Examining Conversational Programming Design Needs with Convo, a Voice-First Conversational Programming System Using Natural Language

by

Kevin Weng

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by .....
Harold Abelson
Class of 1922 Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Examining Conversational Programming Design Needs with Convo, a Voice-First Conversational Programming System Using Natural Language

by

Kevin Weng

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The emergence of voice-based technology and voice assistants has paved the way for new opportunities to democratize learning computational thinking and programming skills, especially for people who do not have access to the traditional programming experience, due to external circumstances or disabilities. In this thesis, I created Convo, a voice-first conversational programming system that allows users to create programs and develop programming skills simply through natural interactions and conversations with a programming agent and voice feedback. Additionally, I conducted a user study to study the effectiveness of voice-first conversational programming systems like Convo as well as receive user feedback to explore the design needs of such systems. The user study involved forty-five participants ranging from fourteen to sixty-five years old completing tasks in Convo using and comparing three different input modalities - using just voice inputs; using just text inputs; and using both voice and text inputs. The participants answered questions about their experiences with each input modality and general feedback on conversational programming systems. Results showed that participants have preference towards text and found voice-based programming the most difficult to use among the three input modalities. However many participants, especially those new to programming, saw the value and future potential of voice-based programming, especially when speech recognition becomes more accurate. Additionally, I created design recommendations based on the results for conversational programming systems, including being flexible and accessible, reducing cognitive load and improving speech recognition and natural language understanding.

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering

Acknowledgments

Thank you to the MIT App Inventor team for giving me the opportunity to work on this project with them. More specifically, I would like to thank my advisor, Hal Abelson, for giving me guidance on my project and pushing me in the right direction; my teammates, Catherine and Phoebe, for all the great work they did with Convo, the user study and our paper; Selim, for helping us on the days of the user study and helping me move laptops;

Separately, I would like to thank my supervisor and mentor, Jessica Van Brummelen. Thank you for introducing me to the project. Thank you for giving me ideas and guidance for throughout the course of developing Convo. Thank you for testing Convo and finding my bugs and leaving suggestions. Thank you for filling out the COUHES application and preparing everything we needed for the user study. Thank you so much for supporting me every step of the way.

I would like to thank Chris and Sooraj, my roommates, and Nathan, my long-time friend, for supporting me while I was on MIT campus, providing some necessary breaks during the weekends; Monica, my girlfriend, for keeping me sane at home after moving away from MIT because of the COVID-19 pandemic. Lastly, I would like to thank my family and my friends for all of their endless support this year. I would not have made it without everyone's help and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Voice-based artificial intelligence (AI) technology is becoming increasingly prevalent, with companies like Amazon, Apple, and Google developing popular voice-first devices like the Amazon Echo and the Google Home. These devices allow people to automate and streamline daily tasks, acquire information more easily or simply order food. Programmers and non-programmers alike have already begun using conversational AI through voice-first devices like Amazon Alexa to automate simple, few-turn tasks in their homes and elsewhere. Google's automated, AI-powered calling service Duplex can even call, in natural language, restaurants and businesses to book reservations and appointments on behalf of the user[18]. The increase in voice-based technology is supported by the vast improvements in speech recognition and natural language understanding in computer systems.

The emergence of voice-based and voice-first technologies and conversational AI creates new opportunities to democratize computational thinking and programming, opening the doors for more people to use their creativity to develop their own applications in the modern technology-driven world. One such opportunity is to leverage the quickly-maturing technology of conversational AI and speech recognition in developing a conversational programming system. The system can take advantage of the advances made in these fields in providing alternative pathways towards learning programming and even lower its barrier to entry.

In my thesis, I present:

1. Convo, a voice-first conversational programming system developed based on existing conversational principles.

2. A user study using Convo to explore the utility of a conversational programming system and to receive responses and feedback from participants on such a programming system.

3. Results from the user study, including responses showing that while participants preferred text-based programming over voice-based, many of them saw the future potential of voice-based programming.

4. Design recommendations based on the results and feedback of the participants for future versions of Convo and other conversational programming systems.

Chapter 2 outlines related work in the conversational programming field that this thesis hopes to build upon. Chapter 3 details the system overview of Convo, the design considerations and the existing conversational principles that went into developing the system and an example scenario of how a user might use Convo. Chapter 4 details the implementation of Convo, explaining each part of the system and how they work together. Chapter 5 details the user study's design, implementation and data collection. Chapter 6 presents the results and discussion from the user study and the design recommendations that were conceived from the results for conversational programming systems to consider. Chapter 7 is the conclusion, containing an overview of the material presented in this thesis. Lastly, chapter 8 suggests potential avenues for future work and improvements that can be made to Convo, including applying the recommendations made in Chapter 6 as well other potential paths.

