Sorry to Bother You: Designing Bots for Effective Recommendations

Chris Brown, Chris Parnin
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
dcbrow10@ncsu.edu, cjparnin@ncsu.edu

***Abstract*—Bots have been proposed as a way to encourage developer actions and support software development activities. Many bots make recommendations to users, however humans may find these recommendations ineffective or problematic. In this paper, we argue that while bots can help automate many tasks, ultimately bots still need to find ways to interact with humans and handle all of the associated social and cognitive problems entailed. To illustrate this problem, we performed a small study where we generated 52 pull requests making tool recommendation to developers. As a result, we only convinced *two* developers to accept the pull request, while receiving several forms of feedback on why the pull request was ineffective. We summarize this feedback and suggest design principles for bot recommendations, including how psychology frameworks, such as *nudge theory*, can be used to improve human-bot interactions.**

***Index Terms*—software engineering, developer actions, tool adoption, digital nudge**

I. Introduction

Software robots, or *bots*, are useful for automating a variety of tasks and can improve efficiency and effectiveness [12]. However, their interactions with humans can be inconvenient due to limited responses from rule-based reasoning [11] and poor UX design.[1] For example, software developers found bots relevant and useful for automatically running tests, but they also reported facing numerous challenges, such as poor decision support and non-comprehensive feedback [16]. Poor interactions between humans and bots can lead to negative responses toward automated suggestions generated by bots. For instance, users expressed more inappropriate language and negative emotions in conversations with Cleverbot[2] compared to humans [5].

To understand the impact of poorly behaving bots on humans, we create a naive *telemarketer design*, where a bot delivers actionable and technically sound recommendations to software developers; however all interactions are *static*. In many ways, our design is like a telemarketer: calling users to deliver a static message. However, the interactions never deviate from the script and lack the social intelligence required to adjust the message or respond to questions. With this simple design, our goal is to initiate and then identify reactions from software developers. Future bots could then be designed to account for these interactions with users.

To contextualize our design, we developed a bot, *tool-recommender-bot*, which makes tool recommendations to software developers by automatically configuring a software project to use the tool. Using development tools, such as static analysis tools to automatically check for programming errors in source code, is a useful behavior for developers to adopt for improving the quality of their work. For example, software engineers at Google used the static analysis tool FindBugs[3] to find thousands of warnings in their code [4]. Despite the benefits of using static analysis tools, software engineers do not use them often in practice [7]. In this study, we seek to explore designing bots as recommender systems to improve the tool adoption problem in software engineering.

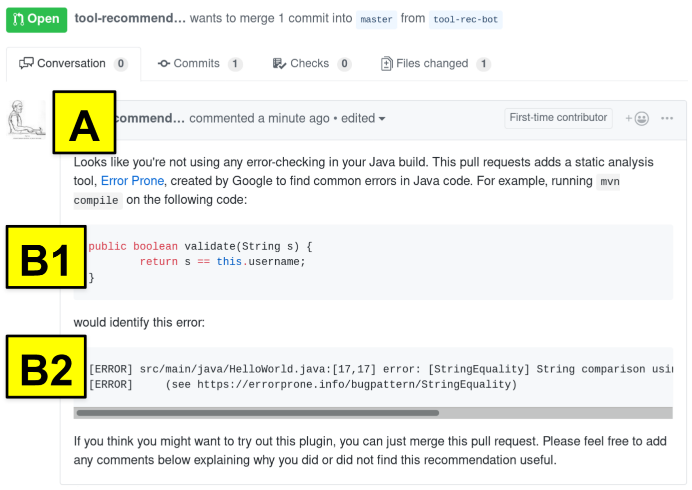
We evaluate pull requests made by *tool-recommender-bot* and categorize the responses from software developers. Our results suggest that bots with simple technical knowledge alone are ineffective in influencing human behavior. *tool-recommender-bot* was only able to make two successful recommendations out of 52 (4%). Analyzing these responses, we found that *tool-recommender-bot* lacked the social context required to interact with users and understand the contextual factors to make effortless integration into a project's development workflow. To remedy this, we propose integrating concepts from psychology and persuasion theory in bots to improve human-bot interactions. For example, *nudge theory* is a behavioral science concept that examines simple ways to influence human decision-making and convince people to adopt certain behaviors [13]. Long-term, we hope this research provides a baseline that future designs can surpass.

