Context-Aware Conversational Developer Assistants

Nick C. Bradley
Department of Computer Science
University of British Columbia
Vancouver, Canada
ncbrad@cs.ubc.ca

Thomas Fritz
Department of Informatics
University of Zurich
Zurich, Switzerland
fritz@ifi.uzh.ch

Reid Holmes
Department of Computer Science
University of British Columbia
Vancouver, Canada
rtholmes@cs.ubc.ca

ABSTRACT

Building and maintaining modern software systems requires developers to perform a variety of tasks that span various tools and information sources. The crosscutting nature of these development tasks requires developers to maintain complex mental models and forces them (a) to manually split their high-level tasks into low-level commands that are supported by the various tools, and (b) to (re)establish their current context in each tool. In this paper we present Devy, a Conversational Developer Assistant (CDA) that enables developers to focus on their high-level development tasks. Devy reduces the number of manual, often complex, low-level commands that developers need to perform, freeing them to focus on their high-level tasks. Specifically, Devy infers high-level intent from developer's voice commands and combines this with an automatically-generated context model to determine appropriate workflows for invoking low-level tool actions; where needed, Devy can also prompt the developer for additional information. Through a mixed methods evaluation with 21 industrial developers, we found that Devy provided an intuitive interface that was able to support many development tasks while helping developers stay focused within their development environment. While industrial developers were largely supportive of the automation Devy enabled, they also provided insights into several other tasks and workflows CDAs could support to enable them to better focus on the important parts of their development tasks.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

Conversational Development Assistants, Natural User Interfaces

ACM Reference Format:
Nick C. Bradley, Thomas Fritz, and Reid Holmes. 2018. Context-Aware Conversational Developer Assistants. In *Proceedings of 40th International Conference on Software Engineering, Gothenburg, Sweden, May 2018 (ICSE'18),* 11 pages.
https://doi.org/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE'18, May 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN . . . $15.00
https://doi.org/

1 INTRODUCTION

Software development is hard. Empirical studies have shown that developers have to perform a broad variety of development tasks, frequently switching applications and contexts [5, 13, 14]. To successfully complete these higher level tasks, developers must perform a series of low-level actions—workflows—and maintain mental models that combine data from a variety of disparate sources including the source code of the system, version control, issue trackers, test executions, and discussion threads [8, 9, 11].

While it would benefit developers to automate these workflows, it is challenging for three reasons. First, it is hard to determine a priori all the tasks and workflows developers will need to complete. Second, these workflows consist of various low-level actions that often span across tool boundaries and require a diverse set of parameters that depend on the current context and developer intent. Third, even if there are scripts configured for automating workflows, the developer needs to remember their existence and how to invoke them manually in the current context.

In this paper, we explore the potential of conversational agents to support and automate common development workflows. We designed a conversational developer assistant (CDA) that (a) provides a *conversational interface* for developers to specify their high-level tasks in natural language, (b) uses an *intent service* to automatically map high-level tasks to low-level development actions, and (c) automatically tracks developers' actions and relevant state in a *context model* to automate the workflows and specification of parameters. The CDA allows developers to express their intent conversationally, eliminating the need for learning and remembering rigid syntax, while promoting discoverability and task flexibility. The automatic mapping and execution of workflows based on the developer's high-level intent, augmented by the context model, reduces developers' cognitive effort of breaking down high-level intents into low-level actions, switching context between disparate tools and parameterizing complex workflows.

In order to conduct a meaningful industrial evaluation of the feasibility, usability, and potential use cases of CDAs in software development, we implemented Devy, a prototype voice-controlled CDA with a pre-defined set of automated Git and GitHub tasks. Devy's primary goal is to help developers maintain their focus on their development tasks, enabling them to offload low-level actions to an automated assistant.

We performed a mixed methods study—a combination of an interview and an experiment—with 21 industrial software engineers using Devy as a technology probe. Participants had the opportunity to interact with our Devy prototype so they could offer concrete feedback about alternative applications of CDAs to their industrial workflows. Each engineer performed multiple experimental tasks

Table 1: Steps for the common 'share changes' workflow.

(a) Manual steps.	(b) Literal CDA steps.
(a) Open a web browser for the issue tracker and check the issue number for the current work item.	"CDA, Create a branch 'issue1223' in the FrontEnd repo." → "Branch created."
(b) Open a terminal and run the tests against the changed code to ensure they work (e.g., `npm run tests`).	"CDA, Run all the tests in the FrontEnd repo." → "Tests executing."
(c) Open a terminal and commit the code, tagging it with the current work item number (e.g., `git commit -m 'See issue #1223'`).	"CDA, Commit with 'Fix #1223' in the FrontEnd repo." → "Commit made."
(d) Pull any external changes from the remote repository (e.g., `git pull`).	"CDA, Pull the FrontEnd repo." → "Pulled."
(e) Push the local change to the remote repository (e.g., `git push`).	"CDA, Push the FrontEnd repo." → "Pushed."
(f) Open the commit in the version control system using the GitHub web interface and open a pull request.	"CDA, Open GitHub for the FrontEnd repo and create a pull request for branch issue1223." → "Pull request created."
(g) Determine a set of reviewers and assign them to the pull request with the GitHub web interface.	"CDA, Open GitHub for the FrontEnd repo and add alice79 as a reviewer for the issue1223 pull request." → "Reviewer added."

with Devy and answered a series of open-ended questions. The evaluation showed that engineers were able to successfully use Devy's intent-based voice interface and that they saw promise in this type of approach in practice.

This feedback provides evidence of the potential and broad applicability of both Conversational Developer Assistants and developer's interest in increased automation of their day-to-day workflows.

The primary contributions of this paper are:

- A context model and conversational agent to support automated development assistants.
- Devy, a prototypical voice-activated CDA that infers developer intent and transforms it into complex workflows.
- A mixed methods study demonstrating the value of this approach to industrial developers and providing insight into how CDAs can be used and extended in the future.

We describe a concrete scenario in Section 2, our approach in Section 3 and our experiment in Sections 4 and 5. Related work, discussion, and conclusions follow in Sections 6–8.

2 SCENARIO

Development projects often use complex processes that involve integrating numerous tools and services. To perform their high-level tasks, developers need to break down their intent into a list of atomic actions that are performed as part of a workflow. While the intent may be compact and simple, workflows often involve interacting with a variety of different tools.

