub [McD13] and is necessary for the development, maintenance, and evolution of open source software [Mid18]. We use this metric to evaluate the adequacy of recommendation systems on pull requests. To study the impact of the recommendations on contribution acceptance, we calculated the merge rate of pull requests containing a code suggestion, either *suggested changes* or pull request review comments with fenced code, and compared it to pull requests without code recommendations.

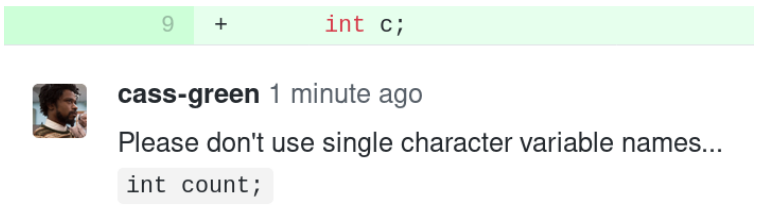


Figure 6.3 Example pull request review comment with code

Recommendation acceptance refers to how often suggestions from each system are approved by developers. The percentage of *suggested changes* and fenced code review comments incorporated into pull requests from our dataset was used to investigate recommendation acceptance. To identify instances of each system, we developed a script to analyze pull request review comments and search for occurrences of ``` suggestion to identify *suggested changes* and ``` to identify review comments with code. The recommended code within the tags was automatically extracted from the comment. Then, we programmatically checked whether the recommendation was accepted by determining if the extracted code existed in subsequent changes to the pull request by analyzing additional commits to the file after the recommendation was made. If so, we consider the suggestion accepted by the developer. This process was implemented to compare the recommendation acceptance for pull request review comments with fenced code and *suggested changes*.

Time

This metric examines the impact of pull request recommendation systems on the overall development process. *Contribution time* refers to the lifespan of pull requests submitted by developers. Prior work outlines factors that influence pull request evaluation latency, or the amount of time to review pull requests on GitHub [Yu15] and analyzes factors that influence merge time of pull requests [Gou14b]. Furthermore, research shows the longer it takes to repair issues in code the more expensive and difficult they are to fix [Lay07]. Here, we aim to determine if the presence of code suggestions from review comments or *suggested changes* impacts amount of time to make decisions about contributions from developers. The difference between time of a pull request being opened until when it is merged into the repository by a project maintainer was used to measure contribution time.

Recommendation time evaluates the amount of time for developers to make recommendations to peers with a given pull request recommendation system. To analyze the impact of *suggested changes* and pull request review comments with fenced code on development time, we compared how quickly developers compose comments making recommendations with with code on pull requests. This was measured by calculating the amount of time from the creation of a pull request until a reviewer adds a code suggestion with the ``` suggestion or ``` tags and comparing this recommendation time between each system.

Similarly, *recommendation acceptance time* refers how long it takes developers to accept recommendations with *suggested changes* and review comments with fenced code from reviewers on pull requests. To assess this, we measured the amount of time between a reviewer commenting on a pull request using the ``` suggestion or ``` tags until the time of a subsequent commit containing the extracted code between the fences. This data was used to compare the effectiveness of GitHub *suggested changes* and pull request review comments with code as mechanisms for receiving recommendations from peers on pull requests.

6.3.2.4 Evaluating the impact of suggested changes

To analyze *suggested changes* on development practices, we explored the impact of this system on the pull-based software development model, a popular programming methodology for distributed development teams on code hosting websites like GitHub [Gou14b]. Each year millions of pull requests are contributed by developers and merged into repositories on GitHub [Oct], excluding other online development collaboration platforms such as GitLab[8] and BitBucket.[9] Pull requests are the primary method for developers to contribute to projects [Gou15], and thus a mechanism through which developers provide recommendations to each other virtually. For instance, pull request review comments and *suggested changes* allow users to provide feedback and make recommendations on code contributions to developers. To further evaluate GitHub *suggested changes*, we analyzed metrics from existing literature to explore the effect of this feature on pull request timing, coding activity, and collaboration.

Pull Request Characteristics

Prior work posits a variety of metrics derived from the *pullReqs* dataset to explore the pull-based software development model [Gou14a]. This dataset, consisting of roughly 350,000 pull requests on 900 GitHub repositories, provides a variety of characteristics to describe GitHub pull requests and quantify their impact on the pull-based development model. To explore the impact of GitHub *suggested changes* on development practices on GitHub, we calculated the following metrics to analyze pull requests with and without this feature:

- ***lifetime_minutes***: Number of minutes between pull request opening and closing (if closed)
- ***mergetime_minutes***: Number of minutes between pull request opening and merging (if merged)
- ***num_commits***: Total number of commits for a pull request
- ***src_churn:*** Total number of lines changed
- ***files_changed:*** Total number of files touched by pull request
- ***num_commit_comments:*** Number of review comments
- ***num_issue_comments:*** Number of discussion comments
- ***num_participants:*** Number of discussion participants

These characteristics were selected to describe the impact of *suggested changes* on development processes and better understand how this feature influences contributor and reviewer behavior on pull requests. To determine the impact of GitHub *suggested changes*, we analyzed ***lifetime_minutes*** and ***mergetime_minutes*** to determine their impact on development time, ***num_commits***, ***src_churn***, and ***files_changed*** to analyze their affect on developer productivity, and ***num_commit_comments***, ***num_issue_comments***, and ***num_participants*** to observe their impact on collaboration between developers. We amassed these metrics and compare these characteristics for pull requests with GitHub *suggested changes* and those without this feature.

[8] `https://about.gitlab.com/`
[9] `https://bitbucket.org/`

Table 6.4 Developer Impact Study Data

Project (*Primary Language*)	**SCs**	**RCs (w/ code)**	**PRs**
qmk/qmk_firmware (*C*)	8525	9127 (675)	3600
nodejs/node (*JavaScript*)	2252	14364 (818)	5889
rust-lang/rust (*Rust*)	2615	24139 (1068)	8475
go-gitea/gitea (*Go*)	1317	6604 (338)	2822
rapid7/metasploit-framework (*Ruby*)	754	4057 (517)	1252
Qiskit/qiskit-terra (*Python*)	626	3505 (204)	1735
kubernetes/kuberbetes (*Go*)	556	52297 (2645)	12106
qgis/QGIS (*C++*)	481	2893 (37)	2548
neovim/neovim (*Vim script*)	208	3087 (163)	1746
python-pillow/Pillow (*Python*)	156	354 (19)	650
mono/mono (*C#*)	134	6043 (182)	5821
lydiahallie/javascript-questions (*Markdown*)	49	43 (2)	231
jpmorganchase/quorum (*Go*)	10	192 (13)	193
firebase/quickstart-android (*Java*)	5	89 (6)	155
mavlink/qgroundcontrol (*C++*)	5	261 (14)	863
qbittorrent/qBittorrent (*C++*)	5	3966 (205)	486
ironhack-labs/lab-advance-querying-mongo (*Markdown*)	4	159 (12)	753
kenwoodjw/python_interview_question (*Markdown*)	3	2 (0)	38
lballabio/QuantLib (*C++*)	3	57 (4)	146
Azure/azure-quickstart-templates (*PowerShell*)	2	2718 (2)	1630
h5bp/Front-end-Developer-Interview-Questions (*HTML*)	1	81 (2)	56
qunitjs/qunit (*JavaScript*)	1	77 (11)	55
Total:	17,712	134,318 (6,937)	51,250

SCs: GitHub *Suggested Changes* **RCs:** Pull Request Review Comments **PRs:** Pull Requests

6.3.3 Phase 2: *Developer Feedback on Suggested Changes*

The second phase of this study explores the usefulness of GitHub *suggested changes* by collecting feedback from developers.

6.3.3.1 Data Collection

Participants

To collect feedback from developers, we recruited users who interacted with GitHub *suggested changes*. Surveys were emailed to users with publicly available email addresses who either received *suggested changes* on their pull request or made a suggestion on a developer's pull request within six months of the time of this study. Surveys were sent to a total of 580 GitHub users who interacted with the GitHub *suggested changes* feature, and we received responses from 43 developers (7.4% response rate). Throughout the remainder of this chapter, the C- prefix is used describe *suggestees*, or contributors who received a *suggested change* on their pull request, and the R- prefix indicates a *suggester*, or a reviewer who made a comment with *suggested changes* on a pull request.

6.3.3.2 Determining the usefulness of suggested changes

To explore the usability of GitHub *suggested changes*, we surveyed developers to gain insight on their experiences and usages of this feature. The survey first asked participants to provide a 5-point Likert scale ranking on how useful they found the *suggested changes* feature. We also incorporated free response questions for participants to provide additional open-ended feedback and specific details about what they find useful or unuseful about this system as well as how they integrate this feature into their development workflow. To evaluate the usefulness of *suggested changes*, we aggregated the Likert scores and present the collective responses from developers. To further analyze feedback from participants, two independent researchers performed an *open* coding on the open-ended responses from users on various aspects of GitHub *suggested changes*. The researchers reviewed text responses, came up with themes based on participant responses, met to discuss their derived categories, and came to an agreement on major themes provided in feedback from developer comments (inter-rater agreement = 72%, Cohen's $\kappa = 0.6828$).

6.3.4 Results

This study presents an empirical analysis of the GitHub *suggested changes* feature exploring types of suggestions, the effectiveness of code recommendation systems, its impact on pull-based software development, and feedback from developers.

6.3.4.1 RQ1: Types

To answer RQ1, we used an open coding technique to categorize types of recommendations with GitHub *suggested changes*. Our qualitative analysis derived four categories to describe types of suggestions

made by developers. The identified categories are presented with examples below and we provide further analysis exploring 100 randomly sampled suggestions on the frequency of each finding:

6.3.4.1.1 Categories

Corrective: The corrective category refers to developers fixing problems found in code. Prior work suggests fixing defects is the primary motivation for code reviews at Microsoft [Bac13] and *algorithm errors*, or problems that arise in the correctness or implementation of a program, are the most common type of issue discovered during code inspections [Alo00]. Figure 6.4a presents an instance of a corrective suggestion made by a user on a pull request. The developer referenced a variable as a global variable in Python instead of as a class variable, leading the reviewer to propose a fix by adding the `self` keyword.[10]

Formatting: The formatting category represents code changes that impact presentation and style without changing the functionality of code. For example, research shows refactoring, or the process of restructuring code without changing its behavior, is beneficial for improving code quality [Str07]. Studies also show code reviews are useful for ensuring source code is formatted correctly and style guidelines are consistent for development teams [Bac13] and repairing the *visual representation* of code [Män08]. A sample formatting suggestion is presented in Figure 6.4b,[11] where the reviewer recommends modifying the whitespace within the code to adhere to spacing requirements delineated by the Python Enhancement Proposal (PEP 8) coding style guide.[12]

Improvement: This category describes when reviewers recommend changes to refactor or optimize a user's code. Developers at Microsoft reported improvements to code quality are the most important benefit of code reviews.[13] Furthermore, prior work asserts correct algorithms may need additional improvements during code reviews, such as to the efficiency of the implementation [Alo00] and problems in the structure and organization of the code that can be optimized [Män08]. Figure 6.4c presents an example of an improvement suggestion, where the reviewer recommends enhancing the readability of the code by renaming a variable from `x` to a more descriptive name `manifest`.[14]

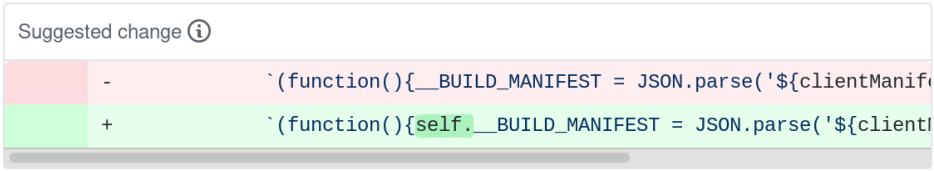
Non-Functional: The non-functional suggestions occur when reviewers recommend changes that don't impact code. For instance, suggestions to fix spelling and grammatical errors in documentation and code comments. Research shows that documentation issues are prevalent in software, and are the most frequent type of fix applied for code reviews in open source software [Bel14]. Similarly, problems with the understandability and maintenance of software, including documentation errors, are the most common type of defects found during code reviews [Män08]. An example of a non-functional recommendation is depicted in Figure 6.4d. In this instance the reviewer discovers a typo within a code comment misspelling *deserialize* as "deseriale", and uses *suggested changes* to fix it.[15]

[10] `https://github.com/zeit/next.js/pull/7696#discussion_r302333269`
[11] `https://github.com/numba/numba/pull/4204#discussion_r310598073`
[12] `https://www.python.org/dev/peps/pep-0008/#whitespace-in-expressions-and-statements`
[13] `https://www.michaelagreiler.com/code-reviews-at-microsoft`
[14] `https://github.com/gatsbyjs/gatsby/pull/13471#discussion_r277948539`
[15] `https://github.com/microsoft/terminal/pull/1258#discussion_r293932790`

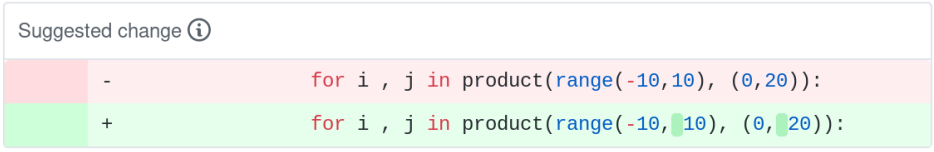
6.3.4.1.2 Usage

To analyze these categories, two independent researchers classified 100 randomly sampled instances of the GitHub *suggested changes* feature. Table 6.5 presents the findings for each category, and we found that *non-functional* recommendations are the most popular type of change submitted with this system. While non-functional changes were the most frequent type of suggestion, we also found *suggested changes* are useful for proposing code improvements to enhance and optimize developers' code. Overall, we determine GitHub *suggested changes* have the ability to accommodate a variety of recommendations between developers to improve the quality of code during pull requests reviews.

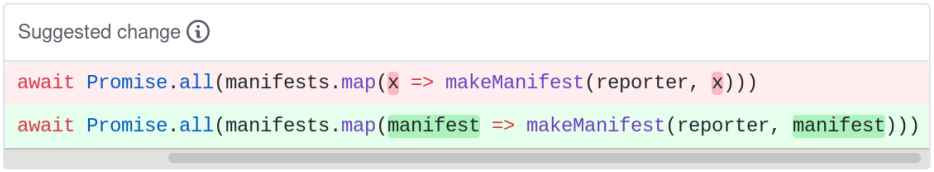
(a) Corrective:



(b) Formatting:



(c) Improvement:



(d) Non-Functional:

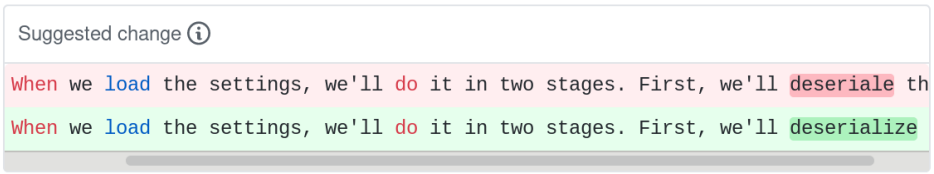


Figure 6.4 Categories of *suggested changes* Results

Table 6.5 GitHub *Suggested Changes* Categories Results

	n	**Percentage**
Non-Functional	36	36%
Improvement	34	34%
Corrective	16	16%
Formatting	14	14%

6.3.4.2 RQ2: Effectiveness

To evaluate the RQ2, we investigated the effectiveness of code recommendation systems on pull requests, specifically GitHub *suggested changes* and pull request review comments with fenced code, on the acceptance and timing of pull requests and recommendations. We found that *suggested changes* and review comments with code snippets made up approximately 12% and 5% of all pull request review comments in our dataset, respectively. The Pearson's chi-squared (χ^2) test was used to analyze acceptance while the Mann-Whitney-Wilcoxon test was used to evaluate time.

Acceptance

Contribution Acceptance. Overall, 69% of pull requests in our dataset were merged into repositories ($n = 35{,}521$). Table 6.6 presents the merge rate for pull requests with and without code suggestions from either system. Our results show both groups have approximately the same merge rate, and we found no significant difference in the outcome of contributions based on the existence code recommendations with *suggested changes* or review comments ($\chi^2 = 0.0182$, $p = 0.8928$, $\alpha = .05$). Thus, while suggestions are useful for helping developers resolve problems in code reviews, they do not have a major influence on whether pull requests are accepted and merged into repositories.

Recommendation Acceptance. We were also interested in the acceptance rate of recommendations from each system. Overall we found GitHub *suggested changes* were much more effective (Table 6.7). 60% ($n = 10{,}556$) of *suggested changes* were incorporated into pull requests by developers compared to only 1% ($n = 65$) of review comments with markdown code. Additionally, we found this difference was statistically significant ($\chi^2 = 6961.3765$, $p < 0.00001$, $\alpha = .05$). This indicates the *suggested changes* feature is more effective for encouraging developers to adopt recommendations from peers moreso than markdown code presented in pull request review comments.

While general code suggestions do not impact the acceptance of contributions on GitHub, we conclude *suggested changes* are effective for proposing improvements to contributors on their pull requests compared to other systems, such as review comments with code.

Timing

Contribution Time. To further evaluate the impact of code suggestions, we measured the amount of time to accept code contributions with *suggested changes* or review comments with fenced code compared to those with neither system. The results, presented in Table 6.8, show that on average pull requests with code suggestions take over twice as long to merge into repositories. Additionally, we found that pull requests without code suggestions from these systems are merged into repositories significantly faster than contributions with recommendations ($W = 87857043$, $p < 0.00001$, $\alpha = .05$). We deduce contributions with recommendations to developers via *suggested changes* or review comments with code may take longer to be accepted due to increased comments from reviewers, discussions about changes, and time needed to make modifications compared to non-suggestion pull requests.

Recommendation Time. To compare GitHub *suggested changes* and review comments with fenced code as mechanisms for recommendations, we analyzed the amount of time for reviewers to comment with suggestions using each system. Our results, displayed in Table 6.9, show *suggested changes* recommendations are made about four days faster than review comments. Furthermore, this difference in recommendation time is statistically significant between the code suggestion systems ($W = 49186174$, $p < 0.00001$, $\alpha = .05$). This indicates reviewers are able to make recommendations much faster using the *suggested changes* feature compared to code within review comments, allowing developers to provide feedback more efficiently and accelerate the review process.

Recommendation Acceptance Time. Additionally, we analyzed the amount of time for developers to accept recommendations from reviewers using each system. Table 6.10 presents the average acceptance time for developers for GitHub *suggested changes* and review comments with fenced code, and we found *suggested changes* are accepted on average three days faster. Our results also show a significant difference in the amount of time taken by developers to incorporate recommendations from each system into their pull requests ($W = 256013$, $p = 0.0001$, $\alpha = .05$). This shows GitHub *suggested changes* allow developers to evaluate suggestions and make decisions on recommendations quicker than with markdown code in pull request review comments, for example including the "Commit suggestion" button to automatically apply changes.

Overall, our timing results show that, while contributions take longer overall to get merged when code is recommended, *suggested changes* facilitate development processes by decreasing the amount of time needed to make suggestions and approve recommendations during code reviews.

