Context-Aware Conversational Developer Assistants

by

Nicholas Bradley

B.Sc., Queen's University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Context-Aware Conversational Developer Assistants

submitted by **Nicholas Bradley** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Reid Holmes, Computer Science
Supervisor

Gail Murphy, Computer Science
Supervisory Committee Member

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 thesis I 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.

Lay Summary

Software engineers use many different tools for any one project. They work with millions of lines of computer code and run their code through various independent tools to help edit, build and test systems and for project management to get their programs running smoothly. Needing all of these tools can complicate engineers' workflows because they each use a unique syntax and you have to understand how to put them together. In this thesis, I describe Devy, a tool for software engineers that enables Amazon Alexa to take care of mundane programming tasks, helping increase productivity and speed up workflow. I also present an evaluation of Devy's ability to support the workflows of 21 professional software engineers and discuss some of Devy's benefits and challenges.

Preface

All of the work presented henceforth was conducted in the Software Practices Laboratory at the University of British Columbia. All projects and associated methods were approved by the University of British Columbias Research Ethics Board [certificate #H17-01251].

A version of this material has been published [Nick C. Bradley, Thomas Fritz, Reid Holmes. *Context-Aware Conversational Developer Assistants*. ICSE 40:993–1003, 2018]. I was the lead investigator, responsible for all major areas of concept formation, data collection and analysis, as well as manuscript composition. Thomas Fritz was involved in study design and participant recruitment and contributed to manuscript edits. Reid Holmes was the supervisory author on this project and was involved throughout the project in concept formation and manuscript composition.

Table of Contents

List of Tables

List of Figures

Glossary

API	Application Programming Interface
CDA	Conversational Developer Assistant
FSM	Finite State Machine
GUI	Graphical User Interface
IDE	Integrated Development Environment
NLP	Natural Language Processing
UI	User Interface
URL	Unqiue Resource Locator

Acknowledgment

I would first like to thank my supervisor Dr. Reid Holmes for his guidance and support whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank Dr. Thomas Fritz for helping me design the study, recruit participants, and for seeing me through my first qualitative analysis and all the students in the Software Practices Lab who helped pilot the study and who make coming to work a pleasure each day.

I would like to acknowledge Dr. Gail Murphy as the second reader of this thesis, and I am gratefully indebted to her for her very valuable comments on this thesis.

I would also like to thank the 21 professional software engineers (and their employers and managers) who dedicated their time to trying out Devy. Without their passionate participation and valuable insights, this work could not have been successfully completed.

Finally, I must express my very profound gratitude to my wife Susanne for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

This accomplishment would not have been possible without the support of everyone mentioned and many others. Thank you.

Chapter 1

Introduction

The process of developing software is complex. Developers discuss, write, test, debug, version, and deploy code with the aid of a plethora of different tools which often need to be used in concert to achieve a high-level task. Combining these tools to complete tasks can be cumbersome and inefficient since developers need to manually execute tools in correct sequences (workflows) and specify correct parameter values (contextual information). These manual steps force developers to maintain mental models combining data from a variety of disparate sources including project source code, version control systems, issue trackers, discussion threads, and test and deployment environments. Manual processes are also inefficient since developers need to switch frequently between tools and contexts [5, 10, 11] and because they are more error prone. Additionally, tools today are designed for different platforms including the web, desktop, and terminal requiring developers to learn and work with different interfaces and to act as bridges moving information between tools. For example, terminal tools are often considered to be very powerful but their rigid syntax and limited support for feature discovery can make them difficult to use. Web tools on the other hand aim to be intuitive but typically lack the customizability of terminal or desktop tools. For common tasks, developers can learn the required steps of the workflow but even small variations in the task may require significant changes to the workflow. Many tasks can also be completed in several different ways which could lead developers to learn less efficient workflows, or ones that cause problems with team processes. A complete workflow for

sharing code changes is given in Chapter 2.

Tools operate within a certain context which is specified by both the system (e.g. the user's home directory) and by the developer (e.g. the particular file to operate on). While the system context is specified automatically, the developer needs to understand, obtain, and transform the remaining contextual information and provide it to the tool. This can be cognitively demanding on the developer and can distract them from their actual task. Alternatively, we can track developer activities and automatically extract and synthesize much of the necessary contextual information. This approach is commonly employed in integrated development environments where developer interactions are used to provide more intelligent tools. In our case, capturing and maintaining this information is more challenging since it comes from a variety of tools but allows us to run multi-step workflows that span tools.

Techniques for automating developer tasks have been around for decades and typically take the form of scripts or macros. Scripts, consisting of sequences of commands, are commonly used in software development due to their versatility in connecting different tools and services together; however, they can be brittle to changes in the system, tool, or task and can be expensive to implement initially. Macros work instead by recording the steps developers take when completing a task but they are often only available within specific tools and may still require effort to make them robust. More recently, tools for testing, deployment, and service integration have become popular choices for automating parts of the software development process but impose their own restrictions. In general, 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 thesis, I argue that automated assistants are an intuitive and robust method of automating workflows and that they benefit developers by automatically mapping intentions to commands, eliminating application and mental context

switches, and reducing the need to manually maintain and specify context. To support this claim, 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 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 thesis 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 Chapter 2, our approach in Chapter 3 and our experiment in Chapter 4 and 5. Related work, discussion, and conclusions follow in Chapters 6–8.

Chapter 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 2.1 (for simplicity, we illustrate this workflow using GitHub).

In this scenario, developers use three tools: GitHub (Table 2.1-(a),(f),(g)), the test runner (Table 2.1-(b)), and git (Table 2.1-(c),(d),(e)). They also performed four overlapping subtasks: *running* the tests (Table 2.1-(b)), *linking* the commit (Table 2.1-(a),(c)), *managing* version control (Table 2.1-(c),(d),(e)), and *configuring* the code for review (Table 2.1-(f),(g)). In addition, they relied on several pieces

Table 2.1: Manual steps for the common 'share changes' workflow.

(a) Open a web browser for the issue tracker and check the issue number for the current work item.
(b) Open a terminal and run the tests against the changed code to ensure they work (e.g., `npm run tests`).
(c) Open a terminal and commit the code, tagging it with the current work item number (e.g., `git commit -m 'See issue #1223'`).
(d) Pull any external changes from the remote repository (e.g., `git pull`).
(e) Push the local change to the remote repository (e.g., `git push`).
(f) Open the commit in the version control system using the GitHub web interface and open a pull request.
(g) Determine a set of reviewers and assign them to the pull request with the GitHub web interface.

Table 2.2: Literal CDA steps for the 'share changes' workflow.

(a) "CDA, Create a branch 'issue1223' in the FrontEnd repo."
→ "Branch created."
(b) "CDA, Run all the tests in the FrontEnd repo."
→ "Tests executing."
(c) "CDA, Commit with 'Fix #1223' in the FrontEnd repo."
→ "Commit made."
(d) "CDA, Pull the FrontEnd repo."
→ "Pulled."
(e) "CDA, Push the FrontEnd repo."
→ "Pushed."
(f) "CDA, Open GitHub for the FrontEnd repo and create a pull request for branch issue1223."
→ "Pull request created."
(g) "CDA, Open GitHub for the FrontEnd repo and add alice79 as a reviewer for the issue1223 pull request."
→ "Reviewer added."

of implicit contextual knowledge: (1) the repository being used, (2) the current is-

sue, (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 2.2.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 2.1-(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 2.1-(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.

Chapter 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 3.1.