Consider a developer whose task is to submit their source code changes for review, which requires using version control, issue tracking, and code review tools. At a low level, the developer needs to: commit their changes, push them to a remote repository, link the change to the commit in the issue tracker, and assign reviewers in the code review system. Our context model is able to track what project the developer is working on, what issue is currently active, and who common reviewers for the changed code are in order to enable the developer to just say *"Devy: I'm done"* to complete this full workflow without having to change context between different tools.

To perform this task manually, the developer must follow a workflow similar to that shown in Table 1a (for simplicity, we illustrate this workflow using GitHub). In this scenario, developers use three tools: GitHub (Table 1a-(a),(f),(g)), the test runner (Table 1a-(b)), and git (Table 1a-(c),(d),(e)). They also performed four overlapping subtasks: *running* the tests (Table 1a-(b)), *linking* the commit (Table 1a-(a),(c)), *managing* version control (Table 1a-(c),(d),(e)), and *configuring* the code for review (Table 1a-(f),(g)). In addition, they relied on several pieces of implicit contextual knowledge: (1) the repository being used, (2) the current issue, (3) how to run the tests, (4) the project's commit linking protocol, and (5) the project's code review assignment protocol.

Providing a voice-activated CDA for this workflow without any additional abstraction (or context model) offers little benefit as shown by the transcript in Table 1b which has been aligned with the manual steps in Table 1a. Grey rows are the developer's speech, white rows are the CDA's responses. This implementation has obvious shortcomings: it provides no meaningful benefit over just accessing the commands directly, as the developer must say all of the commands with the right names, in the right order, with the right parameters, and it would no doubt be faster if they performed the actions directly.

Automating this workflow would require knowing the five pieces of the contextual information along with knowledge of how to use the three tools employed by the developer. Fortunately, these are all pieces of information that are tracked directly by the context model of our conversational developer assistant Devy. The same workflow can be completed using Devy's verbal natural language interface:

Dev Devy, I'm done.
Devy You have uncommitted changes. Should I commit them?
Dev OK.
Devy OK, I'm about to open a pull request, should I assign Alice?
Dev Yeah.

During this interaction, the developer did not need to use any other tools or switch their context away from their source code. The context model automatically tracked the project being worked on, the files being changed, and the current issue number. To show the results of the tests (Table 1a-(b)), Devy appends the output of the test execution as a comment to the pull request thread when it is complete. To identify the list of appropriate reviewers (Table 1a-(g)), Devy is able to query a simple service that examines past reviewers for the code involved in the developer's change.

While the developer's intent, submitting changes, is simple, it can only be realized through the indirect process listed above that involves interacting directly with the version control system, issue tracker, test execution environment, and code review system. Each of these systems incurs their own mental and temporal costs, and provides opportunities for a team's workflow to be ignored (e.g., if new team members are not aware of all steps, or an experienced one skips a step). Ultimately, this task involves four context switches between the issue tracker, the version control system, the pull request interface, and the code review system; Devy abstracts away this minutiae so the developer can focus on their high level intent.

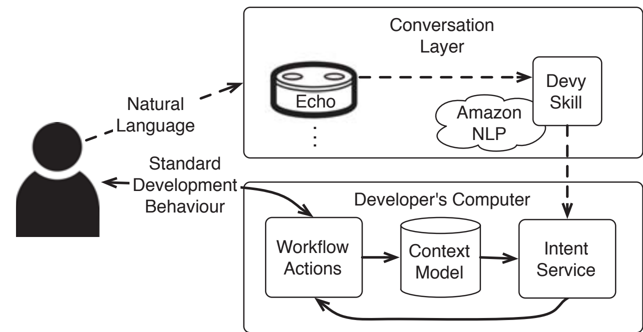

Figure 1: Devy's architecture. A developer expresses their intention in natural language via the conversational layer. The intent service translates high-level language tokens into low-level concrete workflows which can then be automatically executed for the developer. Dotted edges predominantly communicate in the direction of the arrow, but can have back edges in case clarification is needed from the user.

3 APPROACH

Devy, our conversational developer assistant (CDA), has three main components: a conversational user interface, a context model, and an intent service. Developers express their intent using natural language. Our current prototype uses Amazon Echo devices and the Amazon Alexa platform to provide the conversational interface; this interface converts developer sentences into short commands. These commands are passed to our intent service which runs on the developer's computer. The context model actively and seamlessly updates in the background on the developer's computer to gather information about their activities. Using the context model and the short commands from the conversational interface, the intent service infers a rich representation of the developer's intent. This intent is then converted to a series of workflow actions that can be performed for the developer. While the vast majority of input to the intent service is derived from the context model, in some instances clarification is sought (via the conversational interface) from the developer. Devy's architecture is shown in Figure 1.

3.1 Conversational Interface (Devy Skill)

The conversional interface plays a crucial role by allowing developers to express their intents naturally without leaving their development context. The Devy Skill has been implemented for the Amazon Alexa platform (apps on this platform are called skills). To invoke the Devy Skill, a developer must say:

"Alexa, ask Devy to ..."

They can then complete their phrase with any intent they wish to express to Devy. The Amazon microphones will only start recording once they hear the 'Alexa' word, and the Devy skill will only be invoked once 'Devy' has been spoken.

The Amazon natural language APIs translate the developer's conversation into a JSON object; to do this, the Devy skill tells the Amazon Alexa platform what kinds of tokens we are interested in. We have provided the platform with a variety of common version control and related development 'utterances' we identified in the literature and from discussions with professional developers; many utterances also have synonyms (e.g., for 'push', we also include 'submit', and 'send'). For a sentence like "Alexa tell Devy to push." the Amazon JSON object would contain one primary field `intent.name` with the value `'pushChanges'`.

While Alexa has been useful for constructing our prototype, it imposes two restrictions that hinder our approach:

(1) The requirement to use two names, "Alexa" and "Devy" is cumbersome.
(2) More technically, Alexa doesn't allow push notifications and requires the client app to respond within ten seconds; both of which cause issues for long running commands.

While our current approach uses the Amazon APIs for voice input, using a text-based method (e.g., a ChatBot) would also be feasible for scenarios where voice-based input is not appropriate.