Table 6.6 Contribution Acceptance Results

Pull Requests	*n*	Rate
with suggestions	6982	69.4%
without suggestions	44268	69.3%

Table 6.7 Recommendation Acceptance Results

Type	*n*	Rate
suggested changes	17712	59.6%
review comments with code	6937	0.9%

Table 6.8 Contribution Time (in days) Results

Type	*n*	Average	Median
with suggestions	6982	16.4	5.0
without suggestions	44268	6.4	1.1

Table 6.9 Recommendation Time (in days) Results

Type	*n*	Average	Median
suggested changes	17712	10.5	0.7
review comments with code	6937	14.6	1.9

Table 6.10 Recommendation Acceptance Time (in days) Results

Type	*n*	Average	Median
suggested changes	17712	5.4	0.3
review comments with code	6937	8.0	0.7

6.3.4.3 RQ3: Impact

To evaluate RQ3, we analyzed the impact of the *suggested changes* feature on pull-based software development by examining its effect on GitHub pull requests. Overall, we analyzed the timing, coding activity, and collaboration between developers. The Mann-Whitney-Wilcoxon test ($\alpha = .05$) was used to compare characteristics on 4,319 pull requests containing GitHub *suggested changes* and 46,931 without recommendations between developers through this system.

Pull Request Characteristics

To understand the impact of *suggested changes* on development practices, we analyzed metrics used in prior work to examine pull requests on GitHub [Gou14a]. These results are presented in Table 6.11. Here, we explain specific results based on the pull request characteristics we observed.

Timing. Similar to our analysis evaluating the impact of code suggestions on contribution time, we found that pull requests with GitHub *suggested changes* take almost twice as long and significantly longer to be accepted (***mergetime_minutes***, $p < 0.00001$) and closed, with or without merging (***lifetime_minutes***, $p < 0.00001$). This may be due to the fact that pull requests with *suggested changes* extend reviews and add more time to the code inspection process, have more complex development activity, and more conversations between developers on contributions.

Coding Activity. One reason pull requests utilizing *suggested changes* take longer is because developers are significantly more active on these contributions. We found pull requests with this feature have more commits (***num_commits***, $p < 0.00001$) and modified lines of code (***src_churn***, $p < 0.00001$) from contributing developers. However, we found on that average pull requests with this feature impact fewer files. For RQ2, we found *suggested changes* are well-accepted by developers. With this system, developers can automatically apply recommendations from reviewers as additional commits to their pull requests on a line of code within a file, leading to increased commits and code changes to improve the quality of the program.

Collaboration. Additionally, our results show that GitHub *suggested changes* impact feedback and discussions during code review processes. We found that pull requests with developers using the *suggested changes* feature have significantly more review comments on lines within the submitted code (***num_commit_comments***, $p < 0.00001$), general discussion between programmers about contributions (***num_issue_comments***, $p < 0.00001$), and developers engaging in conversations about pull requests (***num_participants***, $p < 0.00001$). This indicates *suggested changes* are a valuable system for providing feedback, facilitating communication between developers, and discussing improvements to pull requests.

6.3.4.4 RQ4: Usefulness

To study RQ4, we surveyed developers to collect feedback on the GitHub *suggested changes* feature. Of the 43 survey responses we received, 24 responses from were from developers who received suggestions and 19 responses were from users who made a recommendation. Here we present the results from analyzing 5-point Likert scale responses and open-ended replies from developers.

Likert Scale. Figure 6.5 displays the results of the Likert scale asking developers how useful they found the *suggested changes* feature. Overall, we found 92% of suggestees ($n = 22$) and 79% of suggesters ($n = 15$) reported GitHub *suggested changes* are Useful or Very Useful. Furthermore, no developer who

Table 6.11 Pull Request Impact Results

Characteristic	Dataset	Mean	Median	*p-value*
lifetime_minutes*	with suggested changes	34799.26	8190.43	-
	without suggested changes	18993.48	2489.12	$p <$ **0.00001**
mergetime_minutes*	with suggested changes	23645.15	5900.27	-
	without suggested changes	10378.82	1776.32	$p <$ **0.00001**
num_commits*	with suggested changes	8.62	4	-
	without suggested changes	6.72	1	$p <$ **0.00001**
src_churn*	with suggested changes	866.86	133	-
	without suggested changes	3212.64	26	$p <$ **0.00001**
files_changed	with suggested changes	7.71	2	-
	without suggested changes	11.54	2	$p = 0.5051$
num_commit_comments*	with suggested changes	13.56	7	-
	without suggested changes	2.00	0	$p <$ **0.00001**
num_issue_comments*	with suggested changes	8.63	5	-
	without suggested changes	5.03	3	$p <$ **0.00001**
num_participants*	with suggested changes	3.18	3	-
	without suggested changes	2.30	2	$p <$ **0.00001**

*** denotes statistically significant results (**p-value < 0.05**)

interacted with this system reported that it was Not at All Useful. One participant (R12) ranked the system as Somewhat Useful due to the fact that *suggested changes* incorporate accepted suggestions as individual pull request commits and desired a "force push" option to squash all of the commits and better adhere to their team's development workflow. Overall, our results show GitHub users find this feature effective for both receiving recommendations as well as making suggestions to developers on GitHub.

Qualitative Feedback. To further examine GitHub *suggested changes*, we asked surveyed developers to provide open-ended responses describing what they find useful or unuseful about suggested changes. Two researchers conducted an open coding on free response feedback to derive themes describing how developers perceive this feature. The main negative feedback received about *suggested changes* focused on the limitations of the system (i.e. "*no multiline support*" (R1)), its conflicts with other development tools (i.e. "*approval process handlers like PullApprove and Zappr now will not recognize my approval of the PR*" (R16)), and the functionality of applying suggestions as separate commits (i.e. "*Some changes could be grouped in a single commit...[individual commits] is less convenient*" (C9)).[16] We extracted eight themes from our analysis to describe what developers found useful about GitHub *suggested changes*. Below, we define each category derived from the qualitative coding and provide example responses to summarize developer feedback on the advantages and disadvantages of *suggested changes*:

[16]At the time of this experiment, GitHub *suggested changes* only supported single line changes. However, after the study date GitHub has since updated the feature and introduced *suggested changes* for multiple lines (`https://github.blog/changelog/2020-02-26-multi-line-code-suggestions-beta/`).

Actionability: Developers reported finding *suggested changes* useful because of the actionability of recommendations from this system. This refers to ability for users to automatically act on recommendations from reviewers and commit proposed code changes into their on pull requests. Survey respondents frequently lauded the fact that *suggested changes* are are actionable and easy to integrate into code review processes.

> *"the fact that small changes can be applied immediately, and the fact that they can be described by the reviewer in a way that a button fixes it instead of going to your code"* (C3)

Communication: Participants also reported that GitHub *suggested changes* are effective for communication between developers. This refers to the capability of developers to understand recommendations and the transfer knowledge between peers through this system. Our survey feedback suggests *suggested changes* provide a mechanism for clear communication between developers and reviewers during the code review process.

> *"We can understand reviewer's intention more. If not using this feature, there are only pull request comment text, so we may misunderstand reviewer's intention"* (C16)

Code: Users also found the GitHub *suggested changes* feature useful is because it allows reviewers to make recommendations as code. Several participants mentioned they preferred the ability to give and receive suggestions using code rather than writing out recommendations as text in pull request review comments.

> *"It is very convenient that the reviewer can write what they suggest to change in code instead of formulating it in words (which will often be longer)"* (R6)

Conciseness: Similarly, participants also appreciated the brevity of recommendations with *suggested changes*. Many developers found the concise code recommendations to be useful for easily making and reacting to suggestions when using this system. Examples of this can found in responses such as the one below where a suggester found suggested changes useful because they can:

> *"Suggest small one line changes directly and concisely"* (R9)

Ease of Use: Respondents also noted that recommendations with *suggested changes* are effective because the system is easy for suggesters and suggestees to use. Feedback from participants commended this feature for its intuitive interface design and ability to effortlessly integrate into the pull request review process.

> *"It's really easy to use, it's really easy to accept the suggestion so some works are really easy"* (C15)

Location: Participants also found the location, or where *suggested changes* recommendations are made, useful for making and receiving suggestions. We found developers liked the ability to make recommendations directly on the line of code to be improved. For example, one respondent mentioned this feature is useful because there is:

"No need to leave the pull request page to make a suggested change, [and it's] easy to see what is being suggested" (R12)

Scalability: Another reported benefit of *suggested changes* is the ability of this feature to make and receive numerous recommendations on the same pull request. Users found the scalability of this system useful, especially for reviewing code and making several recommendations to developers on the same pull request during reviews.

"It...gives me the ability to suggest multiple options during review" (R11)

Timing: Developers also found the timing of *suggested changes* makes them useful. For example, many participants mentioned this feature improves the speed with which users can make and address recommendations on pull requests and also noted its overall impact on accelerating the code review process.

"Being allowed to add specific changes speeds up the review process. Sometimes it is easier to make the changes yourself rather than make a suggestion and wait for a change" (C17)

No Response: Several participants who completed the survey did not respond to the free response question to provide feedback on the usefulness of GitHub *suggested changes*. This question was optional in the survey, and we cannot conclude the reason developers chose not to respond is due to not finding anything useful about the feature.

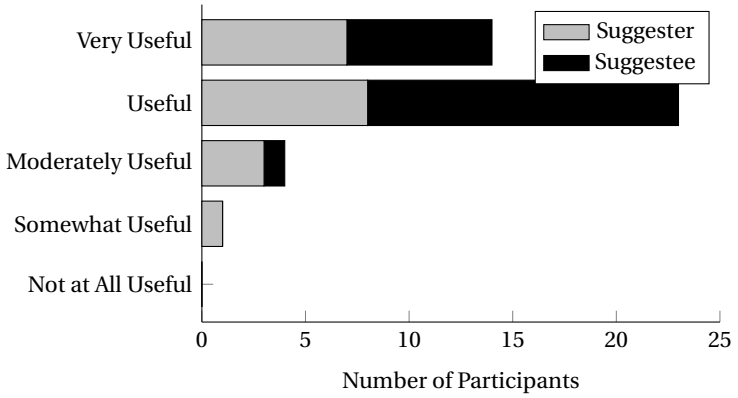


Figure 6.5 Survey Results on the Usefulness of GitHub *suggested changes*

6.3.5 Summary

Our findings show that GitHub *suggested changes* have a major influence on development practices and behavior. We discovered this feature is useful for suggesting different types of code changes, effective for

facilitating recommendations during reviews, and beneficial for improving coding activity and collaboration between developers. Additionally, we found developers find *suggested changes* useful for making and receiving recommendations on GitHub pull requests. Based on qualitative analysis examining user feedback on the advantages of *suggested changes*, we submit two main implications for improving the future recommender systems and their developer impact. These implications include incorporating ***user-driven communication*** (*Communication*, *Code*, and *Conciseness*) and ***workflow integration*** (*Actionability*, *Ease of Use*, *Location*, *Scalability*, and *Timing*) into automated recommendations to improve developer behavior.

6.4 Discussion

To evaluate *developer recommendation choice architectures*, this work analyzes GitHub *suggested changes* as a recommendation system incorporating all the principles from this conceptual framework. The findings from the evaluations of *suggested changes* presented in this chapter show that developers prefer to receive recommendations from this system and it has a major impact on development practices on GitHub. Overall, we provide four implications for designing effective recommender systems based on the results of these studies: ***recommendation content***, ***recommendation design***, ***user-driven communication***, and ***workflow integration***. Here, we explain these conclusions from the context of each study and through the lens of the *developer recommendation choice architectures* framework.

6.4.1 Actionability

Our analysis of the GitHub *suggested changes* feature suggests that actionability influences the perception and adoption of recommendations by developers. For example, the recommendation styles and developer impact study offer *recommendation design* and *workflow integration* as implications to improve automated suggestions from future recommender bots. To describe the impact of actionability in *suggested changes*, I describe themes derived from qualitative feedback from developers in the studies presented in this chapter.

Workflow Details

The recommendation styles results show that *integration* details within the content of automated suggestions can influence adoption. These workflow details, describing how to install and integrate tools into development processes, also contribute to the actionability of recommendations by providing information on how easy it is to adopt tools and practices. For example, participants mentioned they were interested in learning if suggested tools would easily work with "*whatever CI or test runner system I've got*" (P8) and wanted to know if they had "*simple integration*" (P1). Similarly, developers desired information on the *reliability* of systems to learn about their stability and how well tools do the things they claim to do.

Furthermore, participants mentioned designing automated recommendations to include relevant *examples* of usage. For instance, P14 stated "*specific examples...would be a lot more compelling*" and

P10 noted "*It would be better if they can show me some examples with some very clear results like this is something you can get with our tool*". Additionally, developers were interested in testing out the actionability of recommendations saying "*I would like to try it by myself first*" (P11) and "*I would test it out locally*" (P8). By incorporating details on the actionability of recommendations this concept into developer recommendations such as how to install tools, integrate them into CI build systems, and the existence of plugins within popular IDEs, automated recommendations can increase the likelihood of adoption for developer behaviors.

6.4.1.1 Workflow Integration

Both GitHub *suggested changes* studies show the importance of integrating recommendations effectively into developer workflows. For instance, developers were likely to adopt tool recommendations from *suggested changes* over other recommender systems because of its ability to automatically apply recommendations and its "neat integration" (P7). Our preliminary work also shows that integration is a key factor in developer adoption, with the disruptiveness of notifications from `tool-recommender-bot` discouraging users from adopting recommendations. However, we found developers were more likely to adopt automated static analysis tool recommendations from actionable systems such as GitHub *suggested changes*.

The developer impact study also found actionability increased adoption of code change suggestions. We found *suggested changes* were more popular and more effective than static pull request review comments with fenced code. Our qualitative results also showed this was one of the most useful aspects of this recommendation system. For example, participants mentioned they found "*the ability for people to be able to suggest changes and to be able to incorporate those changes immediately*" (C8) and the fact suggestions can be "*accepted right away, without requiring copy pasting and committing on my side*" (C9) useful for receiving recommendations. Additionally, developers reported *ease of use* and *scalability* contributed to the ability to easily adopt *suggested changes* into code review processes and make suggestions to developers. Hence, we show actionability can enhance automated recommendations and improve the effectiveness of recommender systems.

6.4.2 Feedback

The *suggested changes* studies also show that the information provided to users in automated recommendations impacts adoption. For example, we found *recommendation content* influences the style of automated recommendations and *user-driven communication* is a favorable aspect of GitHub *suggested changes*. To illustrate the importance of feedback in recommendations with this system, I outline themes provided by developers from the results of the recommendation styles and developer impact studies.

6.4.2.1 Relevance

Developers in the recommendation styles study reported details concerning the relevance of tools impacts their decisions on adoption. For instance, P2 mentioned the content of recommendations helps them decide whether "*it's something I really need or want*", P9 stated "*the recommendation itself matters less than how much I need the thing...If I don't need it then I'm not going to try it*", and P12 declared they "*don't really care*" about tools that are irrelevant to their work. Participants were specifically interested in receiving information about integration details, the reputation of tools, and the trustworthiness of systems. Alternatively, irrelevant recommendations that "*show some other useless stuff which may confuse the potential user*" (P10) can lead to ineffective recommendations. For example, we found naive *telemarketer design* suggestions were not useful because they provided irrelevant and generic feedback in recommendations to users.

Similarly, developers expressed disdain for recommendation content that appears to be *marketing* or advertisements. For example, developers described the email recommendation stating it sounds "*pretty suspicious*", "*email...there's so much stuff that comes through email*" (P1), "*email is definitely a no*" (P7), and "*I'd immediately delete it*" (P6). Additionally, P5 mentioned "*I'm not somebody who likes to get unsolicited marketing stuff*". The negative perception of marketing in software engineering can also be found from the backlash received by maintainers of the StandardJS JavaScript style guide, linter, and formatting tool,[17] who attempted to raise funds for development by incorporating advertisements inside the terminal.[18] In this instance, we found poor feedback in suggestions discouraged users from adopting recommendations and contend automated systems should avoid marketing language in recommendations.

However, we found details about the *popularity* of systems plays a key role in adoption according to participants in the recommendation styles study. Most developers desired information the reputation of tools when making decisions, stating "*one of the things I want to see is what other people think about it*" (P3), "*how many people use it...the popularity of the tool being used would influence me to try that*" (P6), "*when there's a buzz around a tool, that's when you know it's good and you know it's worth checking out*" (P12), and recommendations should "*try to highlight the popularity, popularity is so crucial*" (P13).

Participants also mentioned using various sources for learning about the reputation of tools, such as from peers (i.e. "*word of mouth and people that I actually trust who use it*" (P6)) and online searches (i.e. "*Google the tool's name...[and get] a link on the first page*" (P5)) in addition to social media and other online programming communities. Prior work also shows displaying details about popularity impact adoption of developer behaviors, such as increasing updates to repositories and contributions from collaborators [Agg14]. This indicates automated recommendations should consider incorporating popularity statistics such as number of users, downloads, social media followers, and reviews to encourage adoption. Thus, we argue including relevant feedback about tools and practices is necessary to encourage the adoption of developer behaviors

[17] `https://standardjs.com/`

[18] `https://github.com/standard/standard/issues/1381`

6.4.2.2 Conciseness

The recommendation styles and developer impact study found users appreciated the brief and compact nature of recommendations with GitHub *suggested changes*. For example, when describing the email and issue static analysis tool recommendations mentioned they contained "*a ton of words*" and were "*way too lengthy*" (P13). Additionally, P10 noted effective recommendations "*just get[sic] to the point*". However, P8 liked *suggested changes* because they provide a "*nice, concrete error*" (P8). Even though all of the recommendation styles contained similar text in the suggestion, systems such as email appeared to contain more text for developers to read. Likewise, research suggests software engineers prefer instant messaging systems because they are more concise and quicker than emails [Bla13].

Concise recommendations with GitHub *suggested changes* also played a role in impacting developer behavior. For example, R11 noted "*it uses less words*". Developers also reporting preferring recommendations as *code* stating this system "*can quickly and precisely show what change they expect. Describing the change with words is pretty annoying*" (C12), "*removes guesswork from interpreting a prose explanation*" (R4), and "*removes all ambiguity about what I'm asking for if I can just directly put the code there*" (R12). Prior work also found conciseness is important designing interactive software systems [Was81] and in *quick fix designs* to increase adoption of code-checking systems [Joh13].

6.4.2.3 Communication

The developer impact study results show one of the most useful aspects of *suggested changes* is their ability to effectively communicate to developers. Participants found this system effective for facilitating clear communication between developers on pull requests. For instance, users replied this system is useful because it "*lets someone else directly make changes instead of writing out instructions on how to make changes*" (C10), "*gives the suggestion in a very clear way*" (R5), and provides "*easy information on what to change in your pull request*" (C5). This *user-driven communication* led to effective recommendations from developers on pull requests. Prior work also suggests poor communication from automated systems prevents static analysis tool adoption [Joh13] and frustrates developers during interactions with bots [Wes18], while *user-driven conversations* improves the effectiveness of recommender chatbots [Cer19] while specific and clear language can overcome barriers to security adoption adoption [Xia14]. We propose implementing clear communication focused on users to improve automated developer behavior recommendations. Overall, we conclude developers are more likely to adopt recommendations with relevant, concise, and clear feedback.