Chapter 2

Related Work

The development of Convo takes inspiration from and builds upon prior work done in the fields of conversational AI and natural language and voice-based programming. The first section details the conversational principles that Convo was initially developed on and the conversational agents that inspired the development of Convo. The second section details existing natural-language-based or voice-based programming systems that are similar to Convo or have contributed to the development of Convo.

2.1 Conversational AI and Programming

2.1.1 Conversation Principles

Many conversational AI frameworks or conversational systems based their design principles on conversational principles developed by H.P. Grice, an influential linguist [14]. They are listed and summarized here:

Quantity: Be concise yet sufficient when providing information.

Quality: Be correct when providing information.

Relation: Be relatable and relevant to the current conversation.

Manner: Be clear and natural when communicating, without ambiguity and obscurity.

Companies like Google and Amazon, when developing their conversational AI frameworks for their devices, followed Grice's principles. Google's Cooperative Principle for their conversation design guidelines is one such example [13]. Throughout the development of Convo, we also strive to follow Grice's four maxims.

2.1.2 Programming Conversational Agents

Conversational agents are software programs or systems that can interpret and respond to users in natural language. Right now, the most common and well-known form of conversational agents are voice assistants like Amazon Alexa and Google Home. Voice assistants can be found everywhere and they are especially prominent as smart devices or speakers. In fact by 2019, nearly 90 million U.S. adult consumers have smart speakers, just over thirty percent of the entire U.S. adult population [17].

With the explosion of conversational AI through conversational agents, companies like Amazon and Google have taken this opportunity to create platforms to allow users to program and develop their own applications to extend the functionality of their voice assistants. Initially, users looking to develop on these platforms needed to be somewhat comfortable with programming, but hoping to lower the barrier to entry, companies like Amazon and Google started provided tools and resources to allow more people to develop on their platforms.

Amazon, for example, released Alexa Skills Blueprint that enables anyone, including non-programmers, to create Alexa Skills easily through a graphical user interface [2]. An individual simply needs to pick a blueprint and fill in the blanks from a list of supported actions (Figure 2-1). Then, they can play their Skill and even share it with others. Others like Van Brummelen have taken this approach and translated it towards other environments. For example, Van Brummelen developed tools in App Inventor, allowing users to create Alexa Skills through block-based programming and enabling more people, even elementary school students, to create complex conversational applications [30]. While neither approaches explore the idea of programming *with* conversational agents, these initiatives helped lay the groundwork for this idea.

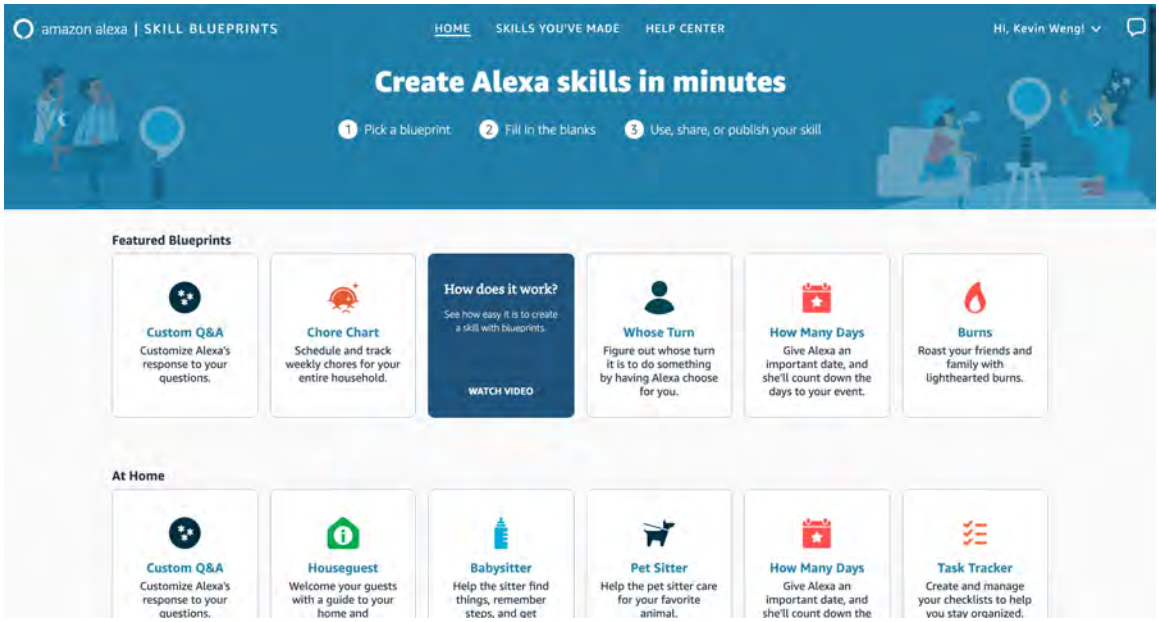


Figure 2-1: Home page of Alexa Skills Blueprint. It details the simplicity of creating Alexa Skills with just three steps. The page also lists the many blueprints that anyone can choose from and use.