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:

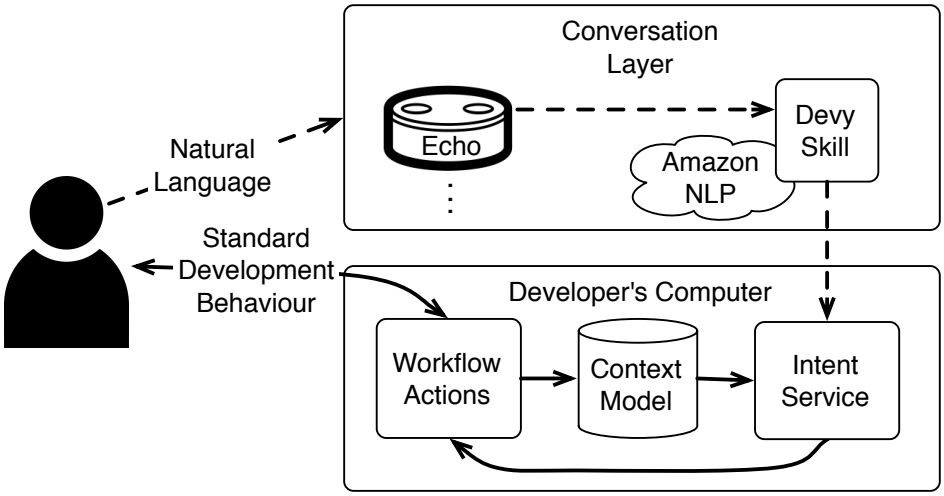


Figure 3.1: Devy's architecture. A developer expresses their intention in natural language via the conversational layer. The intent service translates high-level language tokens into low-level concrete workflows which can then be automatically executed for the developer. Dotted edges predominantly communicate in the direction of the arrow, but can have back edges in case clarification is needed from the user.

"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 Chapter 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 3.1. 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 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

Table 3.1: 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`

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 devel-

opers 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.

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 3.2. 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 3.2; 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.

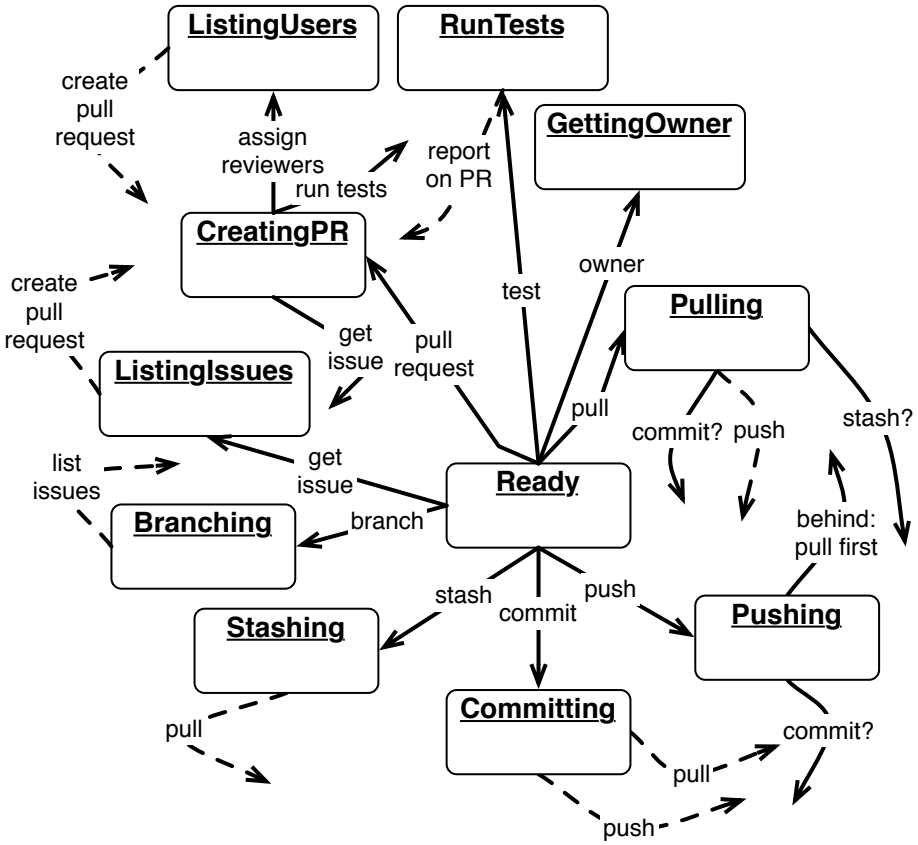


Figure 3.2: Devy's finite state machine for handling workflows. Stack-push transitions are shown with solid lines while stack-pop transitions are shown with dotted lines. For readability, some arrows do not connect with their state. However, all lines are labelled with the action that causes the state transition and correspond to the next state. Edges between the FSM and the Context Model are elided for clarity.

Chapter 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 Chapter 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 (6) or senior developers (15) 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 4.1. 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 4.1 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 work-

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

flows that a conversational assistant such as Alexa could help them with in their workplaces.

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.

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 4.1) 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 4.1 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 4.1 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 [14]. 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 thesis, 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...".

Table 4.1: 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?

Chapter 5

Results

In this chapter 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.

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. We guided participants through the subtasks confirming successful attempts and prompting them to retry failed attempts. Successful attempts, shown in Table B.2, were intuitive to participants since Devy's responses directly answered or confirmed the requested action. If a subtask failed for technical reasons (e.g. lost WiFi connection) Alexa responded "The requested skill took too long to respond", or if the user

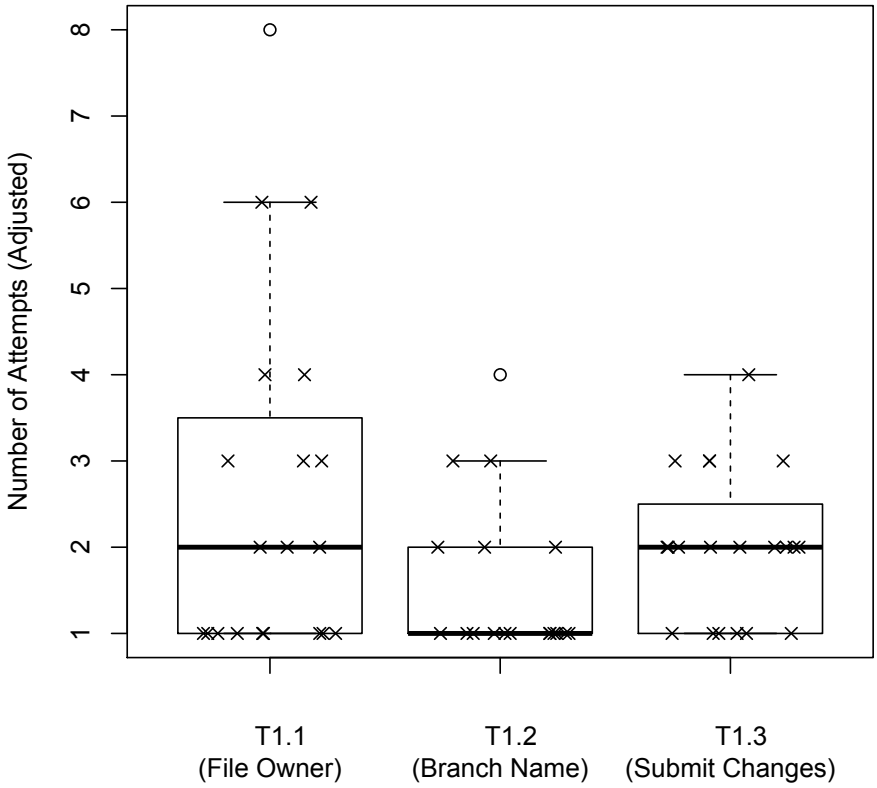


Figure 5.1: Adjusted number of attempts required to complete each task of T1 across 20 participants.

forgot to invoke Devy, Alexa responded "Sorry, I'm not sure". Due to limitations with Alexa, Devy would sometimes misinterpret a user's request. In these cases we would ask the participant to try again, although it was usually clear from Devy's response that the task was not successful (e.g. "You are on branch iss2" in response to "Alexa, ask Devy who owns this file").