3.2 Context Model

The context-aware development model represents the 'secret sauce' that enables advanced voice interactions with minimal explicit developer input. The differences between manual CDA and context-aware CDA (Devy) approaches are exemplified in Section 2. The model acts as a knowledge base allowing the majority of the parameters required for performing low-level actions to be automatically inferred without developer intervention. In cases where required information is not present in the context model, it can be prompted from the developer using the conversational interface.

The context model for our prototype system is described in Table 2. The current model supports version control actions and online code hosting actions. Our prototype tool includes concrete bindings for Git and GitHub respectively for these two roles but also supports other semantically similar systems such as Mercurial and BitBucket. While other sets of information can be added to the model, these were sufficient for our current prototype.

The `ActiveFile` model parameter is the most frequently updated aspect of the model. As the developer moves from file to file

Table 2: Context Model elements.

Current Focus
`ActiveFile`
Each Local Repository
`Path`
`Version Control Type`
`OriginURL`
`UserName`
`CurrentBranch`
`FileStatus`
Each Remote Repository
`OpenAssignedIssues[]`
`Collaborators[]`
Other Services
`BlameService`
`TestService`
`ReviewerAssignmentService`

in any text editor or IDE, the full path of the active file is noted and persisted in the `ActiveFile` field. The context model is also populated with information about all version control repositories it finds on the developer's machine. From these local repositories, using the `OriginURL`, it is also able to populate information about the developer's online code hosting repositories. The path component of `ActiveFile` lets the model index into the list of version control repositories to get additional details including the remote online repository, if applicable. Our prototype also allows developers to exclude specific repositories and paths from being tracked by the context model and only minimal information about the state of the command is exchanged with Amazon to ensure the privacy of the user.

We designed our context model to pull changes from a developer's computer when they interact with Devy. Due to the limited size of our context model, the pull-based architecture is sufficient. However, for more advanced models, a push-based architecture where the model is initialized at startup and continuously updated by external change events would be preferable to avoid delaying the conversational interface.

Extending the model is only required if the developer wishes to support workflows that require new information. Model extensions can be either in terms of pre-populated entries (push-based above), or pointers to services that can be populated on demand (pull-based). For example, the `TestService`, which takes a list of files and returns a list of tests that can run, can be pointed to any service that conforms to this API (to enable easy customization of test selection algorithms). If developers wanted more fine-grained information such as the current class, method, or field being investigated, they could add a relevant entry to the model and populate it using some kind of navigation monitor for their current text editor or IDE.

3.3 Intent Service

The intent service does the heavy lifting of translating the limited conversational tokens and combining it with the context model to determine the developer's intent. This intent is then executed for the developer in the form of workflow actions. The context model is updated as the actions execute since their outcomes may influence subsequent steps of the workflow.

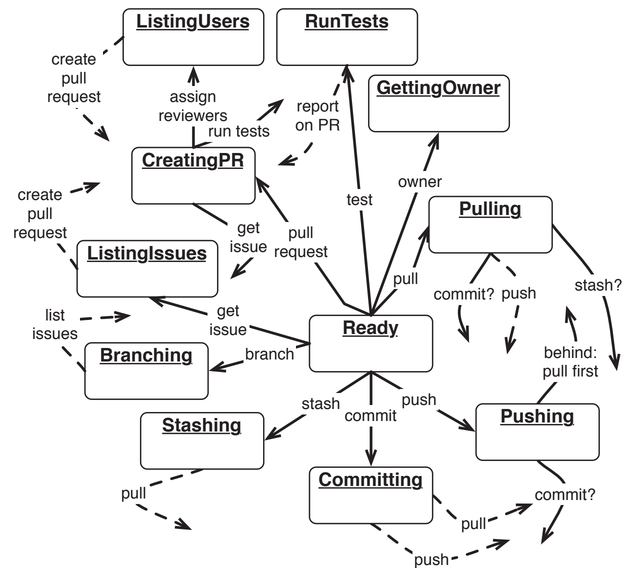

Figure 2: Devy's finite state machine for handling workflows. Stack-push transitions are shown with solid lines while stack-pop transitions are shown with dotted lines. For readability, some arrows do not connect with their state. However, all lines are labelled with the action that causes the state transition and correspond to the next state. Edges between the FSM and the Context Model are elided for clarity.

The conversational layer provides the intent service with extremely simple input commands (e.g., a single verb or noun). The intent service uses a stack-based finite state machine (FSM) to reason about what the input command means in this context. While more constrained than plan-based models, FSMs are simple to implement and are sufficient for the purposes of evaluating the potential of CDAs. The complete FSM for our version control intent service is shown in Figure 2. Within the FSM, transitions between states define workflow steps while states contain the logic needed to prepare and execute low-level actions. Each state is aware of other states that may need to be executed before they can successfully complete (e.g., a pull may be required before a push if the local repository is behind the remote repository). We use a stack-based FSM because workflow actions frequently depend on each other. By using a stack, we are able to just push commands on the stack and allow the execution to return to the right place in an understandable way. These potential return edges are denoted by the dotted arrows in Figure 2; for example, `Stashing` can be accessed either directly by the developer from the `Ready` state, or as a consequence of a `Pulling` precondition. The states in the FSM make heavy use of the context model to provide values for their parameters.

4 STUDY METHOD

The long-term objective of our research is to enable intent-based workflows without software developers having to continuously map their intents to low-level commands. Therefore, we are investigating

when and how software developers would use a conversational developer assistant that supports intent-based workflows. Specifically, we are examining the following research questions:

RQ1 How well can a conversational developer assistant approach support basic development tasks related to version control?

RQ2 For which workflows would a conversational developer assistant be useful to developers and why?

To answer our research question, we developed the voice-enabled CDA, Devy, as a prototype (see Section 3), piloted it with several student developers, and then conducted a mixed methods study with 21 professional software developers. The study was a combination of an experiment with Devy and semi-structured interviews.

4.1 Participants

We recruited 21 professional software developers (2 female, 19 male) from 6 local software companies of varying size. Participants had an average of 11 (±8) years of professional development experience and an average of 15 (±11) years[1] of programming experience. Participants were classified as either junior developers (8) or senior developers (13) based on job title. All participants had experience using version control systems and all but 1 had experience with Git.