II. Motivating Example

Cassius is a software developer maintaining several popular open source Java projects on GitHub. He does not implement any static analysis in his repositories because he is unaware of tools that can analyze Java code to prevent bugs. One day, he notices a new pull request on his repositories, similar to the one shown in Figure 1.

The pull request introduces Cassius to Error Prone, an open source static analysis tool for Java code [1]. Reading

[1]https://chatbot.fail/
[2]https://www.cleverbot.com/
[3]http://findbugs.sourceforge.net/



tool-recommend... wants to merge 1 commit into `master` from `tool-rec-bot`

commend... commented a minute ago • edited

Looks like you're not using any error-checking in your Java build. This pull requests adds a static analysis tool, Error Prone, created by Google to find common errors in Java code. For example, running `mvn compile` on the following code:

```
public boolean validate(String s) {
        return s == this.username;
}
```

would identify this error:

```
[ERROR] src/main/java/HelloWorld.java:[17,17] error: [StringEquality] String comparison usi
[ERROR]     (see https://errorprone.info/bugpattern/StringEquality)
```

If you think you might want to try out this plugin, you can just merge this pull request. Please feel free to add any comments below explaining why you did or did not find this recommendation useful.

Fig. 1. Example recommendation from *tool-recommender-bot*

the pull request comment (Fig. 1.A), he learns what the tool does and how to use it. He also views a simple Java function (Fig. 1.B1) with an example of an error reported by ERROR PRONE (Fig.1.B2). Then, he realizes not only does the pull request present information about the tool, but it also automatically adds the tool to by modifying the build configuration files! He just needs to simply merge the pull requests to add the static analysis tool to his repositories. Cassius merges the requests to successfully integrate ERROR PRONE in his projects, preventing himself and other contributors from adding programming errors in the future.

This scenario is an example of how bots can be useful for helping users adopt new behaviors. In this case, using a bot effectively automated tasks, such as updating the build configuration file, and was able to make recommendations on multiple projects more efficiently than manual suggestions.

III. TOOL-RECOMMENDER-BOT

tool-recommender-bot is a bot designed to recommend software engineering tools to developers on GitHub. Our system suggests tools to GitHub users by automatically modifying build configuration files and generating pull requests with the tool. Many companies use build systems to automatically integrate, compile, test, and deploy their software more efficiently [10]. While this is not the only context in which software engineering tools can be used, it allows developers to easily integrate and run new tools within their normal workflow. *tool-recommender-bot* generates automated pull requests because they are the preferred method for suggesting changes to repositories,[4], have been used by bots in prior work [8], and create actionable recommendations for users.

Our goal is for *tool-recommender-bot* to be extendable to recommend a wide variety of software engineering tools to developers. The initial implementation for this study naively recommends ERROR PRONE to Java developers on GitHub by adding it to repositories that use Maven, a popular build automation and dependency management tool for Java applications [2]. *tool-recommender-bot* automatically adds the ERROR PRONE Maven plugin to the Project Object Model (*pom.xml*) configuration file to run the tool when the code compiles. The source code for our bot is publicly available online.[5]

Our naive *telemarketer design* for this study provides a generic message and a simple code snippet with a common Java error to show users sample output from ERROR PRONE, shown in Figure 1. This is not the best design for making recommendations, however we implemented this naive approach to create a baseline describing how bots influence human behavior. *tool-recommender-bot* uses a human-presenting account to make recommendations based on research showing bots emulating human users are more effective [9]. In our pilot, we quickly discovered bot accounts are ineffective after our original GitHub account[6] for *tool-recommender-bot* was flagged within a few hours of making recommendations.

IV. METHODOLOGY

A. *Projects*

To evaluate *tool-recommender-bot* we used open source software repositories on GitHub. We sampled projects from the Repairnator evaluation[7] to use for our study [14]. The projects we selected met the following criteria:

- primarily written in Java 8+,
- successfully compile with Maven, and
- do not already use ERROR PRONE

Since ERROR PRONE can only analyze Java code, our evaluation was limited to projects written in the Java programming language. To collect projects that build with Maven, we checked to see if repositories contained a *pom.xml* file in the home directory and confirmed they could be validated and compiled before adding the plugin. We also verified the pom.xml file did not already contain the ERROR PRONE plugin to avoid making recommendations to projects that already use the tool and target developers less likely to know about it.