Figure 5.1 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 Chapter 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 1 *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 se-*

quence 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 2 *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 com-*

mands 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 3 *Participants raised several concerns for conversational developer assistants related to disruptiveness of voice interactions, the need for transparency, completeness, and customization.*

5.4 Summary

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.

Chapter 6

Related Work

We build upon a diverse set of related work in this thesis. 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, including details 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 also includes details 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 [15]. The similarity between task contexts can also be used to help identify related tasks [8]. 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" [13]. 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 [12]). 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 [16]. 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 [9]. 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.

Chapter 7

Discussion

In this chapter 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 criticized aspect of Devy was the voice interface; all participants worked in open-plan offices and believed that conversing with Devy would annoy their co-workers. Thus, one piece of future work is to implement **alternative conversational layers** for Devy, specifically a text-based ChatBot-like interface, and to determine what feedback to give and how it should be presented to users.

Currently, Devy can be extended through the intent service by wiring up new states in the FSM. This requires at least the same amount of work as creating scripts, but 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. In order to support these more diverse workflows, more sophisticated methods of collecting, maintaining, and exposing required contextual information need to be examined. One possible approach would be to develop a framework or platform in which tools share information through a common query language, pushing the burden of supplying required information to the tool provider. This could be combined with a library of atoms for accessing and processing the information and for executing commands. Workflows would then simply be the composition of these atoms.

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 in-

tent 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.

Chapter 8

Conclusion

In this thesis, 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.

The Devy prototype demonstrates that developers can successfully launch complex workflows without interrupting their current tasks. 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.

Bibliography

[1] M. P. Acharya, C. Parnin, N. A. Kraft, A. Dagnino, and X. Qu. Code Drones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 785–788, 2016. → page 31

[2] A. Begel. *Spoken Language Support for Software Development*. PhD thesis, EECS Department, University of California, Berkeley, 2006. → page 32

[3] B. A. Campbell and C. Treude. Nlp2code: Code snippet content assist via natural language tasks. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 628–632, 2017. → page 32

[4] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC)*, pages 241–248, 2005. → page 31

[5] V. M. González and G. Mark. Constant, constant, multi-tasking craziness: Managing multiple working spheres. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pages 113–120, 2004. → page 1

[6] R. Huang. *RQDA: R-based Qualitative Data Analysis*, 2017. → page 18

[7] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11, 2006. → page 30

[8] W. Maalej, M. Ellmann, and R. Robbes. Using contexts similarity to predict relationships between tasks. *Journal of Systems and Software (JSS)*, 128:267 – 284, 2017. → page 31

[9] B. Z. Manaris, J. W. Pritchard, and W. D. Dominick. Developing a natural language interface for the UNIX operating system. *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 26(2):34–40, 1994. → page 32

[10] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 19–29. ACM, 2014. → page 1

[11] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering (TSE)*, 43(12):1178–1193, 2017. → page 1

[12] J. E. Sammet. Survey of formula manipulation. *Communications of the ACM (CACM)*, 9(8):555–569, 1966. → page 32

[13] M.-A. Storey and A. Zagalsky. Disrupting developer productivity one bot at a time. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 928–931, 2016. → page 31

[14] A. Strauss and J. M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998. ISBN 9780803959408. → page 18

[15] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4):25–33, 1981. → page 31

[16] A. Wachtel, J. Klamroth, and W. F. Tichy. Natural language user interface for software engineering tasks. In *Proceedings of the International Conference on Advances in Computer-Human Interactions (ACHI)*, volume 10, pages 34–39, 2017. → page 32

Appendix A

Implementation Notes

A.1 Handling Requests

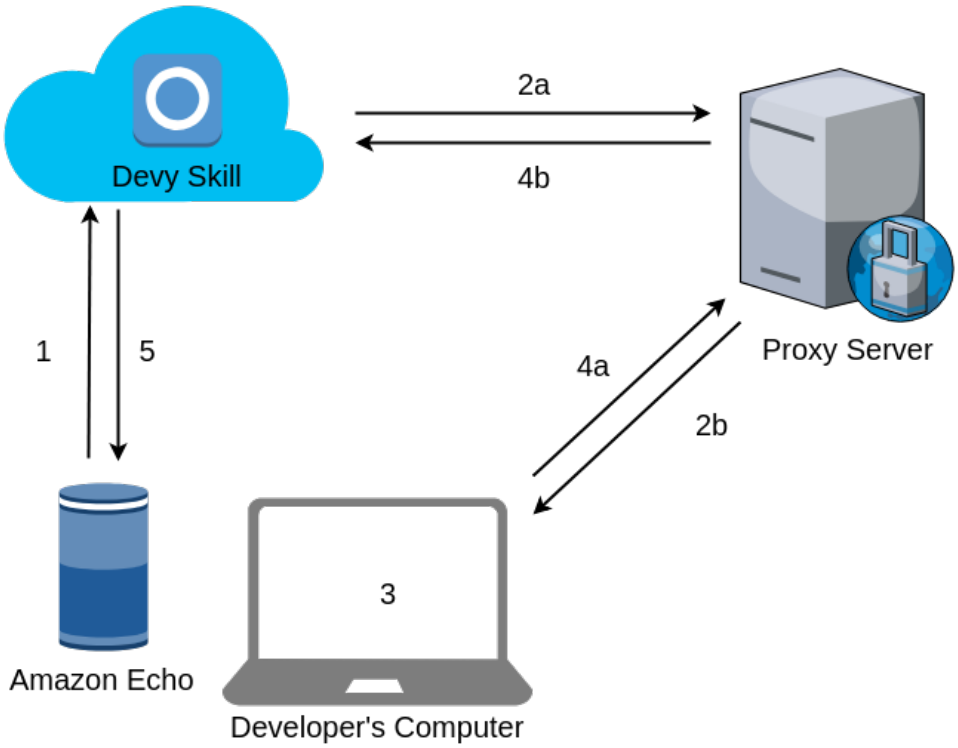
To provide a more detailed version of Devy's architecture than that of Chapter 3, we start by walking through how a user request is handled by the system (see Figure A.1).

When the developer says "Alexa, ask Devy..." to an Amazon Alexa device, the device records the speech and sends it to the Devy skill (step 1 in Figure A.1) hosted by Amazon in the cloud. The skill processes the audio and attempts to match it to one of several predefined utterances, called intents, which we created when we developed the skill. These intents represent the actions the skill can perform. During processing, any variables in the predefined utterance are resolved from the speech and a structured representation of the speech is generated including the name of the matched intent and the values for the variables. If the Alexa service is unable to determine the value to assign to a required variable, it can automatically prompt the user to specify the value explicitly. Once the structured representation is generated Alexa will send it to the configured endpoint as an HTTPS request.