Participants were recruited through personal contacts and recruiting emails. To pique interest and foster participation, we included a short video[2] introducing Devy and demonstrating it with the use case of determining the owner of an open file. In addition, we incentivized participation with a lottery for two Amazon Echo Dots amongst all participants. To participate in our study, subjects had to be software developers and be familiar with some version control system.

4.2 Procedure

The study consisted of three parts: (1) a brief semi-structured interview to ask about developers' tasks and workflows as well as about the possible value of a conversational assistant to support these, (2) an experiment with Devy comprised of two study tasks, and (3) a follow-up semi-structured interview on the experience and use of a CDA. We piloted our study and adapted the procedure based on the insights from our pilots. We chose this three step procedure to stimulate developers to think about a broad range of workflows and how a CDA might or might not help, as well as to avoid priming the participants too much and too early with the functionality that our current Devy prototype provided. The order and sample questions of the parts of our study are illustrated in Table 3. The study was conducted in quiet meeting rooms at the participant's industrial site.

Interview (Part One). To have participants reflect upon their work and workflows, we started our first semi-structured interview by asking general questions about participants' work days and then more specifically about the tasks they are working on as well as the specific steps they perform for these tasks (see Table 3 for sample questions). We then introduced Amazon's Alexa to participants. To get participants more comfortable with interacting with Alexa, we had them ask Alexa to tell them a joke. Next, we asked participants about the possible tasks and workflows that a conversational assistant such as Alexa could help them with in their workplaces.

Table 3: Order, sample questions, and tasks from our mixed methods study.

Interview - Part One	
1.1	Walk me through typical development tasks you work on every day.
1.2	How do you choose a work item; what are the steps to complete it?
1.3	How do you debug a failed test?
2	To help you get familiar with Alexa, ask Alexa to tell us a joke.
3	Can you think of any tasks that you would like to have "magically" completed by either talking to Alexa or by typing into a natural language command prompt?
Experiment - Interaction Task (T1)	
Complete the following tasks:	
	Launch Devy by saying "Alexa, launch Devy" [..]
T1.1	Using Devy, try to get the name of the person whom you might contact to get help with making changes to this 'readme' file.
T1.2	Next, make sure you are on branch 'iss2' and then make a change to this 'readme' file (and save those changes).
T1.3	Finally, make those changes available on GitHub.
Experiment - Demonstration Task (T2)	
Complete the following tasks:	
T2.1	Say "Alexa, tell Devy to list my issues." to list the first open issue on GitHub. List the second issue by saying "Next", then stop by saying "Stop". Notice that the listed issues are for the correct repository.
T2.2	Say "Alexa, tell Devy I want to work on issue 2." to have Devy prepare your workspace for you by checking out a new branch.
T2.3	Resolve the issue: comment out the debug console.log on line 8 of `log.ts` by prepending it with //. Save the file.
T2.4	Say "Alexa, tell Devy I'm done." to commit your work and open a pull request. Devy will ask if you want to add the new file; say "Yes". Next, Devy recommends up to 3 reviewers. You choose any you like. When completed, Devy will say it created the pull request and open a browser tab showing the pull request. Notice the reviewers you specified have been added. Also, notice that tests covering the changes were automatically run and the test results were included in a comment made by Devy.
Interview - Part Two	
1	Imagine that Devy could help you with anything you would want, what do you think it could help you with and where would it provide most benefit?
2	Are there any other tasks / goals / workflows that you think Devy could help with, maybe not just restricted to your development tasks, but other tools you or your team or your colleagues use?
3	When you think about the interaction you just had with Devy, what did you like and what do you think could be improved.
4	Did Devy do what you expected during your interaction? What would you change?
5	Do you think that Devy adds value? Why or why not?

Experiment. To give participants a more concrete idea of a CDA and investigate how well it can support basic workflows, we conducted an experiment with Devy on two small tasks. For this experiment, we provided participants a laptop that we configured for the study. We connected the Amazon Echo Dot to the laptop for power and we connected the laptop and the Echo Dot to the participant's corporate wireless guest network.

[1] Missing data for two participants.
[2] http://soapbox.wistia.com/videos/HBzPb4ulqlQI

The tasks were designed to be familiar to participants and included version control, testing, and working with issues. The objective for the first task—interaction task—was to examine how well developers interact with Devy to complete a task that was described on a high-level (intent-level) in natural language. This first task focused on finding out who the owner of a specific file is and making a change to the file available in the repository on GitHub. The task description (see also Table 3) was presented to the participants in a text editor on the laptop. For the task we setup a repository in GitHub with branches and the file to be changed for the task.

The objective for the second task—demonstration task—was to have participants use Devy for a second case and demonstrate its power and potential of mapping higher-level intents to lower-level actions, e.g. from telling Devy that one "is done" to Devy committing and pushing the changes, running the relevant tests automatically and opening the pull request in a web browser tab (see Table 3 for the task description).

Interview (Part Two). After the experiment, we continued our interview. Interacting with Devy and seeing its potential might have stimulated participants' thinking so we asked them further about which scenarios an assistant such as Devy would be well-suited and why. We also asked them about their experience with Devy during the two experimental tasks (see Table 3 for the questions). Finally, we concluded the study by asking participants demographic questions and thanking them for their participation.

4.3 Data Collection and Analysis

The study, including the interviews and experiment, lasted an average of 36 (±4) minutes. We audio recorded and transcribed the interviews and the experiment and we took written notes during the study.

To analyze the data, we use Grounded Theory methods, in particular open coding to identify codes and emerging themes in the transcripts [17]. For the open coding, two authors coded five randomly selected transcripts independently and then discussed and merged the identified codes and themes. In a second step, we validated the codes and themes by independently coding two additional randomly selected transcripts. For the coding of all transcripts, we used the RQDA [6] R package. In this paper, we present some of the most prominent themes, notably those that illustrate the most common use cases, the benefits, and the shortcomings of CDAs.

From the experimental task T1, we derived a count based on the number of times a participant had to speak to Devy before they were able to complete a subtask. We adjusted this count by removing 55 attempts (out of 175) that failed due to technical issues, i.e. connectivity problems or unexpected application failures of Alexa, due to a participant speaking too quietly, and due to participants trying to invoke Devy without using the required utterance of "Alexa, ask/tell Devy...".