6.4.3 Locality

GitHub *suggested changes* provide support recommendation locality because of their ability to situate suggestions on lines of code and present recommendations during code reviews. We found this *recommendation design* for the placement and timing of notifications increased the likelihood of developers adopting behaviors. These related findings from our studies are presented below.

6.4.3.1 Timing

For the recommendation styles user study, we found the systems with more convenient temporal locality (GitHub *suggested changes*) were more likely to convince developers to adopt static analysis tools than those with low temporal locality (pull requests, issues, and emails). We refer to these systems as having low temporal locality because, unlike *suggested changes*, they can appear to developers in their email or repository any time during the development process. However, *suggested changes* are limited to open pull requests currently under review. Similarly, previous research in software engineering shows untimely recommendations prevented programmers from adopting code navigation recommendations from Spyglass [Vir09] while timely notifications increased the amount of Infer[19] and Zoncolan[20] static analysis warnings by developers at Facebook [Dis19].

In the developer impact study, many survey respondents commented on how GitHub *suggested changes* impacted the timing of reviews. For example, developers mentioned this feature "*lets me do reviews much faster*" (R3), "*accelerates getting pull requests accepted*" (C4), "*it's great to be able to quickly apply changes*" (C23), and "*it's often quicker both to suggest a minor change*" (R8). Our quantitative analysis also found that, while *suggested changes* lengthen the overall pull request review process, they are significantly faster for making recommendations and responding to suggestions during the code review process. Therefore, to design effective automated recommendations, systems should focus on making suggestions to developers at convenient times within the development workflow.

6.4.3.2 Location

Our recommendation style study results also suggest that the placement of notifications is important for developer recommendations. For example, we found participants were least likely to adopt recommendations from emails, the system with the worst spatial locality in a separate window outside of the repository. As the spatial locality of recommendations improved, we found developers were more likely to adopt static analysis tool recommendations with GitHub *suggested changes*, the most conveniently located notification on the line of code, being the most effective. P5 also mentioned the location of recommendations influences their decision on whether or not to adopt, noting they did not want to "*go to [the tool's] website and have to click through a million different links*". Prior work shows tool recommendations within the coding panels of IDEs can increase efficiency and are preferred by developers [Smi17] in addition to minimizing *visual momentum* and helping programmers feel less disoriented in their development environment [DA06].

Participants in the developer impact study also praised the location of *suggested changes*. For example, C24 stated this feature is useful because there is "*no need to leave the pull request page to make a suggested change*". GitHub developers also added this feature is useful because it "*shows suggested code changes integrated with the actual source*" (C22) and "I can just directly put the code there" (R13). Thus, we argue automated recommendations should appear at convenient locations within the programming

[19] `https://fbinfer.com/`

[20] `https://engineering.fb.com/2019/08/15/security/zoncolan/`

environment in order to increase the likelihood of adoption and improve the behavior of developers.

Overall, the results of the GitHub *suggested changes* studies show that this system is preferred by developers for receiving recommendations and effective for improving development practices. This system, which incorporates the *developer recommendation choice architectures* framework, uses this framework to make actionable, relevant, clear, and convenient recommendations to developers. The feedback provided from developers through semi-structured interviews and surveys confirm the presence of these framework principles contribute to the effectiveness of *suggested changes* on GitHub. Ultimately, we conclude *developer recommendation choice architectures* is a useful mechanism for creating automated recommendations because of its focus on recommendation content and design as well as fostering user-driven communication and facilitating workflow integration.

CHAPTER

7

DESIGNING NEW RECOMMENDER BOTS

Based on findings from the previous chapter, automated recommendations containing *developer recommendation choice architectures* are preferred by software engineers and influence their development practices. The studies presented analyzed *developer recommendation choice architectures* through the GitHub *suggested changes* feature, and show that developers are more likely to adopt tool recommendations from systems incorporating this framework [Bro20] in addition to showing the suggestions made through this feature are effective for making and receiving code improvement recommendations and improving the coding activity and collaboration of peers on pull requests [Bro20]. However, the thesis of this dissertation argues that *developer recommendation choice architectures* is able to improve code quality and developer productivity. To investigate this claim, I developed a new system, `class-bot`, that incorporates each of the design principles from this conceptual framework and present an evaluation of this system exploring its impact on the quality and productivity of programmers' work. Study materials for this work are available in Appendix D.

7.1 Study Rationale

Undergraduate Computer Science courses are constantly evolving to handle the significant increase of students [Kay98]. For example, educators have turned to many different automated tools to complete instructional tasks such as grading assignments and generating feedback on student code [Wil15]. However, studies show that despite increasing enrollment and the advantages of automated systems in the classroom, the dropout out rate in Computer Science, especially among first and second year students, is also growing. Beaubouef and colleagues suggest a primary reason for high attrition in these courses is poor behavior, such as ignoring software development processes, on programming

assignments [Bea05]. To further explore the impact of *developer recommendation choice architectures* on improving developer behavior, this work seeks to apply this framework to make recommendations to students on programming projects. Decision-making is vital in software engineering [Woo19], however students frequently make poor choices and adopt bad programming behaviors when writing code for projects [Edw09]. Furthermore, these behaviors persist among professional software engineers who also often underestimate the time and effort required to complete development tasks [Boe84], leading to further problems such as inadequately tested software [Whi00] and insufficient documentation [Bri03].

7.1.1 Research Questions

To examine the impact of digital nudges for improving the behavior of students on programming assignments, we seek to answer the following research questions:

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

To answer these questions, I performed a study implementing `class-bot`, a system that utilizes *developer recommendation choice architectures* to recommend beneficial software engineering behaviors to students, on projects for an introductory undergraduate programming course. The effectiveness of this system was evaluated by examining the code quality of projects and the productivity of students. Our results suggest automated nudges from *developer recommendation choice architectures* improved performance, increased coding activity, and prevented procrastination on assignments. The contributions of this research include `class-bot`, a novel bot for recommend software engineering practices to students on coding projects, and an evaluation of *developer recommendation choice architectures* this system on improving the software engineering behaviors of student programmers.

7.2 `class-bot`: Implementing *Developer Recommendation Choice Architectures*

To analyze automated approaches for improving programmer behavior, this work posits `class-bot`. The `class-bot` system nudges students to improve their behavior on programming assignments by automatically generating and updating issues on project GitHub repositories. This bot, presented in Figure 7.1, utilizes GitHub issues because they allow developers to manage bugs, suggest enhancements, and provide feedback on repositories [Gita]. Furthermore, prior work suggests the GitHub issue tracker is useful for making recommendations to developers [Bis13].

To improve student programming behaviors, `class-bot` encouraged them to follow the *software development process*, or set of activities necessary to develop and maintain software applications. This procedure includes activities related to the *Requirements*, *Design*, *Implementation*, *Testing*, and *Deployment* of software. Prior work suggests students failing to follow the software development process leads to high attrition in undergraduate Computer Science courses [Bea05]. The generated issues from `class-bot` on student repositories contained sections for each software development process phase.

For example, the Rq icon in Figure 7.1 represents the Requirements phase and activities related to understanding project guidelines, such as adding a description in the README. In each section, this system outlined rubric items for the assignment relevant to each phase.

Automated issues from `class-bot` fit the definition of a nudge because they do not provide incentives to students for completing tasks nor prevent students from avoiding items. This system was evaluated against a baseline approach of using an online rubric to illustrate project requirements and the software phases.[1] To improve the programming behaviors of undergraduate Computer Science students, `class-bot` was designed using *developer recommendation choice architectures*: *Actionability*, *Feedback*, and *Locality* [Bro20]. Below, we explain how `class-bot` incorporates each principle to improve student adherence to software engineering process phases.

Actionability

Actionability involves automating tasks to encourage the adoption of developer behaviors and reducing user effort. `class-bot` incorporated this principle by programmatically analyzing project repositories to determine if students complete certain development process tasks according to the rubric. If the tool observes an item was accomplished based on recent commits to the project, then `class-bot` automatically updated the issue to indicate the task was completed. For example, the system would automatically run the unit tests for the code to determine if a project's unit tests were passing the system or validate a .gitignore file was pushed to the repository to verify the configuration file was added. Alternatively, the baseline approach was not actionable as the online rubric required students to manually seek information and compare their development progress to the assignment requirements outlined on the website.

Feedback

Feedback consists of providing clear and coherent information to users in automated notifications. `class-bot` implements a simple feedback mechanism to present information to students on their adherence to software engineering processes. The system displayed a red x (❌) if the assignment requirements for a project task were not completed. When the bot detected a task was completed, it automatically updated the feedback icon next to the rubric item listed in the GitHub issue to be a green check mark (✅). For instance, in the Deployment phase of the `class-bot` example in Figure 7.1, the project repository does contain a .gitignore file but the source code does not compile. However, with the baseline approach students were not provided with any feedback on their project and were forced to determine if project expectations are met on their own.

[1] An example of the online rubric is available here: `https://pages.github.ncsu.edu/engr-csc116-staff/2020-summer/projects/project6/rubric`

Locality

Locality refers to the setting of automated recommendations, specifically when and where interventions are displayed to users during the development processes. To promote *spatial locality*, `class-bot` recommendations were implemented as GitHub issues located on the project repository within the issue tracker. To support *temporal locality*, the system automatically analyzed repositories daily to provide regular updates to the `class-bot` software development processes issues based on students' recent commits and code contributions to their project. However, the baseline approach does not incorporate temporal or spatial locality in that it forced students to search for information at a separate location from their repository on the course website and in ad hoc manner without any specified timing.

7.3 Methodology

To explore the impact of digital nudges on code quality and developer productivity, we implemented a mixed methods study to analyze *developer recommendation choice architectures* on student behavior for projects in a university-level introductory Java programming course.

7.3.1 Data Collection

7.3.1.1 Participants

Participants were undergraduate students enrolled in an introductory Java programming class. The students were from different majors, demographics, and levels of programming experience. For consistency in our data, students who enrolled and then eventually dropped the class were eliminated from this evaluation. Overall, we observed the behavior of 35 out of the initial 42 registered students. The participants were aware of the five phases used to define the software development process: *Requirements*, *Design*, *Implementation*, *Test*, and *Deployment*, as presented to students in the course curriculum.

7.3.1.2 Projects

To analyze *developer recommendation choice architectures* in `class-bot`, we observed and nudged developer behaviors to students on introductory Java programming projects. For the semester the course consisted of seven programming assignments, six projects and a final comprehensive exercise. Projects 3-5 made up the control group for this experiment to avoid the beginning assignments (Projects 1 and 2). `class-bot` was introduced to students on the final two assignments of the course, Project 6 and the Comprehensive Exercise, to examine its impact on the quality of projects and productivity of students. All coding projects for the were hosted and submitted on GitHub repositories. In total, we analyzed a total of 151 projects from students.

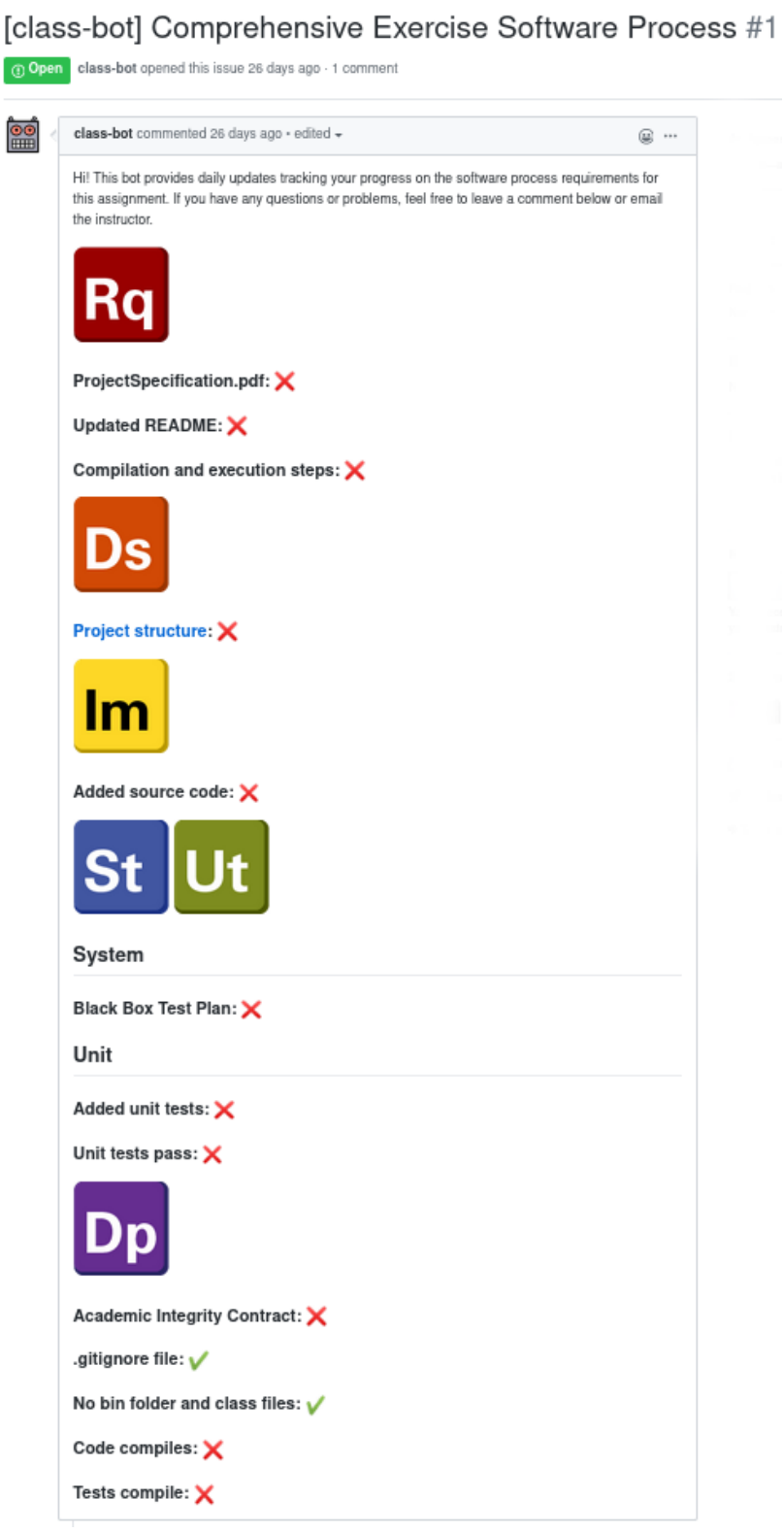


Figure 7.1 Example `class-bot` recommendation

7.3.1.3 Developer Behavior

To improve the decision-making of programmers, the developer behavior `class-bot` focused on encouraged students to follow the software engineering process. Prior work by Beaubouef and colleagues suggests students' failure to adhere to development processes factors into the high attrition rate and failure rate in early programming courses. For example, they note the following about typical student programming methods:

> "*This [students' estimated time to complete projects] minimally includes the processes of analysis, design, coding, testing, and documentation. Software projects developed by professionals are notoriously behind schedule in the real world. It should come as no surprise that software developed (programs written) by students will tend to be even more behind schedule.*
>
> *Unsuccessful students often want to skip analysis and design and begin typing in code immediately. Documentation is an afterthought at best, and little or no testing is performed. Because the student planned to attack the assignment in this manner from the beginning, he will often wait until the last minute to begin and work until its done or time runs out. These students set themselves up for frustration, unnecessary rework, and failure.*" [Bea05, p. 105]

We define the software development process as *Requirements*, *Design*, *Implementation*, *Test*, and *Deployment*. These phases were derived from the course materials and introduced to students during a lecture before `class-bot` was introduced on projects for this study. Our goal is to use *developer recommendation choice architectures* to encourage students to complete all of the software process phases for their projects. Each project contained a `class-bot` issue with sections for each phase listing the relevant tasks. Specific items listed differed based on the project requirements, however in general: Requirements (`Rq`) focused on activities to understand assignment specifications; Design (`Ds`) referred to the organization of code and project structure; Implementation (`Im`) centered on the development of the code; Testing concentrated on actions to develop unit tests (`Ut`) and functional test cases (`St`); and Deployment (`Dp`) concentrated on actions to verify the project and repository were ready for submission. When students completed a specific task, `class-bot` would automatically update the issue to indicate the item was completed.

7.3.2 Determining the effectiveness of `class-bot`

To examine the impact of `class-bot` on student behavior, we mined GitHub repositories to observe metrics measuring its impact on the quality of students' work and the productivity of their project development.

7.3.2.1 Quality

To answer RQ1, we evaluated the quality of student projects by examining their overall *grade* on the assignment in addition to the number of *points deducted* from the project due to students not adhering to requirements for the assignment.

Grade

For coding projects, the assignment grade indicates the overall quality of the project. Research suggests poor project management skills [Bea05] and ineffective behaviors [Edw09] result in low grades for students on programming assignments. Similarly, research shows the behavior and *software process maturity* impacts the quality of programs for professional software engineering teams [Cla97]. Within the course analyzed for this study, project grades were determined by real-world software engineering quality metrics, such as passing unit tests and functional test cases, Checkstyle[2] static analysis tool warnings, and correct project structure. To determine how well `class-bot` supports students in software process decisions, we analyzed projects with and without nudges from this system to determine how this approach impacted the overall grade of coding assignments as a method to determine the project quality.

Deductions

Another determinant of project quality is the number of points deducted on student programs. In education, grading penalties are commonly used to encourage students to perform better on assignments [Ree17]. Students who failed to meet certain requirements designated for the project had additional points subtracted from their overall grade. For example, submitting an assignment within 24 hours after the deadline resulted in a -10% late penalty. In this study, we analyzed projects with and without notifications from `class-bot` to determine if *developer recommendation choice architectures* improves student behavior and project quality by minimizing the number of points deducted from programs.

7.3.2.2 Productivity

To answer RQ2, this study examines if nudges impact the productivity of students working on programming assignments. We measured productivity by observing several different metrics mined from GitHub repositories, including the total *number of commits*, *code churn*, the amount of time until the *first commit*, and the timing between the *last commit* and the assignment deadline.

Commits

GitHub commits are used to record specific changes made to the project files.[3] Prior work in Computer Science education explores analyzing repository commits on version control systems to encourage students to make more frequent contributions to projects [Sin12] and predict student performance [Spr19]. In industry, commits have been used to measure contributions from programmers as well as the prolificacy of developers on GitHub [Vas16]. To discover if nudging students enhances productivity, we

[2] `https://checkstyle.sourceforge.io/`

[3] `https://docs.github.com/en/desktop/contributing-and-collaborating-using-github-desktop/committing-and-reviewing-changes-to-your-project#about-commits`

compared the total number of commits submitted by students to their project repositories with and without `class-bot` notifications.

Code Churn

To determine the impact of automated nudges from our system on the activity of students, we also analyzed the code churn of commits made to repositories on assignments with and without updates from `class-bot`. Prior work in software engineering suggests code churn is a useful metric measuring effort and the impact of code changes [Mun98], predicting the defect density in software [Nag05], and analyzing GitHub contributions in the pull-based software development model [Gou14a]. In this evaluation, code churn was measured by summing the number of lines added, deleted, or modified for each commit made by students to their project repository.