Since developer machines are rarely configured with a internet accessible hostname, we created a simple proxy server and configured it to listen for requests from the Devy skill (2a). The proxy forwards requests to connected developer computers (clients) over a WebSocket connection (2b).

Once the request is received on the developer's local computer, it is further

Figure A.1: Detailed view of a request being processed by our system.



processed (3) using a finite state machine (FSM). The request transitions the FSM from the *ready* state to a state corresponding to the request's intent. The new state contains the executable code to actually perform the intention and, once complete, transitions the state machine back to the *ready* state. For example, if the user says "Alexa, ask Devy who owns this file." then the state machine transitions from *ready* to *gettingOwner* back to *ready*. The *gettingOwner* state queries the context model to get the path of *"this file"* and then calls `git blame` using the file path to see which git user changed the most lines. It then gets the developer's actual name using a GitHub WebHook with the git username found previously. Finally, it generates a response object with the phrase to be spoken to the user e.g., "{developer name} owns the file {file name} having made {n} changes in the past 30 days."

The response object is sent to the proxy which responds to the original request

(4a,b). Upon receipt, the Alexa device will speak the supplied phrase ending the interaction (5).

The user request in the preceding example only required transitioning to a single state; however, many states have preconditions. For example, if a user says "Alexa, tell Devy to push my changes." then we should check if there are any uncommitted local changes and, if so, ask the developer if they would like the changes to be committed first. In this case the state machine transitions from *ready* to *pulling* and pushes the *pulling* state onto a stack. Then the *pulling* state will transition the state machine to *committing* (which is also pushed onto the stack) which checks for uncommitted changes, and if there are any, generates a response object (including the states in the stack) and sends it to Alexa to get feedback from the user e.g., by asking "You have uncommitted changes, should I commit them?" The user answers "yes" or "no" and a new request (still containing the stack) is sent to the client which then pops the state off the stack and executes it. For the second request, the *committing* state will use the user's answer to decided whether or not to commit the changes. It will then pop off the *pulling* state (emptying the stack) and execute it. This time, it bypasses the check for uncommitted changes and simply pulls (and merges) any remote changes.

One thing to note is that the state transitions are designed such that the user cannot cancel an operation that is partially completed i.e., the stack will always be emptied before the interaction terminates. In the previous example, this means that remote changes will always be pulled after local changes are committed so the state of the repository always matches the user's expectations.

A.2 Developing Alexa Skills

Alexa is a voice-operated virtual assistant offered by Amazon which uses natural language processing to understand user intents (the actions the user would like Alexa to perform). While Alexa can handle many intents out-of-the-box, it can be extended by writing skills which are invoked by name. For example, our skill *Devy* is invoked by saying "Alexa, tell Devy...". The remaining speech is then used by Devy to determine the intent.

Skills are developed in a web-based integrated development environment hosted

by Amazon shown in Figure A.2.

Alexa recognizes intents by matching users' speech with textual example phrases supplied during development. Figure A.3 shows the utterances used to train the *startIssue* intent. Notice that the utterances are largely just permutations of each other and simply represent the variation in how the user may request the intent. Alexa is also able to handle small variations (e.g., adding a "the") automatically.

Utterances may also contain slots (i.e., variables) whose values are automatically recognized and assigned from users' speech. The example utterance in Figure A.3 contains a single slot, `IssueNumber`, that expects a numeric value. If Alexa is unable to determine the value for the variable it will remain unset; Alexa can also be configured to specifically prompt for the missing value. In our experience, Alexa is typically able to recognize the correct value if the possible values are constrained (Amazon provides variable types that only accept values from pre-populated lists e.g., the US states). However, Alexa is not able to capture free-text e.g., a commit message.

Dialog management is handled by sending state information in the request and response objects between the skill and Alexa so the Alexa service is effectively stateless.

We encountered several limitations with the Alexa service when developing our Devy skill. In general, skills

1. must receive a response from the client within ten seconds,
2. are not able to initiate interactions with users,
3. cannot accurately capture free-text, and
4. only allow training utterances to be specified during development.

The first two limitations make it difficult to handle long running commands like executing a test suite. In practice, this means that the skill is only able to notify the user that the command has started but cannot notify the user of problems encountered while running the command. Instead, it is up to the user to request the status of the command (or simply observe the execution on-screen). Amazon has partly addressed this shortcoming by providing a Notifications API but it only informs the user that they have pending notifications; they must still retrieve the notifications manually.

The third limitation makes it difficult to support intents requiring unconstrained text. In the case of commit messages, we simply use a static message "Committed by Devy" but tools exist to summarize committed changes and could be incorporated in future iterations. More generally, it would be possible to provide an on-screen text box where users could type the necessary text.

Finally, the requirement to specify all the training utterances at development-time may constrain customization since users cannot add to or alter these training phrases. The training utterances are also global to all users and the myriad utterances necessary to support the potentially large number of developer tasks would be difficult to manage and it could be difficult for Alexa to distinguish between them. A recently released alternative to traditional skills called *Skill Blueprints* may address these limitations by allowing users to customize their own instances of the Devy skill but would require more work on the part of the user.

A.3 Processing User Intents

The Devy skill runs in the cloud and converts users' speech into a JSON-structured intent object.To actually perform the intended actions, the objects are sent to a service running locally on the user's local computer where it is processed.

The service is written in Typescript for Node.js and uses four main libraries: `ws`[1], `Voxa`[2], `NodeGit`[3], and `@octokit/rest`[4] (previously named `GitHub`). `ws` is a WebSocket client and server that is used to communicate with the proxy server (which in turn communicates with the hosted Devy skill). To simplify the code needed to handle requests from the Devy skill we used the `Voxa` library which provides a way to organize a skill into a state machine using a model-view-controller pattern. We found this to be much more convenient then using the officially supported `Alexa Skills Kit SDK for Node.js`[5] which does not use a state machine. Upon receiving a request, `Voxa` automatically transitions the state machine to the associated intent state (coded in the model) and executes its callback func-

[1] https://www.npmjs.com/package/ws
[2] https://www.npmjs.com/package/voxa
[3] https://www.npmjs.com/package/nodegit
[4] https://www.npmjs.com/package/@octokit/rest
[5] https://github.com/alexa/alexa-skills-kit-sdk-for-nodejs

tion. The callback function contains the code to carry out the intent which often uses the `NodeGit` library (to execute `git` functions) and/or `@octokit/rest` (to make calls to the GitHub API). The functions can also transition to other states to check preconditions.

After the function completes, `Voxa` automatically generates a response object by calling the intent's view. Views consist of template strings for each of the possible outcomes of the intent callback function. Variables in the strings are substituted automatically with values set in the intent function and represent the text that will be spoken by the Alexa device in response to the user's request. Finally, `Voxa` generates are response object incorporating the output from the view and sends it back to the Devy skill using the `ws` WebSocket.

Figure A.2: Amazon Alexa developer console showing the Devy skill. Supported intents are shown on the left in the *Interaction Model* section. Other skill development tasks can be completed using the links on the right under the *Skill builder checklist*. Testing and deployment is completed via the tabs in the header bar.