5 RESULTS

In this section we present the results for our two research questions that are based on the rich data gathered from the experimental study tasks and the interview. First, we report on the participants' interaction and experience with our CDA Devy. Then we report on the workflows and tasks that a CDA might support as well as its benefits and challenges.

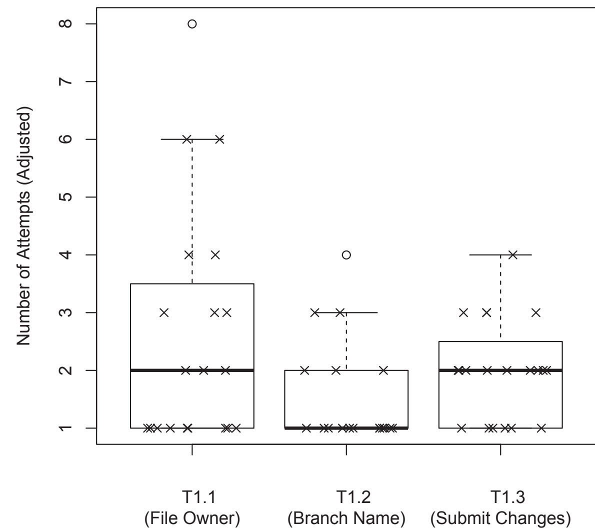

Figure 3: Adjusted number of attempts required to complete each task of T1 across 20 participants.

5.1 Completing Development Tasks with Devy

Overall, all participants were able to complete all subtasks of T1 and T2 successfully with Devy. Many participants expressed that Devy was "*neat*" *(P17)* and "*cool*" *(P18)* and some also stated that Devy did more than they expected. For instance, P9 explicitly stated "*[Devy] exceeded my expectations*" while P8 "*[was] surprised at how much it did [..] it actually did more than [..] expected*".

For the first experimental task T1, we examined if participants were able to interact with Devy and complete specific subtasks that were specified on the intent level rather than on the level of specific and executable (Git) commands. Figure 3 shows the number of Devy interactions (attempts) that it took each participant to complete each of the three subtasks of T1. The numbers in the figure are based on 20 participants (one participant completed T2 before T1 and was excluded due to learning effects); the values were adjusted by removing attempts that failed due to technical issues (see Section 4.3). Across all three subtasks, **participants used very few attempts to complete the subtasks** with an average of two attempts for T1.1 and T1.3 and a single attempt for T1.2.

Subtask T1.1 required getting the name of the person who made the most changes to an open file. This task had the highest variance with one participant taking 8 attempts since he had never used Git before. Six participants required some guidance. This was largely due to Devy only being trained with three utterances that all focused on file ownership and none that focused on the number of changes. In fact, seven participants used an utterance similar to "*who has made changes*" (P1, P2, P3, P4, P14, P17, P19) on their first attempt. This shows that either developers require training to learn the accepted utterances or, better yet, that **Devy should support a broad set of utterances**. One participant compared it to the specific and rigid syntax of command-oriented approaches:

"*Multiple ways you can tell it to do the same thing [because] it might be advantageous where [you] might forget the exact terminology.*" *(P10)*

In their first interactions with Devy, most participants (16 out of 20) did not take advantage of its automatic context tracking and instead included the specific file name in their utterances. This was due to participants thinking of Devy as a traditional tool, "*making the assumption [Devy] would work just like the command line client*" *(P19)* and they "*expected [Devy] to be kind of simple and dumb*" *(P7)*. As their sessions progressed, participants started to express their intentions more naturally and more vaguely, for instance by replacing the file name with "*this*" or "*this file*", and **participants appreciated the automated context tracking**:

"*I was just thinking it knows the context about what I'm talking about. That's kind of cool.*" *(P2)*

Subtask T1.2 required getting the checked-out branch and only took one attempt on average. All but two participants were able to complete the subtask without guidance. Thereby, **participants used various utterances to interact with Devy** from ones that were close to the Git commands "*Alexa, tell Devy to checkout branch iss2*" *(P15)* to less command-oriented phrases "*Alexa, ask Devy to get onto branch iss2 [..]*" *(P8)*. The two participants that did not complete this task accidentally skipped it. The one participant that took 4 attempts paused between starting Devy and issuing the command.

Subtask T1.3 focused on submitting changes to GitHub. Participants took an average of 2 attempts to complete this and had a lower variance than for T1.1. While 14 participants followed the Git command line interaction closely by first committing the changes and then pushing them, the other 6 **participants took advantage of some of Devy's automation** and for example directly said "*Alexa, tell Devy to push to GitHub*" *(P15)* which resulted in Devy committing and pushing the changes. Also, for this subtask, most participants took advantage of Devy's context model and omitted some of the specifics from their phrases, such as what exactly should be committed or pushed.

The second experimental task T2 was to demonstrate Devy's potential. Since all utterances were specified in the task description, and no participants had problems following the steps, we will not include any further analysis of this task.

Observation. *Participants were able to use our conversational developer assistant to successfully complete three common development tasks without explicit training, with very few attempts, and by taking advantage of the automatic context tracking.*

5.2 CDA Benefits & Scenarios

Participants reported a myriad of scenarios and situations in which a CDA could enhance their professional workflow.

One common workflow **supports multi-step and cross-application tasks**. Several development tasks require a certain "*process*" *(P8)* or sequence of steps to be completed that oftentimes require developers to switch between different applications. An example referred to by several participants was the sharing of changes:

"*Once I've committed everything locally, I'll push it to GitHub. I'll go to GitHub in my web browser, create a new pull request, write out a description of the change [..] and how I tried to accomplish that [..] Once the pull request is created, I'll go to Slack and post a message on our channel and say there is a new PR*" *(P15)*

Participants indicated the cost and effort of these multi-step and cross-application tasks and how a conversational assistant would **reduce the necessary application/context switches** and allow developers to not "*lose [..] concentration on the thing I'm looking at*" *(P6)* and stay more focused:

"*If you could do some [of these tasks] automatically, just by talking, I'd be really happy because I usually have a ton of consoles and switching over really confuses me when you have so many screens open. Just alt-tabbing between them consistently, even if you do that like 9 out of 10 times successfully, at the end of the day you start getting sloppy and holding those trains of thought in mind would probably be simpler if you weren't changing screens so often*" *(P7)*