First Commit

To further investigate the impact of `class-bot` on the productivity of students, we analyzed the timing of commits. We examined the timing of the first commit on repositories to indicate when students began development on their project. Prior work by Edwards et al. shows that students who start working on assignments earlier receive significantly higher grades on projects, while students who start later often perform worse [Edw09]. To calculate the first commit time, we measured the amount of time between the creation and designation of project repositories to students after the assignment was announced in class until the first commit was made on the repository. We compare the timing of the first commit to projects with and without issues from `class-bot` to determine if the automated feedback provided from this system encourages students to start earlier and prevents procrastination on programming assignments.

Last Commit

We also analyzed the timing of the last commit on repositories to signify students completing work on their project. Edwards also found the time of last submission on a student programming project also significantly impacts their performance [Edw09]. To help students be more productive and discourage late submissions, we aim to user `class-bot` to nudge students to complete software development tasks. To further explore the impact of *developer recommendation choice architectures* on productivity, we calculated the amount of time between the last commit on a project repository and the project deadline to measure the last commit time.

Overall, we use these quality and productivity metrics to investigate the impact of bots incorporating *developer recommendation choice architectures* on the software engineering behaviors of students working on programming projects.

7.3.2.3 Survey

Additionally, at the end of the semester we sent students a survey to collect qualitative data on their experience with `class-bot` on their repositories. The survey contained a 5-point Likert scale question asking students about the usefulness of updates from this system and sought open-ended feedback to describe what they find useful about the automated updates against manually checking the website and to provide any additional comments about the system. We disseminated the survey to students after the last day of class and the completion of the projects and course assignments. Out of the 35 eligible students, we received 9 responses from participants (25.7% response rate). To analyze the survey data, Likert scale responses were aggregated to determine the usefulness of `class-bot` and we analyzed open-ended responses to derive feedback from students on improving automated bots for enhancing programmer behavior.

7.4 Results

To examine the impact of *developer recommendation choice architectures*, this work analyzed the impact of notifications from `class-bot` on the behavior of students working on programming projects. The Mann-Whitney-Wilcoxon test ($\alpha = .05$) was used to analyze the project quality and student productivity metrics for assignments with and without nudges from `class-bot`.

7.4.1 RQ1: Quality

To observe the impact of automated nudges incorporating *developer recommendation choice architectures* on code quality, we analyzed the grade and number of points deducted on student assignments with and without `class-bot` recommendations. The code quality results are presented in Table 7.1. For grading, students received an average of 2.6 points higher on assignments with automated nudges. Additionally, we found students received significantly higher scores on projects with automated nudges ($p = 0.0097$). This demonstrates notifications from the `class-bot` system improved the overall quality of programming assignments. Additionally, while there was not a significant difference, we noticed projects without automated nudges had an average of 11 more points deducted compared to those with `class-bot` interventions.

7.4.2 RQ2: Productivity

To discover the impact of `class-bot` on development productivity, we analyzed the total commits, code churn, first commit time, and last commit time on student project repositories. Our results, presented in Table 7.2, show that projects with nudges increased development activity approximately three more commits and 897 more lines of code changed for each commit. Additionally, we found this approach improved the timing of student work by encouraging participants to begin work around six days earlier and to submit assignments 12 hours sooner.[4]. Furthermore, we found that `class-bot` significantly

[4]A negative result here indicates assignments submitted late after the deadline

increased the number of lines of code modified in commits (***code churn***, $p = 0.0348$) and significantly prevented procrastination (***first commit***, $p < 0.0001$). Overall, this signifies that nudges from `class-bot` improved the productivity of students working on programming assignments.

Table 7.1 `class-bot` Quality Results

	Nudge?	**Mean**	**Median**	***p-value***
Grade* **	No	74.29	87.66	-
	Yes	76.89	95	***0.0097****
Deductions	No	-20.71	-5	-
	Yes	-9.43	0	0.0672

*** denotes statistically significant results (p-*value* < 0.05)

Table 7.2 `class-bot` Productivity Results

	Nudge?	**Mean**	**Median**	***p-value***
Commits	No	9.84	7	-
	Yes	12.64	9	0.1646
Code Churn* **	No	205.03	4	-
	Yes	1101.57	11	***0.0348****
First Commit* **	No	8.32	7.41	-
(days)	Yes	1.99	5.94	< ***0.0001****
Last Commit	No	-21.72	-1.60	-
(hours)	Yes	-9.67	-2.47	0.7909

*** denotes statistically significant results (p-*value* < 0.05)

7.4.3 Survey

To further evaluate the `class-bot` system, we surveyed students to collect feedback on their experience with this bot on their repositories. Figure 7.2 shows Likert scale questions results on students' perceptions of the usefulness of the automated nudges. Overall, most respondents (88.9%) found the notifications from `class-bot` at least moderately useful. Participants commented on the `class-bot` interventions on their projects noting they "*liked the class-bot updates*" (P1) and used it to "*make sure everything was running smoothly*" (P6). We analyzed open-ended responses and derived two themes to provide insight into improving automated tools for recommending developer behaviors to programmers.

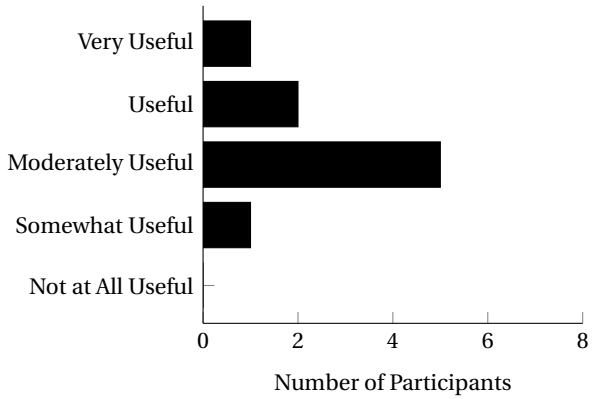


Figure 7.2 Survey Results on the Usefulness of `class-bot`

7.4.3.1 Validation Frequency

One area of improvement for the `class-bot` system is to improve how students verify their project. Even though we found students started programming assignments significantly earlier when automated issues were present, we also found that students reported waiting until the end of the development process to validate their work met the project expectations. For example, several students noted that, while they they found `class-bot` notifications useful, they "*didn't really check them until the final day*" (P1) and "*checked it once at the end to make sure everything was correct but thats it*" (P7). Furthermore, P7, who ranked the system the lowest as Somewhat Useful, added "*I kept track of my own progress so I did not feel the need for this*". This problem also exists in software engineering where, while research shows validating and verifying software through out the development process improves code quality [Wal89], professional developers often face challenges and delay validating their program meets project requirements [Gar08]. This motivates the need for automated recommendations to better integrate into development workflows by encouraging programmers to validate their code more frequently.

7.4.3.2 Update Frequency

Similarly, participants desired more frequent updates from `class-bot`. For example, students mentioned "*I liked it but would like it more if it could be updated more often, or maybe later in the day*" (P1) and "*the class bot didn't update frequently enough*" (P2). Additionally, P4 thought the bot was broken due to infrequent updates and then "*did not trust it when it started to work*". Prior work in software engineering, including the results of the naive *telemarketer design* study, shows that untimely notifications discourage developers from adopting recommendations [Vir09; Bro19]. Furthermore, researchers found presenting static analysis tool output more frequently encouraged programmers to fix more bugs [Dis19]. While this system incorporated temporal locality into its design by consistently updating GitHub issues, students desired even more frequent feedback and updates from the system to support them in their work. Providing frequent updates to projects can improve the integration of automated recommendations into programmers' workflows and increase the effectiveness of digital nudges for

improving the adoption of developer behaviors.

7.4.4 Summary

The results from this study show that that nudges are useful for improving student programming behaviors. Specifically, we found that `class-bot` significantly improved student grades, increased the number of changes to repositories, and encouraged students to start programming earlier. Participants also reported this system was useful and provided insight for improving automated nudges to encourage better software engineering behaviors by better incorporating student workflows. We analyzed this feedback and present two themes based on how frequently students checked the automated notifications from `class-bot` and how often the issues were updated.

7.5 Discussion

Ultimately, this work shows incorporating *developer recommendation choice architectures* into automated recommendations can improve the behavior and decision-making of developers. To support this, I implemented `class-bot` as a recommender system incorporating the conceptual framework. This system was designed to include each of the choice architecture principles by generating *actionable* GitHub issues with automated updates to recommend development tasks, providing straightforward and visual *feedback* to track progress on projects, and presenting updates with persistent and convenient *locality* on issues in the repository during development. To analyze the impact of this approach on the behavior of programmers, I developed `class-bot` to encourage student programmers to adhere to software engineering processes in their work.

Research shows adopting software processes is a beneficial developer behavior for improving the quality of applications [Cla97]. However, Computer Science education research shows failing to adhere to software engineering processes leads to high failure and attrition in early programming courses [Bea05]. Moreover, the adoption problem for this developer behavior translates into industry, where professional software engineers also frequently fail to follow advised development practices and inappropriately allocate the time and effort for completing tasks [Boe84]. By encouraging students to follow software engineering processes, we found `class-bot` improved the code quality and developer productivity of students by boosting grades, increasing code contributions, and preventing procrastination. I believe these automated nudges incorporating *developer recommendation choice architectures* are a step towards encouraging Computer Science students to adopt useful developer behaviors while working on programming projects, and thus improving the behavior of software engineers.

CHAPTER

8

CONCLUSION

8.1 Thesis Statement Revisited

This dissertation presents research to evaluate and support my thesis statement (Chapter 1). The thesis of this dissertation is:

> 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.

To support this claim, my research makes the several contributions to advance knowledge on designing automated recommendations to increase adoption of developer behaviors. First, I conducted a *set of experiments* examining *peer interactions* and the naive *telemarketer design* as developer recommendation approaches to determine what makes an effective recommendation to software engineers (Chapter 4). The results from these studies show that user-to-user recommendations are effective because of their ability to foster receptiveness, specifically desire and familiarity, while naive bots are ineffective because of their inability to conform to social context and development workflows.

The preliminary studies motivated the need for new techniques to recommend developer behaviors to programmers as opportunities for peer interactions decline and simple systems such as `tool-recommender-bot` generate intrusive and unsuccessful notifications. To improve the effectiveness of automated recommendations, I introduce *developer recommendation choice architectures*, a *conceptual framework* for designing effective automated recommendations to developers by applying

concepts from nudge theory (Chapter 5). This framework utilizes practical tools for choice architecture [Joh12] to obtain principles for creating automated recommendations to improve the environment surrounding developers' decisions, and the formative evaluation shows software engineers preferred actionable suggestions compared to static ones.

To evaluate *developer recommendation choice architectures*, I devised a *set of experiments* analyzing an existing recommender system, GitHub *suggested changes*, through the lens of this framework to discover its impact on developer preferences and development activity (Chapter 6). These evaluations show that developers significantly prefer to receive tool recommendations from systems incorporating *developer recommendation choice architectures* than those that don't, and that this framework is useful for recommending a variety of changes and effective for improving development activity and collaboration between developers during reviews.

Finally, to further assess this framework, I developed a novel *automated recommender system*, `class-bot`, that incorporates *developer recommendation choice architectures* principles to generate digital nudges recommending useful developer behaviors to Computer Science students (Chapter 7). The findings show that this bot was able to improve code quality and the productivity of students on their programming assignments by encouraging them to follow software engineering processes, a behavior that is also often ignored by professional developers in industry.

In this work, I use concepts from nudge theory to encourage software engineers to adopt better behaviors. I analyze existing and novel recommendation techniques, define *developer recommendation choice architectures* as a method to enhance automated recommendations, investigate how this framework impacts existing recommender systems, and develop a bot to show this approach influences programmer behavior by improving code quality and developer productivity.

8.2 Future Work

This dissertation motivates, presents, and evaluates *developer recommendation choice architectures*, a novel framework that incorporates nudge theory to design effective automated recommendations to encourage developer behaviors. Future directions of this research can further enhance automated developer recommendations by *analyzing the behavior* of software engineers and *developing new tools* and techniques to improve the decision-making, behavior, and productivity of programmers in their work.

8.2.1 Behavior

Examples of future studies related to this work involve using *developer recommendation choice architectures* to suggest additional developer behaviors, predicting the actions of programmers to proactively make recommendations to prevent bad behaviors, and exploring other disciplines to improve the behavior and decision-making of software engineers.

- **Recommending developer behaviors.** The research presented in this dissertation explores recommendations for several developer behaviors, namely tool adoption (Chapter 4, Chapter 6.1), code

improvements (Chapter 6.2), and following software engineering processes (Chapter 7). Future directions of this work can examine the following question: how can *developer recommendation choice architectures* impact the adoption of other developer behaviors? For example, research shows software engineers often ignore beneficial development practices such as pair programming [Lui10], software dependency updates [Mir17a], software migration [Sma21], and more. Future work can explore using the framework presented in this dissertation to design developer recommendations for additional tools and practices.

- **Predicting developer behavior.** For the most part, my research is largely reactive in that it makes recommendations to suggest developer behaviors after programmers have completed a programming task inefficiently. To improve the effectiveness of automated recommendations to developers, future work can explore developing proactive nudges to predict the actions of developer and present suggestions before bad behaviors occur. For example, machine learning techniques such as collaborative filtering [MH12b] or Bayesian user modeling [Hor98] can be applied to analyze previous development activities of software engineers and anticipate poor decisions in advance. Then, recommender bots can proactively suggest better practices to help developers avoid poor practices and increase adoption of beneficial behaviors.

- **Interdisciplinary behavioral concepts.** Nudge theory is a behavioral science concept for improving human decision-making and behavior. To advance this research, future work can explore techniques for modifying human behavior from other disciplines. For example, behavioral science also posits *shoves* as an alternative to nudges that force humans to adopt target behaviors [Arn15]. Likewise, user experience and human factors research submits *dark patterns*, or deceptive user interface designs, as another form of indirect influence to alter user behavior online.[1] Prospective studies can explore multidisciplinary techniques to influence human behavior and apply these concepts to influence the behavior of software engineers.

8.2.2 Tools

Potential advancements of this research also include subsequent studies examining using *developer recommendation choice architectures* to improve the adoption of systems produced by research in industry, enhance the output of development tools, and creating new bots to encourage the adoption of developer behaviors.

- **Improving research products.** As mentioned in Section 3.1.1, there are a variety of factors that contribute to the developer behavior adoption problem. While the research presented in this dissertation primarily focuses on improving the decision-making of software engineers, future work can investigate ways to improve other barriers to the adoption of development tools and practices. For example, studies show the development products and tools developed by researchers are often ineffective for industry practitioners [Nor10; Woh13]. Future research can explore ways to

[1] `https://darkpatterns.org/index.html`

bridge the research-practice gap and improve the adoption of developer behaviors by convincing researchers and toolsmiths to develop and evaluate products relevant to software engineers that accommodate developer needs. By doing this, we can increase the awareness of software engineering research, techniques, and findings and adoption of useful tools and practices in industry.

- **Tool output.** This dissertation introduces *developer recommendation choice architectures* as a framework to design recommendations encouraging developers to adopt code fixes (GitHub *suggested changes*) and software engineering processes (`class-bot`) to apply to their work. Another application of this research is to enhance how problems are presented to programmers. Research shows developers often ignore warnings for code smells, or potential problems within code [Yam13], and avoid static analysis tools due to incomprehensible output [Joh13]. Prior work has also explored ways to improve code smell notifications, including techniques to provide actionable static analysis alerts that mitigate false positives [Hec09], lightweight visualizations to inspect smells during code reviews [Par08], ambient interactive designs to support identifying and refactoring code smells [MH10], and developer-driven code smell prioritization to rank bugs based on criticality [Pec20]. To reduce code smells and increase the quality of code, future work can build on this research by using *developer recommendation choice architectures* to design code smell notifications and nudge developers to fix reported issues and further encourage the adoption of useful tools and practices.

- **Nudge bots.** To further improve the behavior of developers, future work can develop automated tools incorporating *developer recommendation choice architectures* principles and concepts from nudge theory to make recommendations using different interventions. For example, this work examines recommendations on GitHub through automated pull requests (Chapter 4.2), *suggested changes* (Chapter 6), and automated issues (Chapter 7). Future directions of this work can explore delivering recommendations to software engineers through similar techniques on other code hosting websites like GitLab[2] or BitBucket.[3] Additionally, future work can recommend developer behaviors to software engineers through mechanisms evaluated in prior work such as StackOverflow posts [Cai19],[4] instant messages through Slack [Lin16],[5] posts to social media [Beg10] or blog sites [Bar15],[6] and other online programming communities. Furthermore, research can also explore other interventions such as chatbots [Cer19] or automated program repair techniques [Mon19] to recommend developer behavior. These examples can provide further methods to incorporate *developer recommendation choice architectures* recommendations to encourage the adoption of beneficial behaviors by developers.

[2] `https://about.gitlab.com/`
[3] `https://bitbucket.org/`
[4] `https://stackoverflow.com/`
[5] `https://slack.com/`
[6] `https://news.ycombinator.com/`

8.3 Epilogue

"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?'"[7]

[7] https://twitter.com/gvwilson/status/1142245508464795649?s=20

BIBLIOGRAPHY

[Acq17] Acquisti, A. et al. “Nudges for privacy and security: Understanding and assisting users’ choices online”. *ACM Computing Surveys (CSUR)* **50**.3 (2017), p. 44.

[Agg14] Aggarwal, K. et al. “Co-Evolution of Project Documentation and Popularity within Github”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 360–363.

[Ahm08] Ahmadi, N. et al. “A survey of social software engineering”. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*. IEEE. 2008, pp. 1–12.

[Alo00] Alonso, G. & Hagen, C. “Exception Handling in Workflow Management Systems”. *IEEE Transactions on Software Engineering* **18**.10 (2000), pp. 943–958.

[And96] 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* (1996), pp. 174–187.

[Ani18] Aniche, M. et al. “How Modern News Aggregators Help Development Communities Shape and Share Knowledge”. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 499–510.

[Ank06] Ankolekar, A. et al. “Supporting online problem-solving communities with the semantic web”. *Proceedings of the 15th international conference on World Wide Web*. ACM. 2006, pp. 575–584.

[Arn15] Arneson, R. J. “Nudge and Shove.” *Social Theory Practice* **41**.4 (2015), pp. 668 –691.

[Aye10] Ayewah, N. & Pugh, W. “The google findbugs fixit”. *Proceedings of the 19th international symposium on Software testing and analysis*. ACM. 2010, pp. 241–252.

[Bac13] Bacchelli, A. & Bird, C. “Expectations, outcomes, and challenges of modern code review”. *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 712–721.

[Bal13] Balachandran, V. “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 931–940.

[Bar15] Barik, T. et al. “I heart Hacker News: expanding qualitative research findings by analyzing social news websites”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 882–885.

[Bar16] Barik, T. et al. “A perspective on blending programming environments and games: Beyond points, badges, and leaderboards”. *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*. IEEE. 2016, pp. 134–142.

[Bar18] Barik, T. et al. “How should compilers explain problems to developers?” *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 633–643.