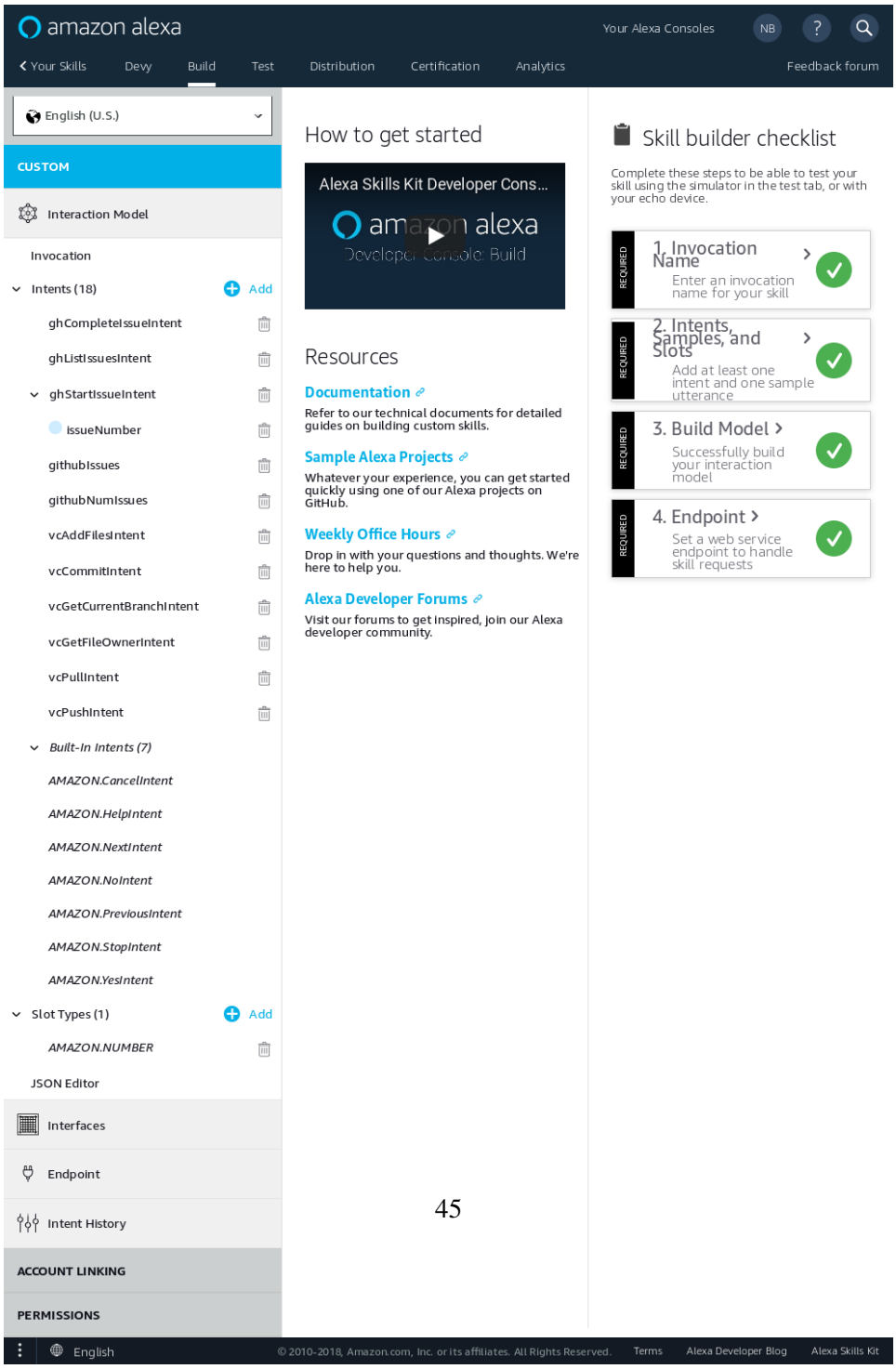
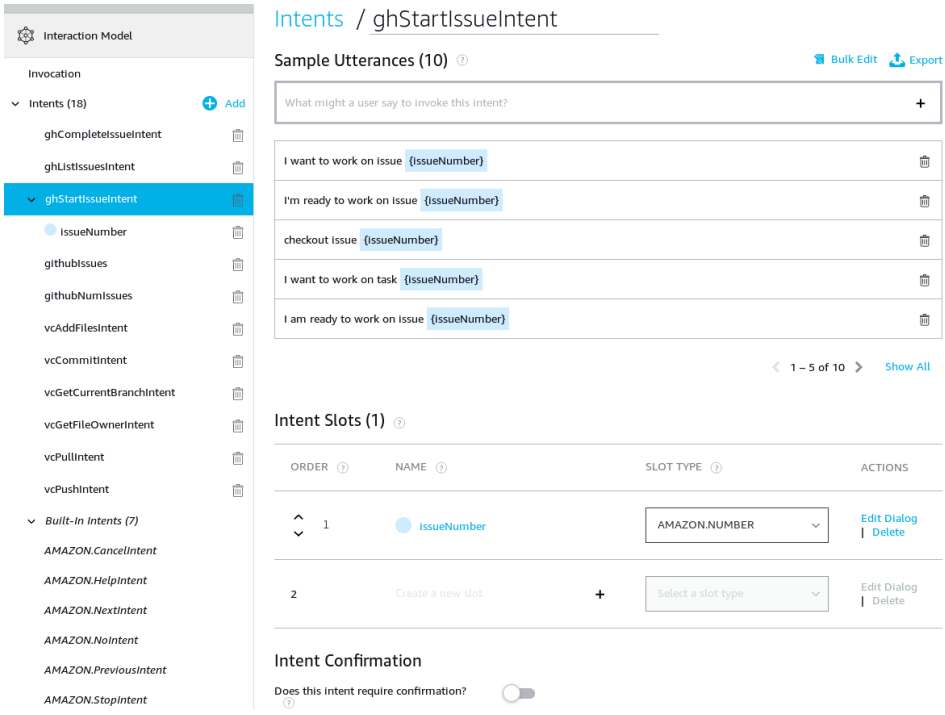


Figure A.3: Training utterances for the *startIssue* intent. Slots (variables) are shown in blue and surrounded by curly braces. The slot type and order is specified below the list of utterances.



Appendix B

Study Details

B.1 Recruiting Participants

After obtaining approval from UBC's Ethics Board, we recruited participants by reaching out to managers at local software companies who were able to recruit developers internally. We sent managers the following study advertisement via email:

The Software Practices Lab at UBC is conducting a study examining objective-based interactions with automated assistants. It will take approximately 30 minutes to complete and will consist of an interview looking at how automated assistants could be used by software developers followed by a short questionnaire. During the study, you will have the opportunity to try our prototype assistant and give suggestions for new features. A short demonstration is available at https://soapbox.wistia.com/videos/HBzPb4ulqlQI. Participants will be entered into a draw to win Amazon Echo Dots.

A short video demonstrating me asking Devy who owned an open file was included in the advertisement to help managers raise awareness and pique interest in the study. The managers also arranged a meeting space in the office where we could interact with participants privately. Due to legal concerns, managers did not want Devy to be used on computers that had access to the company's intellec-

tual property. Therefore, we had participants complete the study on a department Lenovo Thinkpad laptop running the Fedora Linux operating system. Connected to the laptop for power was an Amazon Echo Dot and a mobile phone. Before starting the study at each location, we connected the laptop and Echo Dot to the company's public WiFi network[1] for internet access. The phone was used to take audio recordings of each interview for later transcription.

At the start of each interview session we walked the participant through the study procedure and consent form (Figure B.1) and answered any questions. After obtaining consent we proceeded to begin the study.

[1] Initially, we tried to use cellular data via our mobile phone's WiFi hotspot capabilities but the connection was too slow causing the Devy skill to timeout.

Figure B.1: Consent form contents

Evaluating Automated Software Engineering Assistants

Who is conducting the study?