"*Today, I think I had like 20 emails all related to my pull request and it was all just single comments and I have to link back [to the pull request..] and then come back [to my email client] and then delete, and then go back and [..]. So there's a lot of back and forth there. Those are the main things that I feel: 'oh these are taking time and it shouldn't...'*" *(P3)*

A CDA is also considered particularly beneficial for the **automatic mapping of higher-level tasks to commands**:

"*Anything that helps you stay in the flow is good, so if I can do these higher level tasks with a brief command to some place rather than break them down into a sequence of git commands plus switching to the browser plus doing yet another thing interspersed with some manual reading, it would be a win.*" *(P19)*

This automatic aggregation of multiple steps is seen as a "*simplification*" *(P7)* by participants:

"*[If] we have a bot tell me the IP of my deployed AWS container rather than a 10 step ssh-based process to get it that would be very simple [..and] interacting with a voice assistant to get information [..] out of the development ecosystem would be useful.*" *(P18)*

By abstracting the specific low-level actions, the automatic mapping **reduces the need for memorization** of the commands, which reduces errors and saves time:

"*There are too many command line steps that you can get wrong*" *(P18)*

"*A lot of the time you know what it is in your head, but you still gotta find it. So that's the stuff [..] this would be really helpful for.*" *(P8)*

Participants mentioned that this can be valuable for infrequent but recurring—"*once in a while*" *(P11)*—tasks, since they "*do [them] often enough to want a better interface but seldom enough that [they] can't remember all the right buttons*" *(P10)* or they can't remember the "*crazy flags that you've gotta remember every single time*" *(P8)*.

By continuously switching between applications, developers have to frequently re-establish their working context. For instance, after committing and pushing a code change using a command line tool, developers often "*have to go to the browser, open a new tab, go to GitHub and find the pull request*" *(P15)* which can be a "*a pain in the arse*" *(P8)*. In general, when switching between applications, participants need to do a lot of "*admin work*" *(P14)* just to ensure that the applications are "*in sync*" *(P14)*. Therefore, a major benefit of a CDA that automatically tracks context in the background is that it **reduces the explicit specification of context**.

By automatically aggregating multiple steps and keeping track of the current context, participants also thought that a CDA can **support information retrieval**, especially when "*there isn't really a good interface*" *(P1)* for querying and it can speed up the process:

"*So, looking at the context of what I'm doing and then like highlighting an error text and then copying it and pasting it into Google and looking for it. And then looking down for some of the best matches. Even just 'Devy look this up for me'.*" *(P2)*

"*Right now, you need to do two or three commands and know what all the change list numbers are [..to] look up all the information [about who last touched a file].*" *(P20)*

Instead of just automating and aggregating tasks, participants suggested that a CDA that tracks a developer's steps and context could help to **enforce workflows** and make sure that nothing is forgotten:

"*There are certain flows that often go together. When I start a day, I often need to check the same things [..and it is] not easy to check off the top of my head, so I forget to do that sometimes [..] so that type of thing would be great to do all in one shot. So where am I [which branch], what is my status, do I need to rebase, and if I need to rebase, do it [..]*" *(P3)*

In case a developer does not follow the usual workflow, the context tracking can come in handy and allow her to **go back in history**:

"*sometimes I just go too far down a path. And I've gone through three or four of those branches in my mind and I know I need to go back but because I [..] only want to go part way back, that just becomes really difficult. So if there was some simple way it could recognize that I'm far enough down a path [..it] would be amazing if I could then realize that I have screwed up, rewind 10 minutes, commit and then come back to where I am now.*" *(P21)*

Several participants noted that the additional communication channel offered by Devy could be utilized to **parallelize tasks** that would otherwise require a switch in context and focus. The archetypal case of this was setting reminders:

"*Yeah, I think of them like if I'm coding and I have an idea of another approach I could take but I want to finish the current one I'm doing, I'll throw myself in a little Slack reminder and then I get a notification in the time I specified.*" *(P21)*

However, this idea is more general and can be particularly useful in cases where the secondary task may take some time to complete:

"*Where it'd be useful is where I'm in the middle of one task and I want another being done. If I'm working on one part of it, either debugging or editing some code and I want something to go on in the background... Like we've got a build system so maybe I want to start the build for another tool that I will be using soon and I don't want to switch over to start that.*" *(P11)*

"*If I have a pull request and I'm waiting for it to be approved and I have nothing else to do in the meantime, I'm going to make lunch. I could just be cooking and I could just be like: 'has it been approved yet?' and if it has then merge it before someone else gets their stuff in there. Oh, that would be great.*" *(P3)*

Seven participants explicitly mentioned that a voice-activated assistant provides an **alternative to typing** that allows tasks be performed when the developer's hands are "*busy*" *(P11)* or "*injured*" *(P16,P20)* and that "*as intuitive as typing is [..], talking is always going to be more intuitive*" *(P12)*. Similarly, it provides an **alternative to interacting with GUIs** that "*waste a lot of time just by moving the mouse and looking through menus*" *(P7)* or to navigate code with context, for example by asking "*where's this called from*" *(P10)* or "*what classes are relevant to this concept*" *(P13)*.

Observation. *There are a large number of development tasks in participants' workflows that are currently inefficient to perform due to their multi-step and cross-application nature. A conversational developer assistant might be able to support these scenarios by reducing application switches, the need for context specification and memorization, and by supporting parallelization of tasks.*

5.3 CDA Challenges

Participants also raised several concerns about their interaction with Devy and conversational developer assistants more generally. The predominant concern mentioned by several participants was the **disruptiveness of the voice interface** in open office environments:

"*I work in a shared workspace so there would have to be a way for us to have these dialogs that are minimally disruptive to other people.*" *(P19)*

"*I imagine a room full of people talking to their computers would be a little chaotic.*" *(P2)*

Further concerns of the voice interaction are its "*accuracy*" *(P11)* and that the verbal interaction is slow:

"*I didn't really enjoy the verbal interaction because it takes longer.*" *(P2)*

"*It feels weird to interrupt [Devy]. That's probably more of a social thing [..] it's a voice talking and you don't want to interrupt it and then you have to end up waiting*" *(P15)*

While Devy was able to automate several steps, participants were concerned about the **lack of transparency** and that it is important to know which low-level actions Devy is executing:

"*The downside is I have to know exactly what it's doing behind the scenes which is why I like the command line because it only does exactly what I tell it to do.*" *(P8)*

This can be mitigated by providing more feedback, possibly through channels other than audio:

"*I think for me, when [Devy] is changing branches or something, I'd probably want to see that that has happened in there. Just some indication visually that something has happened. I mean it told me so I'd probably get used to that too.*" *(P6)*

However, there is some disagreement on exactly how much feedback is wanted:

"*I liked that there was definitely clear feedback that something is happening, even for things that take a bit of time like git pushes.*" *(P1)* For a conversational developer assistant **completeness**—the number of intents that the CDA is able to understand—is important. Participant P14 made the case that "*the breadth of commands needs to be big enough to make it worthwhile.*"

This completeness is also related to challenges in **understanding the intent** of all possible utterances a developer could use:

"*It's frustrating to talk to something that doesn't understand you. Regardless of how much more time it takes then another method, it would still be more frustrating to argue with a thing that fundamentally doesn't feel like it understands me.*" *(P12)*

Finally, since developers use a diverse set of tools in a variety of different ways and "*everyone's got a little bit of a different workflow*" *(P2)*, it is necessary for CDAs to support **customization**. For this, one could either "*create macros*" *(P2)* or have some other means for adapting to each developer's particular workflow so that Devy "*could learn how [people are] using it*" *(P9)*. This aspect is related to completeness but emphasizes altering existing functionality to suit the individual or team.

Observation. *Participants raised several concerns for conversational developer assistants related to disruptiveness of voice interactions, the need for transparency, completeness, and customization.*

5.4 Summary

Ultimately, industrial developers were able to successfully perform basic software development tasks with our conversational developer assistant, providing positive evidence for **RQ1**. In terms of **RQ2**, CDAs appear to be most useful for simplifying complex workflows that involve multiple applications and multiple steps because of their unnecessary context switches which interfere with developer concentration.

6 RELATED WORK

We build upon a diverse set of related work in this paper. To support developers in their tasks, researchers have long tracked development context in order to provide more effective analyses and to surface relevant information. The emerging use of bots for software engineering also shows promise for automating some of the tasks, improving developers effectiveness and efficiency. Finally, natural language interfaces show increasing promise for reducing complexity and performing specific development tasks.

6.1 Development Context

Our model of task context is fundamental to enabling Devy to provide a natural interface to complex workflows. Previous work has looked at different kinds of context models. Kersten and Murphy provide a rich mechanism for collecting and filtering task context data, specifically about program elements being examined, as developers switch between different tasks [7]. Our context model is more restrictive in that we mainly track the current task context: past contexts are not stored. Concurrently, our context model includes much more detail about non-code aspects relevant to the developer, their project, and their teammates.

Other systems have looked at providing richer contextual information to help developers understand their systems. For example, TeamTracks uses the navigation information generated by monitoring how members of a team navigate through code resources to build a common model of related elements [4]. MasterScope provides additional context about code elements as they are selected in an IDE [18]. The similarity between task contexts can also be used to help identify related tasks [10]. Each of these systems demonstrates the utility context models can confer to development tasks. Our work extends these prior efforts by providing a context model appropriate for conversational development assistants.

6.2 Bots for SE

In their Visions paper, Storey and Zagalsky propose that bots act as "conduits between users and services, typically through a conversational UI" [16]. Devy clearly sits within this context: the natural language interface provides a means for developers to 'converse' with their development environment, while the provided workflows provide an effective means for integrating multiple different products within a common interaction mechanism. Further to their bot metaphor, Devy is able to interpret the conversation to perform much more complex sequences of actions based on relatively little input, only checking with the developer if specific clarification is required. As Storey and Zagalsky point out, there is a clear relationship between bots and advanced scripts. We firmly believe that the conversational interface, combined with the context model, moves beyond mere scripting to enable new kinds of interactions and workflows that could not be directly programmed. One study participant also pointed out that "*it's nice to have the conversation when there are things that are not specified or you forgot to do; that's when you want to get into a dialog. And when you're in the zone, then you can just tell it what to do*" *(P19)*, showing further benefit of the conversational UI beyond scripting itself.

Acharya et. al. also discuss the concept of `Code Drones` in which all program artefacts have their own agent that acts semi-autonomously to monitor and improve its client artefact [1]. One key aspect of these drones is that they can be proactive instead of reactive. While Devy is not proactive in that it requires a developer to start a conversation, it can proactively perform many actions in the background once a conversation has been started, if it determines that this appropriate for the given workflow. Devy also takes a different direction than `Code Drones` in that rather than attaching drones to code artefacts, Devy primarily exists to improve the developer's tasks and experience directly, rather than the code itself.

6.3 Natural Language Tools for SE

A number of tools have been built to provide natural language interfaces specifically for software engineering tasks.

The notion of programming with natural language is not new (having first been described by Sammet in 1966 [15]). Begel further described the diverse ways in which spoken language can be used in software development [2]. More recently, Wachtel et. al. have investigated using natural language input to relieve the developer of repetitive aspects of coding [19]. Their system provides a mechanism for users to specify algorithms for spreadsheet programs using natural language. In contrast, Devy does not try to act as a voice front-end for programming: it works more at a workflow level integrating different services.

Others though have looked at natural language interfaces as a means for simplifying the complex tools used for software development. One early relevant example of this by Manaris et. al. investigated using natural language interfaces to improve the abilities of novice users to access UNIX tools in a more natural way [12].

NLP2Code provides a natural language interface to a specific task: finding a relevant code snippet for a task [3]. NLP2Code takes a similar approach to Devy in that supports a specific development task, but unlike Devy does not use a rich context model, nor does it involve more complex development workflows.

7 DISCUSSION

In this section we discuss threats to the validity of our study and future work suggested by our study participants.

7.1 Threats to Validity

The goal of our study was to gain insight into the utility of conversational developer assistants in the software engineering domain. As with any empirical study, our results are subject to threats to validity.

Internal validity. We elicited details about participants' workflows before they interacted with our prototype to mitigate bias and again after using Devy to capture more detail. Despite this, it is possible that participants did not describe all ways Devy could impact their workflows; given more time and a wider variety of sample tasks, participants may have described more scenarios. While we followed standard open coding processes, other coders may discern alternative codes from our interview transcripts.