Our evaluation consisted of real-world Java applications with a varying functionalities, number of source code files, and lines of code. While our evaluation focuses on open source projects, recommendations from *tool-recommender-bot* could receive similar reactions from developers of proprietary software in a large company setting. The list of projects used for our evaluation is available online.[8]

B. *Data Analysis*

We categorized tool recommendations from our bot as *effective* or *ineffective* based on the status of pull requests. Developers of projects that receive a recommendation have the option to merge the pull request into the source code, close the pull request without merging, or ignore the pull request

[4]https://help.github.com/articles/about-pull-requests/

[5]https://github.com/chbrown13/tool-recommender-bot/tree/pulls

[6]https://github.com/tool-recommender-bot

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

[8]https://go.ncsu.edu/botse-projects

by leaving it open. Merging the automated pull request from our system indicates an effective recommendation, since the recommendee decided to adopt ERROR PRONE and showed a willingness to integrate the static analysis tool into the project's build configuration to check for errors. Closing or ignoring the pull request from *tool-recommender-bot* implies an ineffective recommendation because recommendees did not attempt to integrate the recommended tool.

We gathered quantitative and qualitative data to examine the effectiveness of *tool-recommender-bot* over the course of a week [19]. We observed merged pull requests from *tool-recommender-bot* to calculate the bot's rate of effectiveness for tool recommendations. Additionally, we aggregated comments from GitHub users on pull requests to analyze how developers reacted to receiving a recommendation. In the pull request, we encouraged developers to provide feedback on whether they found the recommendation useful or not. Additionally, GitHub also allows users to provide feedback on pull requests and comments using emoji reactions.[9] We analyzed responses and reactions to gather insight into how developers felt about receiving automated recommendations from a bot.

V. RESULTS

A. Bot Effectiveness

Out of 52 recommendations, only *two* were accepted by developers. The remaining were categorized as ineffective recommendations: 10 closed and 40 never received any response. Table I presents the results of our preliminary evaluation.

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

TABLE I: Pull Request Results

An overwhelming 96% of *tool-recommender-bot* recommendations were ineffective. Of the 12 recommendations that did receive developer feedback, 83% were rejected by developers who closed the pull request. Two recommendations were merged, however in one case another contributor created a GitHub issue because our pull request caused problems with the project build. The changes made by the bot were then reverted in a later pull request, removing ERROR PRONE from the project. Even though the tool was eventually removed, we still categorize this as an effective recommendation because the developers accepted the pull request to try the tool.

B. Developer Reactions

Our qualitative data also shows developers did not find automated recommendations from our bot effective. We found 24 comments made on 17 unique projects. Of the 24 total responses, six were automated comments made on pull requests that provided information to first-time contributors, requested Contributing License Agreement signatures, and presented code coverage updates. We received 18 developer responses from 15 different GitHub users and no emoji reactions on automated pull requests from *tool-recommender-bot*. There were five positive statements on recommendations, including on pull requests that were not merged:

> "lgtm, Good Contribution" (P9, merged)
>
> "Thanks for sharing it" (P13, merged)
>
> "Thanks for helping spread this wonderful tool that is Error Prone" (P14)
>
> "ErrorProne itself is something I could see us adding." (P7)

However, most of the feedback we received from developers was negative. We categorized the remaining comments into two main cardinal sins we committed in the design of our bot, which led to ineffective tool recommendations.