Principal Investigator:
Dr. Reid Holmes, Associate Professor, Department of Computer Science, UBC, rtholmes@cs.ubc.ca, 604-822-0409.

Co-Investigator:
Nick Bradley, Graduate Student, Department of Computer Science, UBC, nbrad11@cs.ubc.ca.

Who is funding the study?

This study is funded by NSERC.

Why are we doing this study?

You are being invited to participate in a study that investigates the potential applications of an automated assistant in the software development industry. The goal of this study is to develop an effective prototype automated software assistant and to show that it can generalize to support a range of tasks. Data collected will be used to evaluate the prototype and to help inform future features.

How is the study done?

Your participation in the study will involve answering a short questionnaire covering your professional background, completing some software development tasks and discussing your experience with the investigators. The study will take approximately 30 minutes to complete.

What happens next?

The aggregate results of the study will be made available through open channels and may be published in peer reviewed journals without any individual respondent or institutional identifiers.

Is there any way the study could pose a risk for you?

There are no anticipated risks for participants. You do not have to answer any questions you feel uncomfortable answering and you can end the study at any time with no repercussions. The study will only ask very limited questions about your professional background as a software developer and your experience using the automated assistant.

What are the benefits of participating in the study?

This study will provide you with an opportunity to use and provide feedback on our automated assistant. The feedback you provide will shape future versions of the assistant and will contribute to making it useful to professional software developers. Sweets will be provided during the study and you can choose to be entered into a draw to win one of two Amazon Echo Dots.

How will your privacy be maintained?

Your confidentiality will be respected. Original data collected in this study will be examined by the research team members only. Although limited identifying information will be collected, the research team will ensure that any instances of self-disclosure will be anonymized. All data will be encrypted and password protected, and held in a password protected file space accessible only to the research

team. Reports will contain descriptive statistics and include select quotes with any identifiers removed. The reports will not contain any personally identifying data. The research team will not identify individuals in publications.

Who can you contact if you have questions about this study?

If you have any questions or concerns about what we are asking of you, please contact the co-investigator. Contact information is listed at the top of the first page of this form.

Who can you contact if you have complaints or concerns about this study?

If you have any concerns or complaints about your rights as a research participant and/or your experiences while participating in this study, contact the Research Participant Complaint Line in the UBC Office of Research Ethics at 604-822-8598 or if long distance e-mail RSIL@ors.ubc.ca or call toll free 1-877-822-8598.

Taking part in this study is entirely voluntary. You have the right to refuse to participate in this study. If you decide to take part, you may choose to pull out of the study at any time without giving a reason and without any negative impact on you or your employment.

By completing this form, you are consenting to participate in this research.

I have read the Automated Software Engineering Assistants survey consent form

Name

Signature Date

If you would like to be entered into a draw to win one of two Echo Dots, please fill in your email address.

Email

B.2 Conducting the Study

Interviews were conducted in three phases (see also the study guide (Figure B.2)).

Before starting the study, five GitHub accounts were created: *devy-participant*, *devybot*, *a-aaronson*, *jeffmiddle*, and *frypj*. The *devy-participant* account was used by the participant. The *devybot* account was used to post automated feedback on pull requests. The other three were meant to represent the accounts of colleagues. They allowed us to create issues and files under different users and allowed the participant to assign someone to review pull requests. Next, two GitHub repositories were created under the *frypj* account[2]. The repositories were called *devy-study-1* and *devy-study-2*. Creation of the repositories was scripted (Listing B.1) to allow us to quickly and reliably reset them between trials. In particular, we deleted and re-created the repositories on GitHub and then deleted and cloned local versions of the repositories. Specifically, the script:

1. deletes the existing *devy-study-1* and *devy-study-2* repositories from GitHub,
2. re-creates the two repositories on GitHub,
3. adds *a-aaronson*, *jeffmiddle*, and *frypj* as collaborators on the *devy-study-1* and *devy-study-2* repositories,
4. if the repositories were created for the first time, the invitation request was programmically accepted by each user,
5. creates two open issues on *devy-study-2*,
6. creates base files in the repositories on GitHub: a README in *devy-study-1* and a logging class and corresponding test file in *devy-study-2*,
7. deletes the local versions of the repositories from the study laptop, and
8. clones the two repositories from GitHub,

This programmatic approach made it very easy to make changes after collecting feedback during the pilot.

Listing B.1: Ruby script to reset repositories on GitHub between participants.

```
#!/usr/bin/env ruby
# Reset GitHub Devy study repositories and update local copies
```