[Bav14] Bavota, G. et al. “Recommending Refactoring Operations in Large Software Systems”. *Recommendation Systems in Software Engineering*. Ed. by Robillard, M. P. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 387–419.

[Bea05] Beaubouef, T. & Mason, J. “Why the high attrition rate for computer science students: some thoughts and observations”. *ACM SIGCSE Bulletin* **37**.2 (2005), pp. 103–106.

[Beg08] Begel, A. & Nagappan, N. “Pair Programming: What’s in It for Me?” *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’08. Kaiserslautern, Germany: Association for Computing Machinery, 2008, 120–128.

[Beg14] Begel, A. & Zimmermann, T. “Analyze This! 145 Questions for Data Scientists in Software Engineering”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 12–23.

[Beg10] Begel, A. et al. “Social media for software engineering”. *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, pp. 33–38.

[Bel14] Beller, M. et al. “Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?” *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 202–211.

[Bes17] Beschastnikh, I. et al. “Accelerating software engineering research adoption with analysis bots”. *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. IEEE. 2017, pp. 35–38.

[Bes10] Bessey, A. et al. “A few billion lines of code later: using static analysis to find bugs in the real world”. *Communications of the ACM* **53**.2 (2010), pp. 66–75.

[Bis13] Bissyandé, T. F. et al. “Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub”. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 2013, pp. 188–197.

[Bla14] Blackwell, A. et al. “Collaboration and learning through live coding (Dagstuhl Seminar 13382)”. *Dagstuhl Reports*. Vol. 3. 9. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2014.

[Bla13] Blatt, M. & Norman, A. “Email, communication and more: How software engineers use and reflect upon email at the workplace”. MA thesis. 2013.

[Boe84] Boehm, B. W. “Software Engineering Economics”. *IEEE Transactions on Software Engineering* **SE-10**.1 (1984), pp. 4–21.

[Bor18] Borges, H. & Valente, M. T. “What’s in a GitHub star? understanding repository starring practices in a social coding platform”. *Journal of Systems and Software* **146** (2018), pp. 112–129.

[Bri03] Briand, L. C. “Software documentation: how much is enough?” *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* 2003, pp. 13–15.

[Bro19] Brown, C. & Parnin, C. “Sorry to bother you: designing bots for effective recommendations”. *Proceedings of the 1st International Workshop on Bots in Software Engineering*. IEEE Press. 2019, pp. 54–58.

[Bro20] Brown, C. & Parnin, C. “Comparing Different Developer Behavior Recommendation Styles”. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ICSEW’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 78–85.

[Bro20] Brown, C. & Parnin, C. “Sorry to Bother You Again: Developer Recommendation Choice Architectures for Designing Effective Bots”. *2020 IEEE/ACM 2nd International Workshop on Bots in Software Engineering (BotSE)*. IEEE Press. 2020.

[Bro20] Brown, C. & Parnin, C. “Understanding the Impact of GitHub Suggested Changes on Recommendations between Developers”. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, 1065–1076.

[Cab15] Cabot, J. et al. “Exploring the use of labels to categorize issues in open-source software projects”. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2015, pp. 550–554.

[Cai19] Cai, L. et al. “AnswerBot: An Answer Summary Generation Tool Based on Stack Overflow”. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, 1134–1138.

[Cao10] Cao, J. et al. “A debugging perspective on end-user mashup programming”. *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*. IEEE. 2010, pp. 149–156.

[Cao12] Cao, J. et al. “From barriers to learning in the Idea Garden: An empirical study”. *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2012, pp. 59–66.

[Cap15] Caprigliano, G. D. B. D. “Nudging in the UK, in the USA, in Denmark, in Italy: an international comparison of behavioural insights teams”. Bachelor’s Degree Thesis. Libera Università Internazionale degli Studi Sociali “Guido Carli”, 2015.

[Car20] Carvalho, A. et al. “C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests”. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 161–171.

[Cer19] Cerezo, J. et al. “Building an expert recommender chatbot”. *Proceedings of the 1st International Workshop on Bots in Software Engineering*. IEEE Press. 2019, pp. 59–63.

[Cla97] Clark, B. K. *The Effects of Software Process Maturity on Software Development Effort*. 1997.

[Coc01] Cockburn, A. & Williams, L. “Extreme Programming Examined”. Ed. by Succi, G. & Marchesi, M. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. Chap. The Costs and Benefits of Pair Programming, pp. 223–243.

[Coh06] Cohen, J. et al. *Best kept secrets of peer code review*. Smart Bear Somerville, 2006.

[Cos84] Costello, S. H. “Software Engineering under Deadline Pressure”. *SIGSOFT Softw. Eng. Notes* **9**.5 (1984), 15–19.

[Cub03] Cubranic, D. & Murphy, G. C. “Hipikat: recommending pertinent software development artifacts”. *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 408–418.

[Dab12] Dabbish, L. et al. “Social coding in GitHub: transparency and collaboration in an open software repository”. *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM. 2012, pp. 1277–1286.

[Dab06] Dabbish, L. A. & Kraut, R. E. “Email Overload at Work: An Analysis of Factors Associated with Email Strain”. *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. CSCW ’06. Banff, Alberta, Canada: Association for Computing Machinery, 2006, 431–440.

[DNM13] Danescu-Niculescu-Mizil, C. et al. *A Computational Approach to Politeness with Application to Social Factors*. 2013. arXiv: 1306.6078 [cs.CL].

[Dav10] Davidson, J. et al. “The YouTube Video Recommendation System”. *Proceedings of the Fourth ACM Conference on Recommender Systems*. RecSys ’10. Barcelona, Spain: Association for Computing Machinery, 2010, 293–296.

[Dav20] Davis, J. K. & Crawford, J. “Is Technology Actually Making Things Better?” *Pairagraph* (2020). https://www.pairagraph.com/dialogue/354c72095d2f42dab92bf42726d785ff/1.

[DA06] De Alwis, B. & Murphy, G. C. “Using visual momentum to explain disorientation in the Eclipse IDE”. *Visual Languages and Human-Centric Computing (VL/HCC’06)*. IEEE. 2006, pp. 51–54.

[Dia03] Diaper, D. & Stanton, N. *The handbook of task analysis for human-computer interaction*. CRC Press, 2003.

[Dig90] Digman, J. M. “Personality Structure: Emergence of the Five-Factor Model”. *Annual Review of Psychology* **41**.1 (1990), pp. 417–440. eprint: https://doi.org/10.1146/annurev.ps.41.020190.002221.

[Dis19] Distefano, D. et al. “Scaling Static Analyses at Facebook”. *Commun. ACM* **62**.8 (2019), pp. 62–70.

[Duf11] Duflo, E. et al. “Nudging farmers to use fertilizer: Theory and experimental evidence from Kenya”. *American economic review* **101**.6 (2011), pp. 2350–90.

[Ebe08] Ebert, C. et al. “Managing risks in global software engineering: principles and practices”. *2008 IEEE International Conference on Global Software Engineering*. IEEE. 2008, pp. 131–140.

[Edw14] Edwards, C. et al. “Is that a bot running the social media feed? Testing the differences in perceptions of communication quality for a human agent and a bot agent on Twitter”. *Computers in Human Behavior* **33** (2014), pp. 372–376.

[Edw09] Edwards, S. H. et al. “Comparing Effective and Ineffective Behaviors of Student Programmers”. *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*. ICER ’09. Berkeley, CA, USA: Association for Computing Machinery, 2009, 3–14.

[Erl19] Erlenhov, L. et al. “Current and Future Bots in Software Development”. *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. 2019, pp. 7–11.

[Erl20] Erlenhov, L. et al. “An Empirical Study of Bots in Software Development: Characteristics and Challenges from a Practitioner’s Perspective”. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, 445–455.

[Esp02] Espinosa, A. et al. “Shared mental models, familiarity, and coordination: A multi-method study of distributed software teams”. *ICIS 2002 Proceedings* (2002), p. 39.

[Eva02] Evans, D. & Larochelle, D. “Improving security using extensible lightweight static analysis”. *IEEE software* **19**.1 (2002), pp. 42–51.

[Far12] Faridi, M. S. et al. “Human persuasion integration in software development lifecycle (SDLC)”. *International Journal of Computer Science Issues (IJCSI)* **9**.4 (2012), p. 65.

[Fen06] Feng, B. & MacGeorge, E. L. “Predicting Receptiveness to Advice: Characteristics of the Problem, the Advice-Giver, and the Recipient”. *Southern Communication Journal* **71**.1 (2006), pp. 67–85. eprint: `https://doi.org/10.1080/10417940500503548`.

[Fis84] Fischer, G. et al. “Active help systems”. *Readings on Cognitive Ergonomics — Mind and Computers: Proceedings of the 2nd European Conference Gmunden, Austria, September 10–14, 1984*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 115–131.

[Fle13] Fleming, S. D. et al. “An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks”. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **22**.2 (2013), p. 14.

[Fog09] Fogg, B. “Creating Persuasive Technologies: An Eight-step Design Process”. *Proceedings of the 4th International Conference on Persuasive Technology*. Persuasive ’09. Claremont, California, USA: ACM, 2009, 44:1–44:6.

[For02] Forward, A. & Lethbridge, T. C. “The relevance of software documentation, tools and technologies: a survey”. *Proceedings of the 2002 ACM symposium on Document engineering*. ACM. 2002, pp. 26–33.

[Gar08] García, J. et al. “Ten factors that impede improvement of verification and validation processes in software intensive organizations”. *Software Process: Improvement and Practice* **13**.4 (2008), pp. 335–343. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spip.395`.

[Gar15] Gardikiotis, A. & Crano, W. “Persuasion Theories”. *International Encyclopedia of the Social Behavioral Sciences* (2015).

[Gas16] Gasparic, M. & Janes, A. “What recommendation systems for software engineering recommend: A systematic literature review”. *Journal of Systems and Software* **113** (2016), pp. 101–113.

[Gita] GitHub. “About issues”. *GitHub Help* ().

[Gitb] GitHub. “Creating a pull request”. *GitHub Help* ().

[Git18a] GitHub. “Suggested Changes (public beta)”. *The GitHub Blog* (2018).

[Git18b] GitHub. “Suggested changes: what we’ve learned so far”. *The GitHub Blog* (2018).

[GU16] Gomez-Uribe, C. A. & Hunt, N. “The netflix recommender system: Algorithms, business value, and innovation”. *ACM Transactions on Management Information Systems (TMIS)* **6**.4 (2016), p. 13.

[Goo92] Goodman, P. S. & Shah, S. “Familiarity and work group outcomes”. *Group process and productivity* (1992), pp. 276–298.

[Gor15] Gordon, M. & Guo, P. J. “Codepourri: Creating visual coding tutorials using a volunteer crowd of learners”. *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE. 2015, pp. 13–21.

[Gou15] Gousios, G. et al. “Work Practices and Challenges in Pull-Based Development: The Integrator’s Perspective”. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 358–368.

[Gou14a] Gousios, G. & Zaidman, A. “A Dataset for Pull-Based Development Research”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 368–371.

[Gou14b] Gousios, G. et al. “An Exploratory Study of the Pull-Based Software Development Model”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 345–355.

[Guz14] Guzman, E. et al. “Sentiment Analysis of Commit Comments in GitHub: An Empirical Study”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 352–355.

[Han12] Hanks, A. S. et al. “Healthy convenience: nudging students toward healthier choices in the lunchroom”. *Journal of Public Health* **34**.3 (2012), pp. 370–376. eprint: `http://oup.prod.sis.lan/jpubhealth/article-pdf/34/3/370/12782601/fds003.pdf`.

[Hap08] Happel, H.-J. & Maalej, W. “Potentials and Challenges of Recommendation Systems for Software Development”. *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. RSSE ’08. Atlanta, Georgia: Association for Computing Machinery, 2008, 11–15.

[Hat21] Hata, H. et al. *GitHub Discussions: An Exploratory Study of Early Adoption*. 2021. arXiv: `2102.05230 [cs.SE]`.

[Hec09] Heckman, S. & Williams, L. “A Model Building Process for Identifying Actionable Static Analysis Alerts”. *2009 International Conference on Software Testing Verification and Validation*. 2009, pp. 161–170.

[Hec11] Heckman, S. & Williams, L. “A systematic literature review of actionable alert identification techniques for automated static code analysis”. *Information and Software Technology* **53**.4 (2011). Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing, pp. 363 –387.

[Her07] Herbsleb, J. D. “Global Software Engineering: The Future of Socio-technical Coordination”. *Future of Software Engineering (FOSE ’07)*. IEEE, 2007, pp. 188–198.

[Hil15] Hill, J. et al. “Real conversations with artificial intelligence: A comparison between human–human online conversations and human–chatbot conversations”. *Computers in Human Behavior* **49** (2015), pp. 245 –250.

[Hil17] Hilton, M. et al. “Trade-offs in continuous integration: assurance, security, and flexibility”. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 197–207.

[Hor98] Horvitz, E. et al. “The LumièRe Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users”. *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*. UAI’98. Madison, Wisconsin: Morgan Kaufmann Publishers Inc., 1998, pp. 256–265.

[Hua18] Huang, N. et al. “Digital Nudging for Online Social Sharing: Evidence from A Randomized Field Experiment”. *Proceedings of the 51st Hawaii International Conference on System Sciences*. 2018.

[Hwa15] Hwang, E. H. et al. “Knowledge Sharing in Online Communities: Learning to Cross Geographic and Hierarchical Boundaries”. *Organization Science* **26**.6 (2015), pp. 1593–1611. eprint: `https://doi.org/10.1287/orsc.2015.1009`.

[SWEBOK] IEEE Computer Society et al. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.

[Imt19] Imtiaz, N. et al. "Challenges with responding to static analysis tool alerts". *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 245–249.

[Izq15] Izquierdo, J. L. C. et al. "Gila: Github label analyzer". *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 479–483.

[Jac01] Jackson, T. et al. "The cost of email interruption". *Journal of systems and information technology* (2001).

[Jas04] Jaspers, M. W. et al. "The think aloud method: a guide to user interface design". *International Journal of Medical Informatics* **73**.11 (2004), pp. 781 –795.

[Jen17] Jensen, K. B. & Helles, R. "Speaking into the system: Social media and many-to-one communication". *European Journal of Communication* **32**.1 (2017), pp. 16–25. eprint: `https://doi.org/10.1177/0267323116682805`.

[Jia17] Jiang, J. et al. "Why and how developers fork what from whom in GitHub". *Empirical Software Engineering* **22**.1 (2017), pp. 547–578.

[Joh13] Johnson, B. et al. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 672–681.

[Joh15] Johnson, B. et al. "Bespoke Tools: Adapted to the Concepts Developers Know". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 878–881.

[Joh12] Johnson, E. J. et al. "Beyond nudges: Tools of a choice architecture". *Marketing Letters* **23**.2 (2012), pp. 487–504.

[Jor14] Jordan, B. et al. "Designing Interventions to Persuade Software Developers to Adopt Security Tools". *Proceedings of the CCS Workshop on Security Information Workers*. 2014.

[Kal15] Kalliamvakou, E. et al. "Open Source-Style Collaborative Development Practices in Commercial Projects Using GitHub". *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 574–585.

[Kay98] Kay, D. G. "Large Introductory Computer Science Classes: Strategies for Effective Course Management". *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '98. Atlanta, Georgia, USA: Association for Computing Machinery, 1998, 131–134.

[Ko06] Ko, A. J. et al. "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks". *IEEE Transactions on software engineering* 12 (2006), pp. 971–987.

[Koc06] Kocher, M. G. & Sutter, M. “Time is money—Time pressure, incentives, and the quality of decision-making”. *Journal of Economic Behavior & Organization* **61**.3 (2006), pp. 375–392.

[Kon12] Konstan, J. A. & Riedl, J. “Recommender systems: from algorithms to user experience”. *User modeling and user-adapted interaction* **22**.1-2 (2012), pp. 101–123.

[Kri18] Krishna, R. et al. “What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2018, pp. 306–315.

[Lay07] Layman, L. et al. “Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools”. *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE. 2007, pp. 176–185.

[Le 12] Le Goues, C. et al. “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each”. *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Glinz, M. et al. IEEE Computer Society, 2012, pp. 3–13.

[Lee83] Leech, G. *Principles of Pragmatics*. Longman linguistics library ; title no. 30. Longman, 1983.

[Leu10] Leung, A. K. yee & Chiu, C. yue. “Multicultural Experience, Idea Receptiveness, and Creativity”. *Journal of Cross-Cultural Psychology* **41**.5-6 (2010), pp. 723–741. eprint: `https://doi.org/10.1177/0022022110361707`.

[Li10] Li, L. et al. “A Contextual-Bandit Approach to Personalized News Article Recommendation”. *Proceedings of the 19th International Conference on World Wide Web*. WWW ’10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, 661–670.

[Li18] Li, L. et al. “How Are Issue Units Linked? Empirical Study on the Linking Behavior in GitHub”. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Nara, Japan: IEEE Press, 2018, pp. 386–395.

[Li15] Li, P. L. et al. “What makes a great software engineer?” *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 700–710.

[Lin16] Lin, B. et al. “Why Developers Are Slacking Off: Understanding How Software Teams Use Slack”. *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*. CSCW ’16 Companion. San Francisco, California, USA: Association for Computing Machinery, 2016, 333–336.

[Lin03] Linden, G. et al. “Amazon.com recommendations: item-to-item collaborative filtering”. *IEEE Internet Computing* **7**.1 (2003), pp. 76–80.

[Lin00a] Linton, F. et al. “OWL: A recommender system for organization-wide learning”. *Educational Technology & Society* **3**.1 (2000), pp. 62–76.

[Lin00b] Linton, F. et al. “OWL: A recommender system for organization-wide learning”. *Educational Technology & Society* **3**.1 (2000), pp. 62–76.

[Lui10] Lui, K. M. et al. “Pair Programming: Issues and Challenges”. *Agile Software Development: Current Research and Future Directions*. Ed. by Dingsøyr, T. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 143–163.

[Maa14] Maalej, W. et al. “On the Comprehension of Program Comprehension”. *ACM Trans. Softw. Eng. Methodol.* **23**.4 (2014).

[Mad20] Maddila, C. et al. *Nudge: Accelerating Overdue Pull Requests Towards Completion*. 2020. arXiv: 2011.12468 [cs.SE].

[Mad01] 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 (2001), pp. 1149–1187.

[Mak11] Makabee, H. *How Decision Fatigue Affects the Efficacy of Programmers*. Effective Software Design. 2011.

[Mal95] Maltzahn, C. “Community Help: Discovering Tools and Locating Experts in a Dynamic Environment”. *Conference Companion on Human Factors in Computing Systems*. CHI ’95. Denver, Colorado, USA: ACM, 1995, pp. 260–261.

[Män08] Mäntylä, M. V. & Lassenius, C. “What types of defects are really discovered in code reviews?” *IEEE Transactions on Software Engineering* **35**.3 (2008), pp. 430–448.

[Mao15] Mao, K. et al. “Developer Recommendation for Crowdsourced Software Development Tasks”. *2015 IEEE Symposium on Service-Oriented System Engineering*. 2015, pp. 347–356.