1) Cardinal Sins:

a) Formatting: One issue developers found with our automated recommendations was inconsistent file formatting. In some cases, the format of the modifications to the *pom.xml* file in our automated pull requests was different from rest of the file in terms of whitespace, such as indentation, tab length, and line breaks. Several developers noted this, providing responses such as:

> "You messed up the formatting of the pom.xml pretty bad." (P3)
>
> "The automated tool you use messed up the pom.xml formatting to an extent that I could not see it." (P5)
>
> "This change removes quite a lot if important things from the POM file." (P7)

There were five comments mentioning the *pom.xml* file formatting. Despite P7's comment, we did not remove anything from the configuration file and only added the plugin to it. For our evaluation, we attempted to use a general format when updating the build configuration file for repositories but did not exactly follow the format for each individual project, which prevented some developers from merging pull requests.

b) Breaking builds: Another major problem participants faced was that adding ERROR PRONE often broke project builds in Maven. Ironically, this was primarily due to errors reported by the tool itself in the projects' source code. Many comments were made regarding this, including the following:

> "Given the number of errors, I think it would cause more harm than good ;)" (P11)
>
> "Introduced erroneous behavior to the build." (P10)
>
> "Your PR breaks the CI." (P1)

Eight comments mentioned failing builds on recommendations from *tool-recommender-bot*. While we hoped integrating tools into build configuration would make them easier to adopt, we found that developers did not appreciate it when our changes broke the build with errors. This was a major reason users closed automated pull requests from *tool-recommender-bot*, however not generating pull requests updating *pom.xml* files could lead to increased adoption friction.[10]

[9]https://developer.github.com/v3/reactions/

[10]https://www.awh.net/resources/blog/reduce-product-friction-to-increase-user-adoption/

VI. Discussion

Our results suggest that bots alone are not effective for influencing developer behavior. The majority of tool recommendations from our naive telemarketer bot design were ineffective, ignored or rejected by developers. This points to a need for design changes and improvements in order for bots to make better recommendations. Based on our experience implementing and evaluating *tool-recommender-bot* to impact human behavior, we propose design principles for researchers to consider when developing future bots.

A. Social Context

One disadvantage of the design of *tool-recommender-bot* is that it lacks social context. We refer to social context as the practices and activities standard for interacting with developers and contributing to open source software. Examples include formatting guidelines, Contributor License Agreements, README[11] and CONTRIBUTING[12] files, identifying unmaintained projects, participating in conversations during code reviews, and more. Furthermore, Wessel et al. found developers faced challenges interacting with bots and desired improved social interactions and smarter bots when making contributions to open source software [16].

One method to accomplish this is to provide better examples within the project context. P7 desired more details on how ERROR PRONE could "actually help us" and suggested we "could attach a report with actual findings in our code base instead of just some generic example." Using relevant examples can contextualize tools and improve recommendations on projects. Another solution is integrating chatbots into automated recommendations to answer questions from developers about the proposed tool. For example, AnswerBot was able to successfully automate answering Java questions from developers on the Q&A site Stack Overflow [18].

B. Developer Workflow

Many users found *tool-recommender-bot* interrupted existing processes, most notably by breaking builds, and were discouraged from merging pull requests from our system. While social context deals with user interactions, developer workflow refers to interactions with the code and software development practices. Several developers expressed additional information needs that impact their workflow: P3 desired to know if bugs were "worth fixing" and how to "configure the plugin so as to ignore false positives"; P7 noted "it'd be good to analyse the impact in terms of build time"; and P17 asked "Can you fix the errors reported by your tool in the build?". Additionally, Johnson and colleagues found that tools' inability to integrate into developer workflow is a key barrier for adoption [7].

To improve recommendations, bots should suggest tools without breaking existing infrastructure. One solution is to change how potential users interact with tools. For instance, rather than fully integrating tools with automated pull requests, allow developers to incrementally use them, i.e., modifying the ERROR PRONE plugin to report errors as warnings instead of compilation errors or implementing a demo-mode for users to try the tool in a sandbox environment without impacting builds. Another solution is to help developers further by fixing issues introduced in the build. For instance, the program repair bot Repairnator presents developers with patches for errors [14]. Automatically fixing reported bugs can improve recommendations within developer workflows.

C. Nudge Theory

Psychology research studies many ways to impact human behavior. One example of this is *nudge theory*, which examines any factor that affects how people make decisions without incentives or constraints [13]. Nudges influence common everyday decisions such as encouraging people to recycle more [3], and much larger choices like improving civic behavior and reducing crime [6]. Using technology to send nudges in digital environments is referred to as *digital nudging* [15]. For example, FitBit[13] digital watches encourage users to increase physical activity [15]. We envision, when given a developer who is unaware of a useful tool, using bots to digitally nudge developers to adopt the tool. To accomplish this, we propose researchers integrate concepts from nudge theory to improve how bots mesh with social context and development workflows when making recommendations.