[2]Typically, repositories would be set up by the company and using a user different from *devy-participant* better simulates this.

```
require 'dotenv/load'
require 'github_api'
require 'fileutils'
require 'git'

OWNER = 'devy-participant'
REPOS = ['devy-study-1', 'devy-study-2']
COLLABORATORS = ['a-aaronson', 'jeffmiddle', 'frypj']

github = Github.new basic_auth: "#{OWNER}:#{ENV['GITHUB_PASSWORD']}"

# Update repos on GitHub
REPOS.each do |repo|
  puts "Deleting repository #{OWNER}/#{repo}."
  begin
    github.repos.delete OWNER, repo
  rescue Github::Error::GithubError => e
    puts "WARN - Failed to delete repository. #{e.message}"
  end

  puts "Creating remote repository #{OWNER}/#{repo}."
  github.repos.create name: repo,
    has_wiki: false,
    has_downloads: false,
    auto_init: false

  COLLABORATORS.each do |collaborator|
    puts "Adding collaborator '#{collaborator}' to #{OWNER}/#{repo}."
    github.repos.collaborators.add OWNER, repo, collaborator
  end

  # Create Issue 1 on devy-study-2
  puts "Creating issue on #{OWNER}/#{repo}."
  github.issues.create user: OWNER, repo: repo,
    title: 'HTML input tag maxLength',
    body: '',
    assignee: OWNER,
    labels: ['enhancement']

  # Create Issue 2 on devy-study-2
  puts "Creating issue on #{OWNER}/#{repo}."
  github.issues.create user: OWNER, repo: repo,
    title: 'Disable debugging output',
    body: 'comment out the debug `console.log` on line 8 of log.ts.',
    assignee: OWNER,
    labels: ['bug']
```

```
end

# Create README in devy-study-1
puts "Creating README.md on #{OWNER}/#{REPOS[0]}."
github.repos.contents.create OWNER, REPOS[0], 'README.md',
  path: 'README.md',
  message: 'Create readme file',
  content: <<~doc
    # Devy Study Task 1

    Devy is a prototype developer assistant that you can converse with to
    complete development tasks. To start a conversation, prompt Devy by
    saying "Alexa, tell Devy..." or "Alexa, ask Devy...".

    Complete the following tasks:
      1. First, simply launch Devy by saying "Alexa, launch Devy". This is
         just to make sure Devy is working (occasionally there are issues
         with wifi).
      2. Now, using Devy, try to get the name of the person whom you might
         contact to get help with making changes to this README.md file.
      3. Next, make sure you are on branch iss2 and then make a change to
         this README.md file (and save those changes).
      4. Finally, make those changes available on GitHub.

    This task will likely be frustrating to complete since we haven't
    given you the commands Devy supports. However, the purpose of the
    question is not to complete the task but for us to understand what
    developers would say to get their work done. This information will
    be invaluable for creating more useful versions of Devy in the
    future. Note also that this prototype is trained to recognize a
    limited number of commands which also contributes to making this
    a hard task.
  doc

# Create README in devy-study-2
puts "Creating README.md on #{OWNER}/#{REPOS[1]}."
github.repos.contents.create OWNER, REPOS[1], 'README.md',
  path: 'README.md',
  message: 'Create readme file',
  content: <<~doc
    # Devy Study Task 2

    Complete the following tasks.
      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.
```

```
      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.
      3. Resolve the issue: comment out the debug `console.log` on line 8
         of log.ts by prepending it with `//`. Save the file.
      4. Say "Alexa, tell Devy I'm done." followed by "Yes" and "Yes" then
         "No" and finally "Yes". This will commit your work and open a pull
         request.

    Notice that the reviewers you specified have been added. Also, notice
    that tests covering the changes were automatically run and the results
    included as a comment by Devy.
  doc

# Create log.ts in devy-study-2
puts "Creating log.ts on #{OWNER}/#{REPOS[1]}."
github.repos.contents.create OWNER, REPOS[1], 'log.ts',
  path: 'log.ts',
  message: 'Initial version of logging',
  content: <<~doc
    /**
     * Collection of logging methods. Useful for making the output easier to
     * read and understand.
     *
     * @param msg
     */
    export default class Log {
      public static debug(msg: string) {
        console.log("<D>: " + msg);
      }
      public static trace(msg: string) {
        console.log("<T>: " + msg);
      }

      public static info(msg: string) {
        console.log("<I>: " + msg);
      }

      public static warn(msg: string) {
        console.error("<W>: " + msg);
      }

      public static error(msg: string) {
        console.error("<E>: " + msg);
      }
    }
  doc
```

```
# Create test/log.spec.ts in devy-study-2
puts "Creating test/log.spec.ts on #{OWNER}/#{REPOS[1]}."
github.repos.contents.create OWNER, REPOS[1], 'test/log.spec.ts',
  path: 'test/log.spec.ts',
  message: 'Create test suite for logging',
  content: <<~doc
    const sinon  = require('sinon');
    const assert = require('assert');
    import Log from "../log.ts";

    describe("log", () => {
      const spy = sinon.spy(console, 'log');
      const msg = "log message";

      after(() => {
        spy.restore();
      })

      it("should log debug messages", () => {
        Log.debug(msg);
        assert(spy.calledWith(`<D>: ${msg}`));
      });

      it("should log trace messages", () => {
        Log.trace(msg);
        assert(spy.calledWith(`<T>: ${msg}`));
      });

      it("should log information messages", () => {
        Log.info(msg);
        assert(spy.calledWith(`<I>: ${msg}`));
      });

      it("should log warning messages", () => {
        Log.warn(msg);
        assert(spy.calledWith(`<W>: ${msg}`));
      });

      it("should log error messages", () => {
        Log.error(msg);
        assert(spy.calledWith(`<E>: ${msg}`));
      });
    });
  doc


# Update local copy of each repo
```

```
REPOS.each do |repo|
  path = "/home/#{ENV['USER']}/#{repo}"
  puts "Deleting local repository #{path}."
  begin
    FileUtils.rm_rf(path)
  rescue
    puts "WARN - Failed to delete local repo directory #{repo}."
  end

  puts "Cloning #{OWNER}/#{repo} to #{path}."
  g = Git.clone("git@github.com:#{OWNER}/#{repo}.git", path)

  if repo == 'devy-study-1'
    puts "Checking out new branch iss2."
    g.branch('iss2').checkout
  end
end
```

In the first phase, we asked participants questions designed to help us better understand their workflows and what thought(s) prompted, initiated, or preceded each workflow. For instance, a participant might describe their workflow for debugging a failed test and we would then prompt them to tell us what they thought before starting the workflow e.g., "why did this break?". These questions are important in helping us design a richer context model to support a wider assortment of workflows. It also helped us better understand the statements developers would use to initiate a specific workflow. With this information, we can provide more training utterances to the Devy skill and thus offer more natural interactions for the user. We concluded the first phase by introducing the Echo Dot and providing the participant with a list of Devy commands for reference. To get the participant more comfortable with interacting with the Echo before the next phase, we had them ask "Alexa, tell me a joke".

The second phase was an interactive session where the participants completed a series of common developer tasks using Devy. The tasks were meant to be familiar to the participants and included version control, testing, and working with issues. We gave the participants instructions orally, and textually via issues and in the files they were to modify. To start, participants were asked to read the README.md file that was open in Visual Studio Code which was opened on the study laptop and used throughout the study. The README is shown in Listing B.2.

Listing B.2: Instructions for completing the first task.

```
# Devy Study Task 1

Devy is a prototype developer assistant that you can converse with to
complete development tasks. To start a conversation, prompt Devy by
saying "Alexa, tell Devy..." or "Alexa, ask Devy...".

Complete the following tasks:
  1. First, simply launch Devy by saying "Alexa, launch Devy". This is
     just to make sure Devy is working (occasionally there are issues
     with wifi).
  2. Now, using Devy, try to get the name of the person whom you might
     contact to get help with making changes to this README.md file.
  3. Next, make sure you are on branch iss2 and then make a change to
     this README.md file (and save those changes).
  4. Finally, make those changes available on GitHub.

This task will likely be frustrating to complete since we haven't
given you the commands Devy supports. However, the purpose of the
question is not to complete the task but for us to understand what
developers would say to get their work done. This information will
be invaluable for creating more useful versions of Devy in the
future. Note also that this prototype is trained to recognize a
limited number of commands which also contributes to making this
a hard task.
```

The next task was designed to show participants the power of Devy and to get them thinking further about what features they could suggest. This time, we had the participants open the README.md file in the devy-study-2 repository shown in Listing B.3.

Listing B.3: Instructions for completing the second task.

```
# Devy Study Task 2

Complete the following tasks.
  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.
  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.
  3. Resolve the issue: comment out the debug `console.log` on line 8
     of log.ts by prepending it with `//`. Save the file.
  4. Say "Alexa, tell Devy I'm done." followed by "Yes" and "Yes" then
     "No" and finally "Yes". This will commit your work and open a pull
```

```
request.

Notice that the reviewers you specified have been added. Also, notice
that tests covering the changes were automatically run and the results
included as a comment by Devy.
```

In the next phase, we gathered qualitative feedback from the participate about their experience with Devy by asking the following opened-ended questions:

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 the 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?

We concluded the study asking participants demographic questions and thanking them for their participation.

B.3 Data Analysis

In this section, we describe in more detail how we transcribed and coded the study interviews and how we analyzed the data used to generate Figure 5.1. Anonymized participant data including their experience and job title summarized in Section 4.1 and their interview duration summarized in Section 4.3 are given in Table B.1.

B.3.1 Quantifying Number of Invocation Attempts

To get a sense of Devy's usability we had participants complete some common development tasks without having received prior instruction. We recorded the number of invocations required for the participant to successfully complete the task, removing invocations that failed due to technical reasons, most notably, the network connection dropping out and missed steps during the experiment set up. However, we still count incorrect invocations e.g., missing the skill name in the invocation: "Alexa, get the current branch". A task is considered completed once Devy responded with the phrase in Table B.2, regardless of the invocation. Across all tasks we removed 55 out of 175 invocations due to technical reasons. Invocation counts for each participant, by task, are shown in Table B.3.

B.3.2 Open Coding Interview Transcripts

I manually transcribed the audio recordings captured during the interview using VLC media player to control the audio playback and Visual Studio Code to write the text. The raw transcripts were written in markdown and the interviewer text was marked by a preceding (N)ick or (T)homas, the participant's text was marked with a (P)articipant, and Alexa's responses were marked with an (A)lexa. Durations were recorded in the heading of each section of the interview to facilitate linking the transcribed text to the audio recording. Formatted versions of the transcripts are provided in Appendix **??**.