[Mar19] Marcilio, D. et al. “Are static analysis violations really fixed? a closer look at realistic usage of sonarqube”. *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 209–219.

[McD13] McDonald, N. & Goggins, S. “Performance and Participation in Open Source Software on GitHub”. *CHI ’13 Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’13. Paris, France: Association for Computing Machinery, 2013, pp. 13–144.

[McN18] McNamara, A. et al. “Does ACM’s Code of Ethics Change Ethical Decision Making in Software Development?” *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 729–733.

[McN06] McNee, S. M. et al. “Being Accurate is Not Enough: How Accuracy Metrics Have Hurt Recommender Systems”. *CHI ’06 Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’06. Montréal, Québec, Canada: ACM, 2006, pp. 1097–1101.

[Men18] Meng, N. et al. “Secure Coding Practices in Java: Challenges and Vulnerabilities”. *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 372–383.

[Mid18] Middleton, J. et al. “Which Contributions Predict Whether Developers Are Accepted into Github Teams”. *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR ’18. Gothenburg, Sweden: ACM, 2018, pp. 403–413.

[Mir17a] Mirhosseini, S. & Parnin, C. “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?” *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 84–94.

[Mir17b] Mirsch, T. et al. “Digital nudging: Altering user behavior in digital environments”. *Proceedings der 13. Internationalen Tagung Wirtschaftsinformatik (WI 2017)* (2017), pp. 634–648.

[Mon19] Monperrus, M. et al. “Repairnator Patches Programs Automatically”. *Ubiquity* **2019**.July (2019).

[Mun98] Munson, J. C. & Elbaum, S. G. “Code churn: a measure for estimating the impact of code change”. *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, pp. 24–31.

[Mur16a] Murgia, A. et al. “Among the Machines: Human-Bot Interaction on Social QA Websites”. *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’16. San Jose, California, USA: Association for Computing Machinery, 2016, 1272–1279.

[Mur16b] Murgia, A. et al. “Among the machines: Human-bot interaction on social q&a websites”. *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM. 2016, pp. 1272–1279.

[Mur10] Murphy, G. C. & Murphy-Hill, E. “What is trust in a recommender for software development?” *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM. 2010, pp. 57–58.

[MH08] Murphy-Hill, E. & Black, A. “Breaking the barriers to successful refactoring”. *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 421–430.

[MH12a] Murphy-Hill, E. “Continuous Social Screencasting to Facilitate Software Tool Discovery”. *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 1317–1320.

[MH10] Murphy-Hill, E. & Black, A. P. “An Interactive Ambient Visualization for Code Smells”. *Proceedings of the 5th International Symposium on Software Visualization*. SOFTVIS ’10. Salt Lake City, Utah, USA: Association for Computing Machinery, 2010, 5–14.

[MH11] Murphy-Hill, E. & Murphy, G. C. “Peer Interaction Effectively, Yet Infrequently, Enables Programmers to Discover New Tools”. *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*. CSCW ’11. Hangzhou, China: ACM, 2011, pp. 405–414.

[MH12b] Murphy-Hill, E. et al. “Improving software developers’ fluency by recommending development environment commands”. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 42.

[MH15] Murphy-Hill, E. et al. “How Do Users Discover New Tools in Software Development and Beyond?” *Computer Supported Cooperative Work (CSCW)* **24**.5 (2015), pp. 389–422.

[MH19a] Murphy-Hill, E. et al. “Do Developers Learn New Tools On The Toilet?” *Proceedings of the 2019 International Conference on Software Engineering*. 2019.

[MH19b] Murphy-Hill, E. et al. “What Predicts Software Developers’ Productivity?” *Transactions on Software Engineering* (2019).

[Nag05] Nagappan, N. & Ball, T. “Use of Relative Code Churn Measures to Predict System Defect Density”. *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: Association for Computing Machinery, 2005, 284–292.

[NIST02] National Institute of Standards and Technology. “The economic impacts of inadequate infrastructure for software testing”. *U.S. Department of Commerce Technology Administration* (2002).

[Ner05] Nerur, S. et al. “Challenges of migrating to agile methodologies”. *Communications of the ACM* **48**.5 (2005), pp. 72–78.

[Nie93] Nielsen, J. “Usability heuristics”. *Usability engineering* (1993), pp. 115–163.

[Nii11] Niinimaki, T. “Face-to-Face, Email and Instant Messaging in Distributed Agile Software Development Project”. *2011 IEEE Sixth International Conference on Global Software Engineering Workshop*. 2011, pp. 78–84.

[Nor10] Norman, D. A. “The research-Practice Gap: The need for translational developers”. *interactions* **17**.4 (2010), pp. 9–12.

[O’k02] O’keefe, D. J. “Persuasion”. *The International Encyclopedia of Communication* (2002).

[Ort15] Ortu, M. et al. “Would you mind fixing this issue?” *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Lassenius, C. et al. Cham: Springer International Publishing, 2015, pp. 129–140.

[Pad14] Padhye, R. et al. “A Study of External Community Contribution to Open-Source Projects on GitHub”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 332–335.

[Pak14] Pakdeetrakulwong, U. et al. “Recommendation systems for software engineering: A survey from software development life cycle phase perspective”. *The 9th International Conference for Internet Technology and Secured Transactions (ICITST-2014)*. 2014, pp. 137–142.

[Par08] Parnin, C. et al. “A Catalogue of Lightweight Visualizations to Support Code Smell Inspection”. *Proceedings of the 4th ACM Symposium on Software Visualization*. SoftVis ’08. Ammersee, Germany: Association for Computing Machinery, 2008, 77–86.

[Pec20] Pecorelli, F. et al. “Developer-Driven Code Smell Prioritization”. *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 220–231.

[Pie99] Pieters, R. & Warlop, L. “Visual attention during brand choice: The impact of time pressure and task motivation”. *International Journal of Research in Marketing* **16**.1 (1999), pp. 1 –16.

[Pon14] Ponzanelli, L. et al. “Prompter: A Self-Confident Recommender System”. *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 577–580.

[Pra19] Prana, G. A. A. et al. “Categorizing the content of GitHub README files”. *Empirical Software Engineering* **24**.3 (2019), pp. 1296–1327.

[Pur20] Purohit, A. K. et al. “Designing for Digital Detox: Making Social Media Less Addictive with Digital Nudges”. *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–9.

[Rah19] Rahman, A. et al. “The Seven Sins: Security Smells in Infrastructure as Code Scripts”. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 164–175.

[Rah17] Rahman, A. et al. “Which factors influence practitioners’ usage of build automation tools?” *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering*. IEEE Press. 2017, pp. 20–26.

[Red18] Reddy, T. S. “Impact of Bots on eCommerce and Bot Detection Methods”. Atlanta, GA: USENIX Association, 2018.

[Ree17] Reeves, D. B. et al. “Special Topic: What’s Worth Fighting against in Grading?.” *Educational Leadership* **74** (2017), pp. 42–45.

[Res97] Resnick, P. & Varian, H. R. “Recommender systems”. *Communications of the ACM* **40**.3 (1997), pp. 56–58.

[Rij20] Rijn, J. van. “Email is not dead. But email IS changing.” *Email Is Not Dead* (2020).

[Rob10] Robillard, M. et al. “Recommendation systems for software engineering”. *IEEE software* **27**.4 (2010), pp. 80–86.

[Rya06] Ryan, R. M. et al. *The Motivational Pull of Video Games: A Self-Determination Theory Approach*. 2006.

[Sad15] Sadowski, C. et al. “Tricorder: Building a program analysis ecosystem”. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 598–608.

[Sad18] Sadowski, C. et al. “Lessons from building static analysis tools at Google” (2018).

[Sch99] Schafer, J. B. et al. “Recommender systems in e-commerce”. *Proceedings of the 1st ACM conference on Electronic commerce*. 1999, pp. 158–166.

[Sen20] Sengupta, S. & Haythornthwaite, C. “Learning with comments: An analysis of comments and community on Stack Overflow”. *Proceedings of the 53rd Hawaii International Conference on System Sciences*. 2020.

[Sen04] Senyard, A. & Michlmayr, M. “How to have a successful free software project”. *11th Asia-Pacific Software Engineering Conference*. IEEE. 2004, pp. 84–91.

[She12] Shen, L. & Bigsby, E. “The effects of message features: content, structure and style”. *The SAGE handbook of persuasion developments in theory and practice* (2012).

[Sin12] Singer, L. & Schneider, K. “It was a bit of a race: Gamification of version control”. *2012 Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques (GAS)*. 2012, pp. 5–8.

[Sin16] Singer, L. *On the Diffusion of Innovations: How New Ideas Spread*. 2016.

[Sin14] Singer, L. et al. “Software engineering at the speed of light: how developers stay current using twitter”. *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 211–221.

[Sin17] Singh, D. et al. “Evaluating how static analysis tools can reduce code review effort”. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2017, pp. 101–105.

[Sma21] Smartsheet. “Checklists and Tools for Software Migration Planning” (2021).

[Smi16] Smith, E. K. et al. “Beliefs, Practices, and Personalities of Software Engineers: A Survey in a Large Software Company”. *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 15–18.

[Smi17] Smith, J. et al. “Flower: Navigating program flow in the IDE”. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017, pp. 19–23.

[Sni14] Snipes, W. et al. “Experiences Gamifying Developer Adoption of Practices and Tools”. *International Conference on Software Engineering, Software Engineering in Practice Track*. 2014.

[Sor19] Sorbo, A. D. et al. “"Won’t We Fix this Issue?" Qualitative Characterization and Automated Identification of Wontfix Issues on GitHub”. *CoRR* **abs/1904.02414** (2019). arXiv: 1904.02414.

[Spr19] Sprint, G. & Conci, J. “Mining GitHub classroom commit behavior in elective and introductory computer science courses”. *The Journal of Computing Sciences in Colleges* (2019), p. 76.

[Sta86] Stanfill, C. & Waltz, D. “Toward Memory-Based Reasoning”. *Commun. ACM* **29**.12 (1986), 1213–1228.

[Ste00] Sterne, J. & Priore, A. *Email marketing: using email to reach your target audience and build customer relationships*. John Wiley & Sons, Inc., 2000.

[Sto16] Storey, M.-A. & Zagalsky, A. “Disrupting developer productivity one bot at a time”. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 928–931.

[Str07] Stroggylos, K. & Spinellis, D. “Refactoring–Does It Improve Software Quality?” *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. 2007, pp. 10–10.

[Sun19] Sun, P. et al. “Mining Specifications from Documentation using a Crowd”. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 275–286.

[Tha09] Thaler, R. H. & Sunstein, C. R. *Nudge: Improving decisions about health, wealth, and happiness*. Penguin, 2009.

[Tha13] Thaler, R. H. et al. “Choice architecture”. *The behavioral foundations of public policy* (2013), pp. 428–439.

[Oct] *The State of the Octoverse*. `https://octoverse.github.com/`. 2018.

[Til03] Tilley, S. et al. “On the challenges of adopting ROTS software”. *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering*. 2003, pp. 3–6.

[Ton19] Tonder, R. van & Goues, C. L. “Towards s/Engineer/Bot: Principles for Program Repair Bots”. *Proceedings of the 1st International Workshop on Bots in Software Engineering*. BotSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 43–47.

[Tri17] Tricentis. *Software Fail Watch*. Tricentis, 2017.

[Tro18] Trockman, A. et al. “Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem”. *Proceedings of the 40th International Conference on Software Engineering*. ACM. 2018, pp. 511–522.

[Tur17] Turkle, S. *Alone together: Why we expect more from technology and less from each other*. Hachette UK, 2017.

[Twi05] Twidale, M. B. “Over the shoulder learning: supporting brief informal learning”. *Computer Supported Cooperative Work* **14**.6 (2005), pp. 505–547.

[Url18] Urli, S. et al. “How to design a program repair bot?: insights from the repairnator project”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM. 2018, pp. 95–104.

[Vas16] Vasilescu, B. et al. “The Sky is Not the Limit: Multitasking across GitHub Projects”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, 994–1005.

[Vir09] Viriyakattiyaporn, P. & Murphy, G. C. “Challenges in the user interface design of an IDE tool recommender”. *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE, 2009, pp. 104–107.

[Vir10] Viriyakattiyaporn, P. & Murphy, G. C. “Improving Program Navigation with an Active Help System”. *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON ’10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 27–41.

[Wal89] Wallace, D. R. & Fujii, R. U. “Software verification and validation: an overview”. *IEEE Software* **6**.3 (1989), pp. 10–17.

[Was81] Wasserman, A. I. “User Software Engineering and the Design of Interactive Systems”. *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 387–393.

[Wei16] Weinmann, M. et al. “Digital nudging”. *Business & Information Systems Engineering* **58**.6 (2016), pp. 433–436.

[Wes18] Wessel, M. et al. “The power of bots: Characterizing and understanding bots in OSS projects”. *Proceedings of the ACM on Human-Computer Interaction* **2**.CSCW (2018), p. 182.

[Whi00] Whittaker, J. A. “What is software testing? And why is it so hard?” *IEEE Software* **17**.1 (2000), pp. 70–79.

[Whi05] Whitworth, B. “Polite computing”. *Behaviour & Information Technology* **24**.5 (2005), pp. 353–363. eprint: `http://dx.doi.org/10.1080/01449290512331333700`.

[Wil15] Wilcox, C. “The Role of Automation in Undergraduate Computer Science Education”. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: Association for Computing Machinery, 2015, 90–95.

[Wis10] Wisdom, J. et al. “Promoting Healthy Choices: Information versus Convenience”. *American Economic Journal: Applied Economics* **2**.2 (2010), pp. 164–78.

[Wit15] Witschey, J. et al. “Quantifying Developers’ Adoption of Security Tools”. *Proceedings of Foundations of Software Engineering*. 2015.

[Woh13] Wohlin, C. “Empirical software engineering research with industry: Top 10 challenges”. *Proceedings of the 1st International Workshop on Conducting Empirical Studies in Industry*. IEEE Press. 2013, pp. 43–46.

[Woo19] Woo, A. *Decision-Making: The Most Undervalued Skill in Software Engineering*. Hacker Noon. 2019.

[Wyr21] Wyrich, M. et al. *2021 IEEE/ACM 2nd International Workshop on Bots in Software Engineering (BotSE 2021) (to appear)*. IEEE Press. 2021.

[Xia03] Xiao, J. et al. *Be quiet? evaluating proactive and reactive user interface assistants*. Tech. rep. Georgia Institute of Technology, 2003.

[Xia14] Xiao, S. et al. “Social Influences on Secure Development Tool Adoption: Why Security Tools Spread”. *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’14. Baltimore, Maryland, USA: ACM, 2014, pp. 1095–1106.

[Yam13] Yamashita, A. & Moonen, L. “Do developers care about code smells? An exploratory survey”. *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 242–251.

[Yu15] Yu, Y. et al. “Wait for it: determinants of pull request evaluation latency on GitHub”. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 367–371.

APPENDICES

APPENDIX

A

STUDY MATERIALS FOR CHAPTER 4

A.1 *"How Software Users Recommend Tools to Each Other"*

A.1.1 Study Script

Pre Task:

The dataset is picked from kaggle.com's Titanic competition:
`https://www.kaggle.com/c/titanic`

The Variable descriptions are given here:
`https://www.kaggle.com/c/titanic/data`

Answer any questions about the dataset.

Please do not use the internet to answer the questions.

Please work on the tasks together in pairs.

The time to spend on the task is 60 mins, which comes to 7 to 8 mins on each task. However, this is just a recommendation. (the participants were asked to do their best when answering the questions and not to bother about the time etc.)

Please use any tool you are comfortable with to answer these questions. This computer has Excel, Rstudio, Python, SAS JMP Pro 12, and MySQL Workbench. If you need anything else, we can download that as well.

The total time for the task is 60 mins. Please do not spend more than 45 mins in the training tasks which comes to around 7 to 8 mins on each question.

Tasks:

Student Task

Task: For a to e, please to describe the relationship. For f the factors should be ranked from the most significant to least significant. You can use mean, mode, etc. to explain the ranking.

Using training data: (45 mins)

a. What is the relationship between the (gender, age) and number of sibling/spouse (SibSp) traveling?

b. What is the relationship between the Title(you can find this in the name) and the number of children/parents (Parch) traveling?

c. What is the relationship between the Title(you can find this in the name) and the age and gender?

d. What is the relationship between (class, fair) and age?

e. What is the relationship between (the fare and class) and the city embarked?

f. Please rank all the factors (a-e) for their contribution to survival. The factors should be ranked from the most significant to least significant. You can use mean, mode, etc. to explain the ranking.

On the testing data: (10 mins)

Please find whether these people survived.

a. Chaffee, Mrs. Herbert Fuller (Carrie Constance Toogood)

b. Robins, Mr. Alexander A

c. Peltomaki, Mr. Nikolai Johannes

d. Abelseth, Mr. Olaus Jorgensen

e. Mulvihill, Miss. Bertha E

f. Thomas, Mr. John

g. Daniels, Miss. Sarah

h. Delalic, Mr. Redjo

LAS Analyst Task

Task: For a to c, please describe the relationship between the categories. For d the factors should be ranked from the most significant to least significant. You can use mean, mode, etc. to explain the ranking.

Using train.csv: (35 mins)

a. What is the relationship between the gender (Sex), age, and the number of siblings/spouse traveling (SibSp)?

b. What is the relationship between the Title (you can find this in the name- Mr., Mrs., Ms., Miss., Master., Dr., etc. There may be more than this) and the number of children/parents (Parch) traveling?

c. What is the relationship between the fare, class (Pclass), and age?

d. Rank the factors for their contribution to survival. The factors should be ranked from the most significant to least significant. You can use any methods to explain the ranking. (1 = survived, 0 = died)

When you are comfortable with your answers to the tasks above or time is running out, please move on to the final task. Again, you must work together in pairs and you may not use the internet to answer the questions.

Using test.csv and your results from the previous task: (10 min.)

Predict whether the following passengers survived:

a. Chaffee, Mrs. Herbert Fuller (Carrie Constance Toogood)

b. Robins, Mr. Alexander A

c. Peltomaki, Mr. Nikolai Johannes

d. Abelseth, Mr. Olaus Jorgensen

e. Mulvihill, Miss. Bertha E

f. Thomas, Mr. John

g. Daniels, Miss. Sarah

h. Delalic, Mr. Redjo

Post Task:

For the tasks you just completed, I'm interested in when you recommended a tool or program feature to complete the tasks. I noticed the following recommendations you made; let's look back at them briefly. *Open recommendation sheets.*

Do you recall making any other recommendations that I didn't write down, especially ones where help was not specifically asked for? *Write them down.*

Now, let's go through each one. May I turn on my audio recorder? *Turn it on. Fill in interview portion.*

A.1.2 Recommendation Sheet

The following text was used by the researchers to track instances of peer interactions during the study sessions and to guide the discussion in the semi-structured interviews at the end of the tasks:

Recommendation Sheet

Observation Data
Person making the recommendation: _____
Approximate time recommendation made: _____
Tool recommended _____

A.1.3 Interview Data

Semi-structured
Õne effective and one ineffective tool recommendation
Previous experience with tool of choice?
For person making the recommendation:

- Why did you decide to make this recommendation?
- Why did you make it this way?
- Why did you phrase it this way?
- Why did you make it at this time?
- Did the recomendee react in the way you expected?
- Anything else we should know about this?