External validity. Though our 21 interviews with industrial developers yielded many insights, this was a limited sample pulled from our local metropolitan region. While participants had differing years of experience and held various roles at six different organizations, each with a different set of workflows, our findings may not generalize to the wider developer community.

7.2 Future Work

The feedback we received from industrial developers was broadly positive for our prototype conversational developer assistant. Thankfully, our participants had many great suggestions of ways to extend and improve Devy to make it even more effective in the future.

The most pressing piece of future work is to implement **alternative conversational layers** for Devy, specifically a text-based ChatBot-like interface. Participants mentioned this would be especially beneficial in open-plan offices (which all participants used). It also avoids requiring Devy-specific hardware.

Currently, Devy can be extended through the intent service by wiring up new states in the FSM. This requires the same amount of work as creating scripts, although enables better integration with existing states than simple scripting. Based on participant feedback, **supporting a more parameterized view of how the states are connected to form custom workflows** seems like a reasonable tradeoff between complete scripting and a fully autonomous agent. Participants were also forthcoming with suggestions for a diverse set of future workflows that could define the out-of-box-workflows for version control, debugging, testing, collaboration, task management and information retrieval.

A large step beyond this would be for the CDA to support generic workflows out-of-the-box that can **self-adapt to better enable user-specific custom workflows** without user intervention but based on their own usage patterns.

Several participants also wished for **tighter source code integration**. The intent of this integration was to perform more query-based questions of the specific code elements they were looking at without interrupting their current task. For example:

"*the thing people want the most...are abstract syntax trees. I think it is something that would offer a lot of power if you also had assistive technology layered on top.*" *(P8)*

Using **lightweight notes and reminders**, CDAs might enable semantic undos that could be further maintained using the context model to rollback changes to meaningful prior states.

Enabling CDAs to **proactively take action** in terms of awareness or in response to external events was widely requested:

"*influence the output of what I'm working on...by [notifying] me about getting off my regular pattern, that would be the most valuable.*" *(P8)*

This could also help by **preventing mistakes before they happen**:

"*If I tell it to commit and [there are an unusual number of changes], it should confirm.*" *(P15)*

Next, **extending support for industrial tools** to those commonly used by industrial teams will enable Devy to be deployed in a wider variety of practical contexts.

Participants were also enthusiastic about the potential for support for **enhanced cross-application workflows** that otherwise cause them to context switch or 'copy-and-paste' between independent systems. We will further investigate extending support for these kinds of tasks that force developers to context switch.

Finally, we built our prototype using the Alexa service and our intent service to handle the natural language discourse and map it to workflow actions. To support further workflows and ease the natural language discourse with developers, we will examine whether and how to extend the underlying discourse representation structure.

8 CONCLUSION

In this paper, we have explored the potential of conversational agents to support developer workflows. In particular, we have described Devy, a conversational development assistant that enables developers to invoke complex workflows with only minimal interaction using a natural language conversational interface. Through its context-aware model, Devy supports rich workflows that can span multiple independent tools; this frees the developer to offload these low-level actions and enables them to focus on their high-level tasks.

Using our Devy prototype as a technology probe, we evaluated our approach in a mixed methods study with 21 industrial software engineers. These engineers were able to use Devy successfully and appreciated that they did not need to specify and memorize multi-step workflows and that it reduced context switches. They additionally identified a concrete set of challenges and future directions that will improve the utility of future CDAs.

Ultimately, the Devy prototype demonstrates that developers can successfully launch complex workflows without interrupting their current tasks while reducing developer effort. We believe that that future conversational developer assistants will have the ability to improve developer's productivity and/or effectiveness by allowing them to focus on their core development tasks by offloading meaningful portions of their workflows to such automated agents.

REFERENCES

[1] Mithun P. Acharya, Chris Parnin, Nicholas A. Kraft, Aldo Dagnino, and Xiao Qu. 2016. Code Drones. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 785–788.

[2] Andrew Begel. 2006. *Spoken Language Support for Software Development*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[3] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 628–632.

[4] Robert DeLine, Mary Czerwinski, and George Robertson. 2005. Easing Program Comprehension by Sharing Navigation Data. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC)*. 241–248.

[5] Victor M. González and Gloria Mark. 2004. Constant, Constant, Multi-tasking Craziness: Managing Multiple Working Spheres. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 113–120.

[6] Ronggui Huang. 2017. *RQDA: R-based Qualitative Data Analysis*.

[7] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 1–11.

[8] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 492–501.

[9] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1987. Mental Models and Software Maintenance. *Journal of Systems and Software (JSS)* 7, 4 (1987), 341 – 355.

[10] Walid Maalej, Mathias Ellmann, and Romain Robbes. 2017. Using Contexts Similarity to Predict Relationships Between Tasks. *Journal of Systems and Software (JSS)* 128 (2017), 267 – 284.

[11] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4, Article 31 (2014), 37 pages.

[12] Bill Z. Manaris, Jason W. Pritchard, and Wayne D. Dominick. 1994. Developing a Natural Language Interface for the UNIX Operating System. *Proceedings of the Conference on Human Factors in Computing Systems (CHI)* 26, 2 (1994), 34–40.

[13] André N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. The Work Life of Developers: Activities, Switches and Perceived Productivity. *IEEE Transactions on Software Engineering (TSE)* 43, 12 (2017), 1178–1193.

[14] André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. 2014. Software Developers' Perceptions of Productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 19–29.

[15] Jean E. Sammet. 1966. Survey of Formula Manipulation. *Communications of the ACM (CACM)* 9, 8 (1966), 555–569.

[16] Margaret-Anne Storey and Alexey Zagalsky. 2016. Disrupting Developer Productivity One Bot at a Time. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 928–931.

[17] Anselm Strauss and Juliet M. Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications.

[18] Warren Teitelman and Larry Masinter. 1981. The Interlisp Programming Environment. *Computer* 14, 4 (1981), 25–33.

[19] Alexander Wachtel, Jonas Klamroth, and Walter F. Tichy. 2017. Natural Language User Interface For Software Engineering Tasks. In *Proceedings of the International Conference on Advances in Computer-Human Interactions (ACHI)*, Vol. 10. 34–39.