Transcribing the 12 hours of recordings took approximately 22 hours over the course of about one week.

Using the transcripts we open coded participants' responses to the questions outlined in Part III of the study guide (Figure B.2). Thomas Fritz (second author) and I developed the codes by independently coding five randomly selected

Recording Duration	Programming Exp. (Yrs)	Professional Exp. (Yrs)	Job Title	Classification
00:34:50	7.5	6	Senior developer	Senior
00:34:03	-	15	Lead member of technical staff	Senior
00:46:26	5	1	Software developer	Junior
00:36:11	27	20	Architect	Senior
00:29:42	-	20	Principal member of technical staff	Senior
00:31:53	25	17	Senior developer	Senior
00:39:25	2	1.5	Software developer	Junior
00:28:46	20	20	Web developer	Senior
00:31:26	1	1	Frontend developer	Junior
00:35:06	20	15	Software developer	Senior
00:34:19	35	20	Programmer	Senior
00:35:55	7.5	1.5	Programmer	Junior
00:33:31	35	18	Architect	Senior
00:30:33	9	9	Principal software engineer	Senior
00:33:57	7	2	Software developer	Junior
00:33:48	20	20	Senior developer	Senior
00:29:17	8.5	8.5	Systems engineer	Senior
00:42:02	15	9	Software engineer	Senior
00:33:22	25	12	Software developer	Senior
00:33:45	6.5	0.5	Software engineer	Junior
00:33:21	22	15	Technical product architect	Senior

Table B.1: Participant and study metadata.

Table B.2: Devy's response when task is completed successfully.

Task	Response
File Owner	"A-Aaronson owns the file, readme dot md, having made 100% of the changes in the past three months."
Branch Name	"You are on branch i-s-s-2."
Submit Changes	"OK, I've pushed your changes."

Table B.3: Total and valid attempts to complete each task using Devy. Valid attempts exclude attempts that failed due to technical reasons. Data from transcript 6 was excluded because the participant completed the tasks in the wrong order.

Transcript	File Owner Task		Branch Name Task		Submit Changes Task	
	Total	Valid	Total	Valid	Total	Valid
1	4	3	2	1	1	1
2	4	2	3	1	3	2
3	6	2	0	0	4	3
4	2	1	3	1	3	3
5	4	1	2	2	5	3
6	-	-	-	-	-	-
7	9	2	0	0	3	2
8	6	6	1	1	4	2
9	1	1	3	3	2	1
10	4	1	1	1	4	1
11	10	8	3	2	1	1
12	7	4	1	1	1	1
13	1	1	1	1	2	1
14	5	4	3	2	3	2
15	1	1	1	1	1	1
16	3	1	5	4	3	2
17	3	3	1	1	2	2
18	1	1	3	3	2	2
19	11	6	1	1	3	2
20	4	3	1	1	2	1
21	1	1	1	1	3	3

transcripts and discussing and merging the resulting codes. When then validated the codes independently coding two additional randomly selected transcripts and checking agreement.

To facilitate the coding process, we used the R Qualitative Data Analysis (RQDA) package which allows tagging of text selections in plaintext documents using a graphical user interface. The text, tags, and participant metadata are also available in the R environment making them easy to use in further analyses.

Figure B.2: Study guide

Devy Study Procedure

Start

1. ☐ **Sign consent form. Check it before proceeding.**
2. ☐ **Ask if we can record them.**
3. ☐ **Let me know if you want to stop at any point.**

Part I (7 min)

1. Walk me through the typical development tasks you work on everyday. [5 min]

```
- [ ] Task management. Walk me through a change task:
  - choosing next work item
  - preparing to work on it
  - submitting the change for review (and choosing reviewers)
- [ ] Version control
  - Command line or IDE?
  - How often?
  - What information do you need to commit and how do you get it?
  - What features do you use (e.g. commit, log, status, blame)?
- [ ] Testing
  - When?
  - How long to run?
  - Always full test suite? If no, how to choose the tests?
- [ ] Debugging
  - Always with debugger?
  - How do you know where to set breakpoints?
  - How often?
- [ ] Documentation/Google/Problem solving~~
  - How do you formulate your questions?
- [ ] Collaboration
  - Who do you talk to?
  - What about?
  - By what means?
```

2. To help you get familiar with Alexa, I'll have you ask Alexa to tell us a joke. You can say: **"Alexa, tell me a joke."**

3. Can you think of any of your tasks you would like to have "magically" completed just by either talking to Alexa or by typing into a natural language command prompt?

Check Time

Part II (10 min)

Now I'm going to have you complete a couple of tasks using our automated assistant, Devy.

Committing and pushing changes

Have them open the README in `devy-study-1`. **Confirm that the correct README is open**. Tell them to read through the README and to start the tasks therein when they are ready.

> Devy is a prototype developer assistant that you can converse with to complete development tasks. To start a conversation, prompt Devy by saying "Alexa, tell Devy..." or "Alexa, ask Devy...".

> Complete the following tasks:

1. First, simply launch Devy by saying "Alexa, launch Devy". This is just to make sure Devy is working (occasionally there are issues with wifi).
2. Now, using Devy, try to get the name of the person whom you might contact to get help with making changes to this README.md file.
3. Next, make sure you are on branch iss2 and then make a change to this README.md file (and save those changes).
4. Finally, make those changes available on GitHub.

> This task will likely be frustrating to complete since we haven't given you the commands Devy supports. However, the purpose of the question is not to complete the task but for us to understand what developers would say to get their work done. This information will be invaluable for creating more useful versions of Devy in the future. Note also that this prototype is trained to recognize a limited number of commands which also contributes to making this a hard task.

Working with issues

Switch to `devy-study-2` and work through the README.

> Complete the following tasks.

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.
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.
3. Resolve the issue: comment out the debug console.log on line 8 of log.ts by prepending it with //. Save the file.
4. Say "Alexa, tell Devy I'm done." to commit your work and open a pull request. Devy will ask if you what to add the new file; say "Yes". Next, Devy recommend up to 3 reviewers. You choose any you like. When completed, Devy will say it created the pull request and will open a new tab showing the pull request. Notice that the reviewers you specified have been added. Also, notice that tests covering the changes were automatically run and the results included as a comment by Devy.

Check Time

Part III (10 min)

1. Now you've seen a few of the initial capabilities of Devy and we are currently working on more. So, imagine that Devy could help you with anything you'd want, what do you think it could help you with and where would it provide most benefit? ...Can you think of anything else you'd have it do based on the development tasks you mentioned at the start?

2. If we left Devy with you, would you try it on your own? Why or why not. Would you be interested in having one?
3. 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?
4. When you think about the interaction you just had with Devy, what did you like and what do you think could be improved.
5. Did Devy do what you expected during your interaction? What would you change?
6. Do you think that Devy adds value and why or why not?

Check Time

Part IV (3 min)

1. How many years have you been programming?
2. How long have you been working as a professional software developer?
3. What is your current job title?
4. What IDE or text editor do you use?
5. What tools do you use to collaborate with your colleagues?
6. Before this study, had you used voice-operated assistants like Alexa?

Conclusion

Thank you so much for participating. We will let you know about the results of the lottery in the next couple of weeks.

If you have any further ideas or comments on Devy, please let us know, we really appreciate and value your feedback since we want to develop an approach that you might be able to use sometime soon.