For person who received the recommendation:

- What did you think when you got the recommendation?

- How helpful was the recommendation for the task you were doing? For future tasks?
- How helpful was the timing of the recommendation?
- How disruptive was the timing of the recommendation?
- Anything else we should know about this?

A.1.4 Interaction Data List

For each recommendation observed in our study, we collected:

- the type of peer interaction (Peer Observation or Peer Recommendation),
- the approximate time in the video the recommendation took place,
- which participants are the driver and navigator,
- the study task,
- the method of the driver and navigator (if possible),
- the name and type of the recommended feature,
- a transcript of the dialogue concerning the new tool,
- the reaction of the recommendee,
- instances in the study where the tool was re-used,
- instances where the tool was ignored for a less efficient method,
- the effectiveness, politeness, persuasiveness, and receptiveness scores,
- whether the recommendations was under time pressure, and
- if the recommendation was discussed during the interview and time of discussion in the video.

A.1.5 Peer Interaction Characteristic Scoring

This section of the appendix presents the final set of criteria that the two independent researchers agreed upon for Politeness, Persuasiveness, and Receptiveness. We specifically searched for the following when analyzing the study videos and scoring these characteristics for peer interactions.

Politeness

Tact

+1 Recommender provides beneficial reason for using tool
0 No statement on advantages or disadvantages of tool
-1 Recommender notes weakness of using suggested tool

Generosity

+1 Recommender offers to do the work for the recommendee
0 No statement on either peer doing work
-1 Recommender makes partner complete all the work

Approbation
+1 Recommender praises or compliments partner
0 No statements of praise or insults
-1 Recommender insults or offends partner

Modesty
+1 Recommender expresses humility in knowledge or abilities
0 No statements of humility or arrogance
-1 Recommender praises their own knowledge or abilities

Agreement
+1 Recommender agrees with statements made by partner or uses inclusive language
0 No statements of agreement or disagreement
-1 Recommender disagrees or argues with partner

Sympathy
+1 Recommender expresses congratulations, commiseration, or expresses condolences
0 No statements regarding sympathy or apathy
-1 Recommender incites conflict, expresses dismissiveness, or enjoys pain of partner

Persuasiveness

Content
+1 Recommender explicitly explains why the tool suits the purpose by citing a source, relating to previous experience, explaining how it works, or presenting why it's useful
-1 Recommender does not provide any information explaining why to use the suggested tool

Structure
+1 Recommender presents the tool before explaining why it should be used
-1 Recommender explains why a tool should be used before saying the tool or does not provide content

Style **+1** Recommender avoids hedging, hesitating, recommending multiple features simultaneously, asking if a tool should be used, tag questions, and passive and powerless language (i.e. "I think", "I guess", "sort of", excessive number of "Uh...", etc.)
-1 Recommender uses the statements above in their recommendation

Receptiveness

Demonstrate Desire
+1 Recommendee explicitly expresses interest or asks questions to learn more information about tool
0 No statements demonstrating desire

-1 Recommendee explicitly expresses disinterest in using tool

Familiarity

+1 Recommendee explicitly expresses familiarity with tool and environment or compares to a familiar tool

0 No statements on familiarity

-1 Recommendee explicitly states they are unfamiliar with the tool or environment

A.1.6 Demographics Questionnaire

1. Gender: _____
2. Occupation: _____
3. What is your major (or speciality of highest degree earned)? _____
4. How many years have you known your group member? _____ How would you describe your relationship with your partner? (Check all that apply)
    a. Professional ___
    b. Personal ___
    c. Academic ___
    d. None ___
5. What software do you know your partner uses regularly?
   _____
6. Have you and your partner worked together in some capacity in the past? _____
    a. If Yes, please describe any computer-based work you have done together:
       _____
7. How many years of experience do you have with the software(s) your group used to complete the tasks? _____

A.1.7 Participants

Table A.1 Peer Interaction Study Participants

Participant	Gender	Major
S1	Male	Industrial Engineering ♦
S2	Male	Computer Science ♦
S3	Male	Computer Science ♦
S4	Male	Computer Science ♦
S5	Male	Computer Science ♦
S6	Male	Computer Science ♦
S7	Male	Computer Science ♦
S8	Male	Computer Science ♦
S9	Male	Industrial Engineering ♦
S10	Female	Computer Science ♦
S11	Female	Biochemistry ♦
S12	Female	Biochemistry ◊
S13	Female	Computer Science ♦
S14	Male	Computer Science ♦

Participant	Gender	Position
L1	Female	Researcher 🤝
L2	Female	Researcher 🤝
L3	Male	Program Manager 🤝💻
L4	Female	Director of Operations 🤝💻
L5	Male	Researcher, Analyst
L6	Male	Computer Engineer
L7	Female	Researcher
L8	Male	Language Analyst
L9	Female	Engineer 🤝💻
L10	Female	Engineer 🤝💻
L11	Female	Intel Analyst, Researcher 🤝
L12	Male	Systems Researcher 🤝

♦ Graduate Student ◊ Undergraduate Student
🤝 Previous relationship with partner
💻 Previously completed computer-based work with partner

A.2 *"Sorry to Bother You: Designing Bots for Effective Recommendations"*

A.2.1 Naive Telemarketer Design

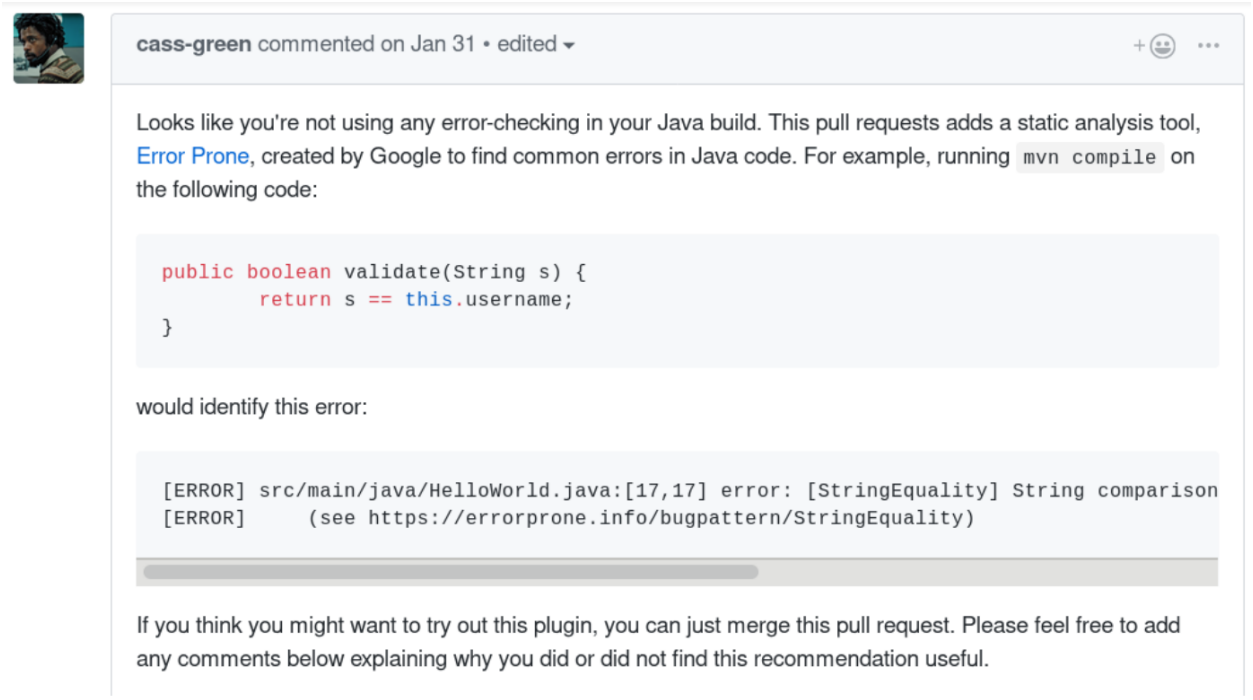


Figure A.1 Naive *Telemarketer Design* recommendation from `tool-recommender-bot`

A.2.2 Study Projects

This section outlines the projects that received recommendations for the naive *telemarketer design* study and provides the automated pull requests submitted to projects by `tool-recommender-bot`:

1. `https://github.com/jponge/lzma-java/pull/15`
2. `https://github.com/fizzed/rocker/pull/102`
3. `https://github.com/GideonLeGrange/mikrotik-java/pull/61`
4. `https://github.com/Asquera/elasticsearch-http-basic/pull/70`
5. `https://github.com/debezium/debezium/pull/760`
6. `https://github.com/dropwizard/dropwizard-elasticsearch/pull/34`
7. `https://github.com/Nodeclipse/nodeclipse-1/pull/229`
8. `https://github.com/forge/roaster/pull/101`
9. `https://github.com/recommenders/rival/pull/131recommenders/rival#131`
10. `https://github.com/tbroyer/gwt-maven-archetypes/pull/58`

11. https://github.com/Hygieia/Hygieia/pull/2696[†]
12. https://github.com/wro4j/wro4j/pull/1069[††]
13. https://github.com/jplag/jplag/pull/61
14. https://github.com/gitbucket/markedj/pull/21
15. https://github.com/jhalterman/expiringmap/pull/60
16. https://github.com/arquillian/arquillian-core/pull/190
17. https://github.com/graphstream/gs-core/pull/305
18. https://github.com/jirutka/validator-collection/pull/27
19. https://github.com/jreijn/spring-comparing-template-engines/pull/37
20. https://github.com/elkan1788/mpsdk4j/pull/8
21. https://github.com/arturmkrtchyan/iban4j/pull/59
22. https://github.com/games647/LagMonitor/pull/51
23. https://github.com/jmxtrans/jmxtrans-agent/pull/137
24. https://github.com/fakereplace/fakereplace/pull/34
25. https://github.com/google/binnavi/pull/113
26. https://github.com/devnied/AndroidBitmapTransform/pull/1
27. https://github.com/ebnew/ki4so/pull/7
28. https://github.com/bujiio/buji-pac4j/pull/83
29. https://github.com/cathive/fx-guice/pull/26
30. https://github.com/vvakame/JsonPullParser/pull/43
31. https://github.com/cderoove/damp.ekeko/pull/2
32. https://github.com/write2munish/Akka-Essentials/pull/8
33. https://github.com/Slim3/slim3/pull/30
34. https://github.com/jirutka/spring-rest-exception-handler/pull/30
35. https://github.com/yyuu/jetty-nosql-memcached/pull/35
36. https://github.com/casidiablo/persistence/pull/17
37. https://github.com/dsyer/sparklr-boot/pull/6
38. https://github.com/dgageot/simplelenium/pull/26
39. https://github.com/jhalterman/concurrentunit/pull/22
40. https://github.com/jplag/jplag/pull/62
41. https://github.com/kwart/jd-cmd/pull/19
42. https://github.com/leveluplunch/levelup-java-examples/pull/5
43. https://github.com/ngageoint/elasticgeo/pull/97
44. https://github.com/perfectsense/dari/pull/317
45. https://github.com/rchodava/datamill/pull/119
46. https://github.com/RichardWarburton/lambda-behave/pull/98
47. https://github.com/roundrop/facebook4j/pull/122

[†]Merged
[††]Merged, then reverted

48. https://github.com/spring-guides/tut-spring-boot-oauth2/pull/97
49. https://github.com/twitter/hbc/pull/195
50. https://github.com/viritin/viritin/pull/362
51. https://github.com/SINTEF-9012/JArduino/pull/82
52. https://github.com/apache/bigtop/pull/461

APPENDIX

B

STUDY MATERIALS FOR CHAPTER 5

B.1 *"Sorry to Bother You Again: Developer Recommendation Choice Architectures for Designing Effective Bots"*

B.1.1 Actionable Recommendations Survey

,

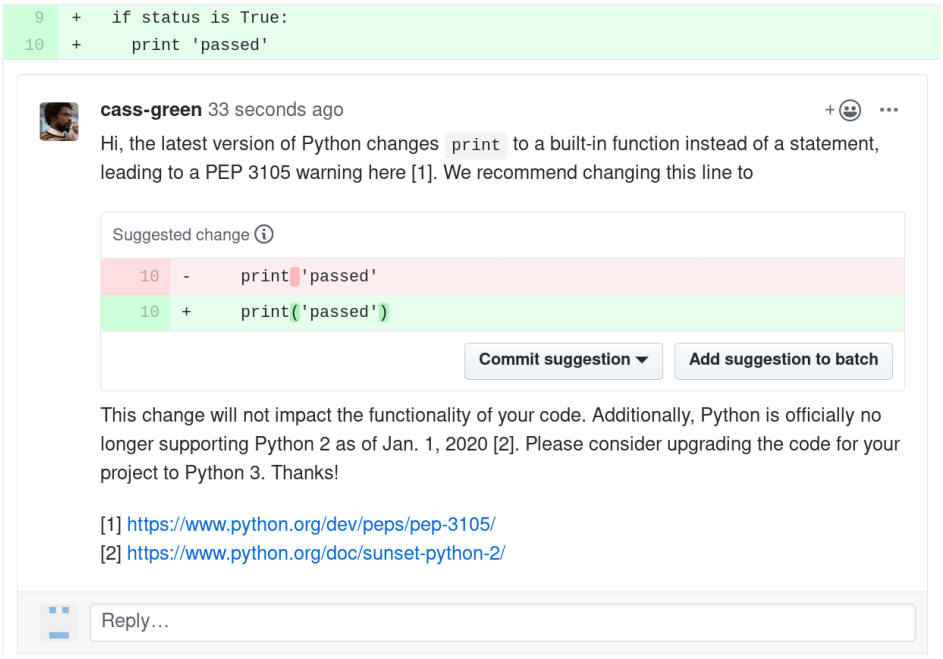
Recommendation Survey

We are researchers at NC State University seeking to better understand how to improve awareness of software engineering tools. Regarding the tool recommendation you just received, please answer the following questions honestly. By submitting this form, you consen to allow the data you submit to be used in a publicly-accessible research paper, in an aggregated and anonymized form.

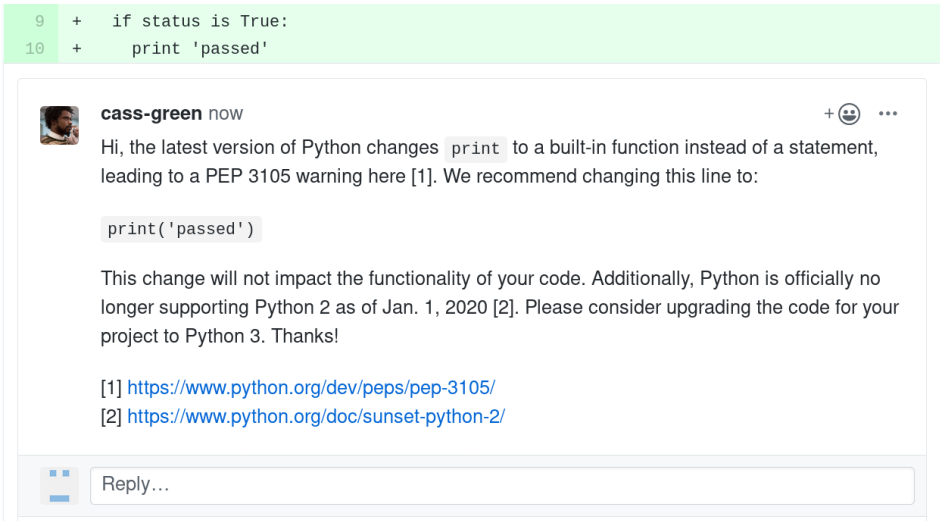
* Required

1. Email address *

Recommendation A



Recommendation B



2. Which of the above recommendations do you prefer? *

Mark only one oval.

- ◯ Recommendation A
- ◯ Recommendation B

3. Why? *

4. Years of Professional Programming Experience *

5. Please provide any other feedback or general comments on effective automated b
recommendations to software developers:

APPENDIX

C

STUDY MATERIALS FOR CHAPTER 6

C.1 *"Comparing Different Developer Behavior Recommendation Styles"*

C.1.1 Recommendation Styles

C.1.1.1 Email Recommendation

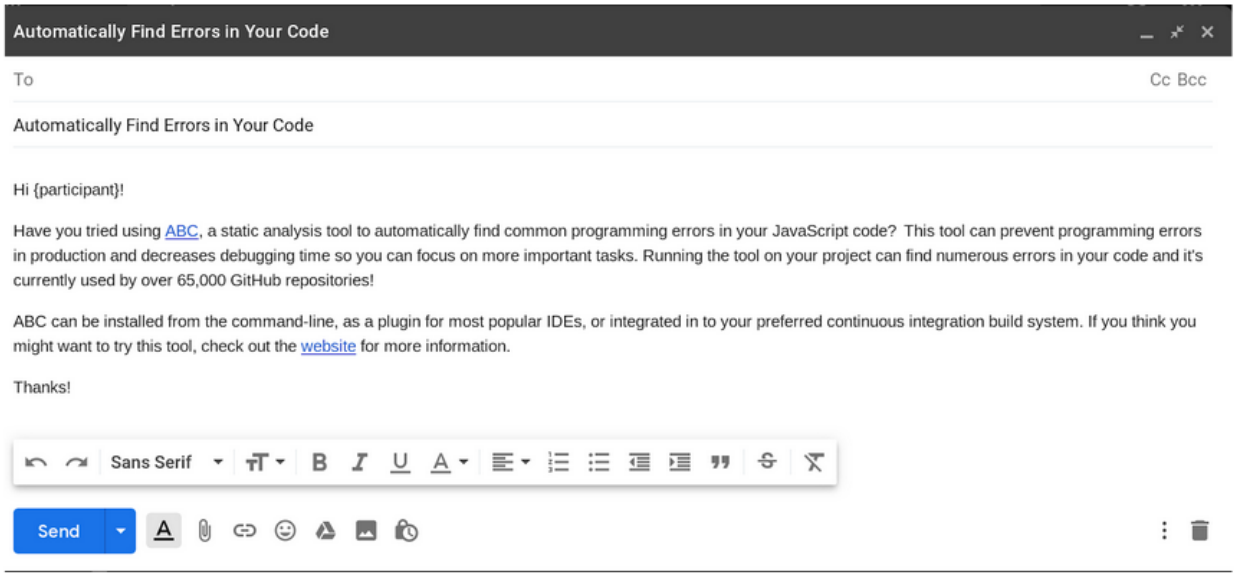


Figure C.1 Example email recommendation style

C.1.1.2 GitHub Issue Recommendation

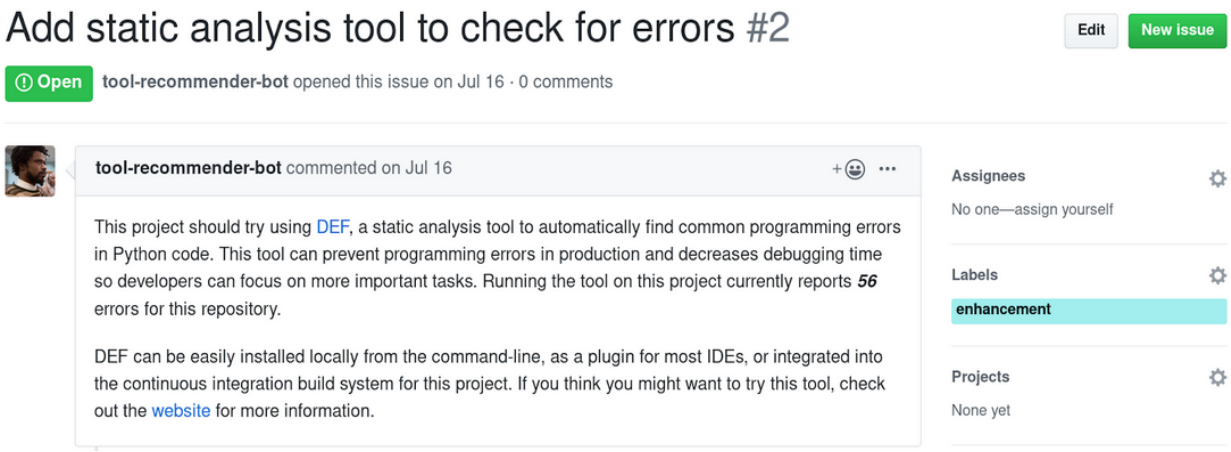


Figure C.2 Example GitHub issue recommendation style

C.1.1.3 GitHub Pull Request Recommendation

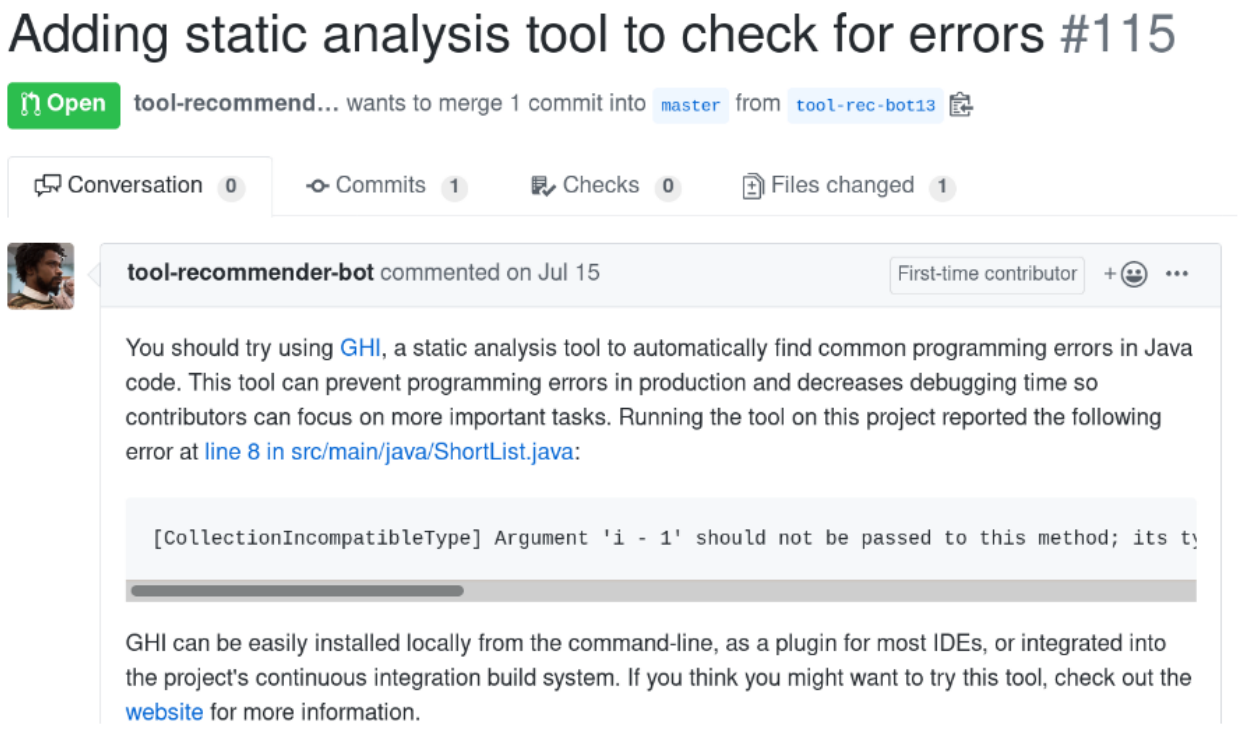


Figure C.3 Example GitHub pull request recommendation style

C.1.1.4 GitHub Suggested Change Recommendation

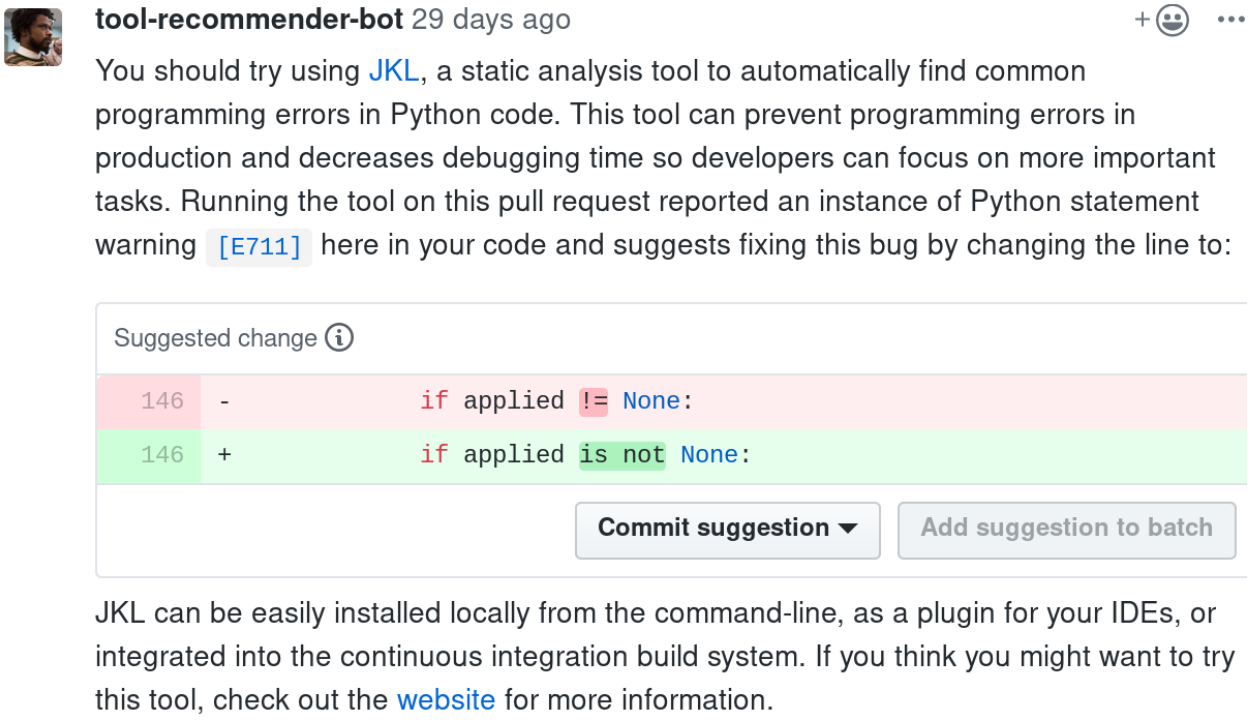


Figure C.4 Example GitHub *suggested changes* recommendation style

C.2 *"Understanding the Impact of GitHub Suggested Changes on Recommendations Between Developers"*

C.2.1 Suggested Changes Random Sample

The following instances of GitHub *suggested changes* were randomly sampled from the most recently updated pull requests on repositories and analyzed by researchers to categorize types of recommendations developers make using this feature:

1. https://github.com/4ian/GDevelop/pull/1112#discussion_r304598490
2. https://github.com/alphagov/govuk-design-system/pull/994#discussion_r307695226
3. https://github.com/angular/angular/pull/31609#discussion_r308959135
4. https://github.com/ansible/ansible/pull/60271#discussion_r312957415
5. https://github.com/apache/cordova-android/pull/764#discussion_r304199473
6. https://github.com/apache/couchdb-documentation/pull/385#discussion_r251147343
7. https://github.com/aragon/aragon-apps/pull/929#discussion_r308151240
8. https://github.com/arXiv/arxiv-search/pull/249#discussion_r310682583
9. https://github.com/aspnet/AspNetCore/pull/10406#discussion_r286161965
10. https://github.com/bbc/simorgh/pull/3048#discussion_r312518076
11. https://github.com/bitcoin/bitcoin/pull/16578#discussion_r312712522
12. https://github.com/BlueBrain/spack/pull/465#discussion_r309184444
13. https://github.com/CasperLabs/CasperLabs/pull/925#discussion_r313281322
14. https://github.com/ceph/ceph/pull/29378#discussion_r309036721
15. https://github.com/chainer/chainerrl/pull/436#discussion_r299015299
16. https://github.com/cosmos/cosmos-sdk/pull/4514#discussion_r311556341
17. https://github.com/cri-o/ocicni/pull/51#discussion_r311892555
18. https://github.com/DataDog/integrations-extras/pull/466#discussion_r305978895
19. https://github.com/dealii/dealii/pull/8384#discussion_r303584690
20. https://github.com/django/django/pull/8119#discussion_r312538500
21. https://github.com/dlang/druntime/pull/2662#discussion_r300630178
22. https://github.com/dotnet/corefx/pull/39917#discussion_r311788312
23. https://github.com/dotnet/dotnet-api-docs/pull/2968#discussion_r312193646
24. https://github.com/elastic/apm-agent-nodejs/pull/1144#discussion_r309565403
25. https://github.com/elastic/kibana/pull/41588#discussion_r311091394
26. https://github.com/ethereum/eth2.0-specs/pull/1361#discussion_r314107368
27. https://github.com/freeCodeCamp/freeCodeCamp/pull/35560#discussion_r287490144
28. https://github.com/galaxyproject/galaxy/pull/8452#discussion_r313373019
29. https://github.com/galaxyproject/tools-iuc/pull/2444#discussion_r304336637
30. https://github.com/gardener/gardener/pull/1128#discussion_r300451126
31. https://github.com/gatsbyjs/gatsby/pull/13471#discussion_r277948539
32. https://github.com/golang/protobuf/pull/785#discussion_r249270350
33. https://github.com/graphql-python/graphene/pull/992#discussion_r290376085
34. https://github.com/greenelab/text_mined_hetnet_manuscript/pull/24#discussion_r310587566
35. https://github.com/HumanCellAtlas/dcp-community/pull/92#discussion_r310355937
36. https://github.com/hyphacoop/handbook/pull/10#discussion_r303095279
37. https://github.com/hyrise/hyrise/pull/1493#discussion_r258051465
38. https://github.com/iterative/dvc/pull/2256#discussion_r302868825
39. https://github.com/jenkinsci/github-branch-source-plugin/pull/235#discussion_r303663151

40. https://github.com/jpmorganchase/quorum/pull/715#discussion_r286201830
41. https://github.com/keybase/client/pull/18045#discussion_r305071747
42. https://github.com/knative/serving/pull/5042#discussion_r310395773
43. https://github.com/Kotlin/KEEP/pull/87#discussion_r276477734
44. https://github.com/kubernetes-sigs/cluster-api/pull/1228#discussion_r311095643
45. https://github.com/kubernetes/kubernetes/pull/79641#discussion_r313118736
46. https://github.com/kubernetes/test-infra/pull/13677#discussion_r309433430
47. https://github.com/LMMS/lmms/pull/4973#discussion_r287565903
48. https://github.com/magento/magento2/pull/22156/#discussion_r272134087
49. https://github.com/mdn/browser-compat-data/pull/4558#discussion_r309696300
50. https://github.com/microsoft/qsharp-compiler/pull/52#discussion_r311190580
51. https://github.com/microsoft/QuantumKatas/pull/111#discussion_r280869695
52. https://github.com/microsoft/terminal/pull/1258#discussion_r293932790
53. https://github.com/mne-tools/mne-python/pull/6233#discussion_r284656692
54. https://github.com/moby/moby/pull/38777#discussion_r259334795
55. https://github.com/neovim/neovim/pull/10071#discussion_r309415338
56. https://github.com/nhsconnect/integration-adaptors/pull/44#discussion_r310155461
57. https://github.com/nistats/nistats/pull/352#discussion_r313598729
58. https://github.com/NixOS/nixpkgs/pull/65724#discussion_r309807069
59. https://github.com/numba/numba/pull/4204#discussion_r310598073
60. https://github.com/numpy/numpy/pull/14197#discussion_r315005011
61. https://github.com/ombulabs/blog/pull/194#discussion_r310756845
62. https://github.com/onnx/onnx/pull/2106#discussion_r313733168
63. https://github.com/open-telemetry/opentelemetry-python/pull/78#discussion_r314418832
64. https://github.com/openhab/openhab2-addons/pull/4664#discussion_r268409815
65. https://github.com/OpenRA/OpenRA/pull/15813#discussion_r298209839
66. https://github.com/operator-framework/operator-sdk/pull/1533#discussion_r293002730
67. https://github.com/ppy/osu-wiki/pull/2419#discussion_r312936003
68. https://github.com/PrismJS/prism/pull/2012#discussion_r310671629
69. https://github.com/publiclab/mapknitter/pull/306#discussion_r251212108
70. https://github.com/pypa/pip/pull/6377#discussion_r274394104
71. https://github.com/Qiskit/qiskit-terra/pull/2650#discussion_r295429688
72. https://github.com/RocketChat/Rocket.Chat/pull/12174#discussion_r302773285
73. https://github.com/rust-lang-nursery/reference/pull/635#discussion_r302287965
74. https://github.com/rust-lang/rust/pull/61708#discussion_r306056531
75. https://github.com/FluidityProject/fluidity/pull/190#discussion_r309586309
76. https://github.com/scikit-learn-contrib/scikit-learn-extra/pull/13#discussion_r307204657
77. https://github.com/scrapy/scrapy/pull/3862#discussion_r302711618
78. https://github.com/security-force-monitor/sfm-cms/pull/585#discussion_r312087662
79. https://github.com/Semmle/ql/pull/1725#discussion_r314333239
80. https://github.com/shopsys/shopsys/pull/1228#discussion_r310062868
81. https://github.com/sipa/bips/pull/52#discussion_r310589709
82. https://github.com/Skyscanner/full-stack-recruitment-test/pull/28#discussion_r312427118
83. https://github.com/sourcegraph/sourcegraph/pull/5062#discussion_r310050346
84. https://github.com/spacetelescope/synphot_refactor/pull/204#discussion_r307013162
85. https://github.com/swcarpentry/git-novice/pull/678#discussion_r311232763
86. https://github.com/syl20bnr/spacemacs/pull/12463#discussion_r305982291
87. https://github.com/sympy/sympy/pull/17266#discussion_r307727949

88. https://github.com/teamleadercrm/api/pull/361#discussion_r281559715
89. https://github.com/tensorflow/community/pull/113#discussion_r314548430
90. https://github.com/terraform-providers/terraform-provider-aws/pull/8916#discussion_r295438718
91. https://github.com/theforeman/smart-proxy/pull/657#discussion_r292927976
92. https://github.com/tombuildsstuff/golang-iis/pull/7#discussion_r314976753
93. https://github.com/vuejs/rfcs/pull/42#discussion_r296097849
94. https://github.com/weaveworks/eksctl/pull/1132#discussion_r313804187
95. https://github.com/WordPress/gutenberg/pull/16873#discussion_r310070193
96. https://github.com/XanaduAI/pennylane/pull/200#discussion_r288290349
97. https://github.com/Zarel/Pokemon-Showdown/pull/5688#discussion_r312716995
98. https://github.com/zeit/next.js/pull/7696#discussion_r302333269
99. https://github.com/zio/zio/pull/1387#discussion_r314954640
100. https://github.com/zkSNACKs/WalletWasabi/pull/2100#discussion_r313333844

C.2.2 Suggestee Survey

,

GitHub Suggested Changes

We are researchers at North Carolina State University seeking to understand and evaluate the effectiveness of the new GitHub suggested changes feature [1]. This survey is in no way affiliated with GitHub. Please answer the questions below honestly, your responses will be used for research purposes. By submitting this form, you consent to allow the data you submit to be used in a publicly accessible research paper, in an aggregated and anonymized form. For completing this survey, you will be entered in a drawing for a chance to win a $100 Amazon gift card.

[1] https://github.blog/changelog/2018-10-16-suggested-changes/

* Required

1. Email *

2. How useful is the GitHub suggested changes feature? *

Mark only one oval.

	1	2	3	4	5	
Not at All Useful	◯	◯	◯	◯	◯	Very Useful

3. What do you find useful about it?

4. What do you find not useful about it?

5. How do your teams or projects integrate this feature into the development process?

6. Have you ever learned anything new from peers through this feature? *

   Mark only one oval.

   - ◯ Yes
   - ◯ No

7. If so, what have you learned (please be specific)?

C.2.3 Suggester Survey

,

GitHub Suggested Changes

We are researchers at North Carolina State University seeking to understand and evaluate the effectiveness of the new GitHub suggested changes feature [1]. This survey is in no way affiliated with GitHub. Please answer the questions below honestly, your responses will be used for research purposes. By submitting this form, you consent to allow the data you submit to be used in a publicly-accessible research paper, in an aggregated and anonymized form. For completing this survey, you will be entered in a drawing for a chance to win a $100 Amazon gift card.

[1] https://github.blog/changelog/2018-10-16-suggested-changes/

* Required

1. Email address *

2. How useful is the GitHub suggested changes feature? *

Mark only one oval.

	1	2	3	4	5	
Not at all Useful	◯	◯	◯	◯	◯	Very Useful

3. What do you find useful about it?

4. What do you find not useful about it?

5. How does your team or project integrate this feature into the development proces

APPENDIX

D

STUDY MATERIALS FOR CHAPTER 7

D.1 *"Nudging Students Toward Better Software Engineering Behaviors"*

D.1.1 Survey

,

class-bot Survey

We are seeking to understand the use of automated feedback in the Introduction to Computing: Java course (CSC 116). This survey is intended to collect information on how you feel about your ability to engage with the course and the final assignments. Please answer the questions below honestly for Project 6 and the Comprehensive Exercise, your responses will remain anonymous and will be used to improve the course and for research purposes. Furthermore, your participation is voluntary and will not impact your performance in the class.

* Required

1. Email address *

2. How useful were the class-bot automated updates? *

Mark only one oval.

	1	2	3	4	5	
Not at All Useful	◯	◯	◯	◯	◯	Very Useful

3. Please explain:

4. How often did you check the automated software process updates from class-bot on your GitHub repository? *

Mark only one oval.

	1	2	3	4	5	
Rarely	◯	◯	◯	◯	◯	Very Frequently

5. Please explain:

6. For Projects 1-5, how often did you check the software process requirements on th website for the assignment? *

Mark only one oval.

	1	2	3	4	5	
Rarely	◯	◯	◯	◯	◯	Very Frequently

7. Please explain:

8. Please provide any additional comments on the course, improving class-bot, or ho feedback to students on GitHub can be generally be improved for CSC116 in the future.