Nudge theorists Thaler and Sunstein note nudges "must be easy and cheap" [13, p. 6]. However, we found the naive *telemarketer design* was cumbersome and expensive for developers by avoiding interactions and complicating project builds. Designing bots to make easy and cheap recommendations can improve their effectiveness by showing the value of tools simply and clearly. For example, with *tool-recommender-bot* we found modifying build configuration files was costly, creating additional build errors needing to be fixed. An example nudge would be notifying developers about bugs reported by tools without impacting the build. To do this, our design needs to modify how recommendations are presented. Nudge theory suggests that location impacts decisions (i.e., placing healthier foods at eye level encourages people to eat better [17]). Instead of automated pull requests, making comments on lines of code with errors may simplify recommendations to users. Analyzing more nudge theory examples and applying them to bot design can improve human-bot interactions for influencing behavior.

VII. Conclusion

We implemented *tool-recommender-bot*, a bot that recommends static analysis tools to developers, using a naive *telemarketer design* approach. We evaluated our system by suggesting ERROR PRONE to GitHub users with automated pull requests. Our results show this approach is insufficient for making recommendations, ignoring social context and integrating poorly with development workflows. To make more effective recommendations, we propose using nudge theory to improve bot designs for impacting human behavior.

[11]https://help.github.com/articles/about-readmes/

[12]https://help.github.com/articles/setting-guidelines-for-repository-contributors/

[13]https://www.fitbit.com/home

References

[1] Error prone. http://errorprone.info.

[2] Maven. https://maven.apache.org/.

[3] A strategy for recycling: Change the recycling-bin-to-garbage-bin ratio. Nudge Blog. http://nudges.org/2011/05/02/a-strategy-for-recycling-change-the-recyling-bin-to-garbage-bin-ratio/.

[4] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[5] J. Hill, W. R. Ford, and I. G. Farreras. Real conversations with artificial intelligence: A comparison between human–human online conversations and human–chatbot conversations. *Computers in Human Behavior*, 49:245–250, 2015.

[6] P. John, G. Smith, and G. Stoker. Nudge nudge, think think: two strategies for changing civic behaviour. *The Political Quarterly*, 80(3):361–370, 2009.

[7] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

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

[9] A. Murgia, D. Janssens, S. Demeyer, and B. Vasilescu. Among the machines: Human-bot interaction on social q&a websites. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1272–1279. ACM, 2016.

[10] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference (AGILE), 2015*, pages 1–10. IEEE, 2015.

[11] C. Stanfill and D. Waltz. Toward memory-based reasoning. *Commun. ACM*, 29(12):1213–1228, Dec. 1986.

[12] M.-A. Storey and A. Zagalsky. Disrupting developer productivity one bot at a time. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 928–931. ACM, 2016.

[13] C. Sunstein, R. Thaler, et al. Nudge. *The politics of libertarian paternalism. New Haven*, 2008.

[14] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. How to design a program repair bot?: insights from the repairnator project. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 95–104. ACM, 2018.

[15] M. Weinmann, C. Schneider, and J. vom Brocke. Digital nudging. *Business & Information Systems Engineering*, 58(6):433–436, 2016.

[16] M. Wessel, B. M. de Souza, I. Steinmacher, I. S. Wiese, I. Polato, A. P. Chaves, and M. A. Gerosa. The power of bots: Characterizing and understanding bots in oss projects. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW):182:1–182:19, Nov. 2018.

[17] A. L. Wilson, E. Buckley, J. D. Buckley, and S. Bogomolova. Nudging healthier food and beverage choices through salience and priming. evidence from a systematic review. *Food Quality and Preference*, 51:47–64, 2016.

[18] B. Xu, Z. Xing, X. Xia, and D. Lo. Answerbot: automated generation of answer summary to developers´ technical questions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 706–716. IEEE Press, 2017.

[19] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *Mining software repositories (MSR), 2015 IEEE/ACM 12th working conference on*, pages 367–371. IEEE, 2015.