2.2 Programming With Natural Language and Voice

The majority of existing voice-based programming tools and assistants require some level of programming experience or do not interpret or respond with natural language. For example, tools like Serenade, VoiceCode, and Talon allow users to program through voice inputs [15][20]. However, they were designed for users who already have knowledge on specific programming language syntax and concepts. Voice commands to these agents are also not in natural language as they require the use of technical keywords and jargon [20].

A more similar programming system to Convo in terms of using natural language is Vajra. Vajra is a programming system that allows users to program step-by-step using natural language, mapping natural language to Python code. The system was evaluated in a usability study involving both programmers and non-programmers, showing both groups being able to produce code [25]. The results reflect the feasibility of using a natural language-based programming system to teach people computational thinking and programming skills. In addition, Convo and Vajra share similar

design aspects such as an iterative workflow and the ability to describe a program using natural language [25]. However, while implemented in Python, Convo does not conform to any syntax-restrictions in Python as it is its own programming language.

Convo shares many design considerations with another conversational programming system called Codi. Created by Tina Quach of MIT Media Lab's Lifelong Kindergarten group, Codi is a voice-based programming system that allows children to build programs using natural language [22]. Just like Convo, Codi uses a agent- and voice-first programming system, but Codi focuses on developing for children with visual impairments who cannot access most programming experiences designed for beginners. Results from children using Codi led to conclusions that systems like Codi and Convo can facilitate learning from children [22]. What differentiates Convo from Codi is that Codi focuses only on audio projects and performs natural language understanding using only a semantic regex-based parser. Conversely, in addition to a semantic NLU, Convo also utilizes a ML-based NLU in parallel. The ML-based NLU allows Convo to be less restrictive on the exact phrasing of its commands and gives users more freedom and variations in their utterances.

Chapter 3

System Design

This chapter details the design considerations that went into Convo and provides an example of the experience and conversations that a user may have when programming with Convo.

3.1 Design Considerations

Several design considerations and choices went into the development of Convo to create an effective conversational programming system. As a system with a goal to teach computational thinking and programming skills in a natural language setting, we made the following design choices

1. As a programming system, Convo should be able to provide a programming experience that is similar to what users experience when using traditional programming systems.

2. Convo should be modular and extensible.

3. As a conversational system, Convo needs to be able to provide the programming experience through natural conversations, almost as natural as human-to-human conversations.

4. Because Convo is a voice-first and conversation-based system, the system needs to be able to reduce cognitive load on users when using Convo.

3.1.1 Providing a Programming Experience

As a programming system, Convo supports the traditional programming experience such as creating functions, variables, loops and conditionals. A current unsupported feature is the creation of classes and objects which is upcoming in the next version of Convo. Currently, a program in Convo is known as a "procedure", analogous to a function. In addition, Convo allows users to edit, debug and run their created programs, much like traditional programming languages like Python and JavaScript. In addition to creating variables and loops in programs, Convo additionally supports user input and adding voice-based actions like utterances and sound and music production to their programs. During execution, Convo will audibly utter what the user directed or play a desired sound or music track. Further improvements will be made to support more programming features.

3.1.2 Designing to be Modular and Extensible

Convo is implemented in such a way that other developers or users of Convo can add features and extensions easily. As a system that is in its early stage, being modular and extensible will allow the system to grow and mature quickly. In Convo, there are many areas that can be improved and extended, so eliminating any roadblocks towards these improvements and extensions will be critical.

3.1.3 Designing to be Conversational

Even though we aim to create design recommendations for conversational programming systems ourselves, when developing Convo, we still want to follow an initial set of guidelines or principles for creating a conversational programming system. Many existing conversational systems and voice user interfaces (VUIs) follow or reference H.P. Grice's four conversational maxims or rules of conversational behavior [14]. The four maxims or rules are summarized and explained in Section 2.1.1. Convo tries its best to follow these conversational design principles as well.

To be concise (from Grice's maxim of quantity), Convo tries to provide enough

information in responses such that users know enough of their current position and context while programming. In addition, if users would like more information or receive the same information again, they can ask Convo questions. To be correct and relevant (from Grice's maxims of quality and relation), Convo uses states and contexts to provide users with accurate information and a stable programming environment. To be natural (from Grice's maxim of manner), Convo formulates responses and messages to be clear and natural-sounding, removing potentially unnecessary programming jargon. For example, when creating a program, if the user provides a name for a new program that is already being used, Convo responds with "The name *name* has already been used." instead of "Procedure *name* already exists."

3.1.4 Reducing Cognitive Load

Because of the lack of visual feedback found in traditional text-based programming languages, Convo provides other ways to reduce cognitive load on users. Using words and phrases that are familiar to the user as well as removing jargon help reduce cognitive load. This ties into Convo's goal to adhere to being clear and natural when communicating (Section 3.1.3). Convo provides feedback whenever possible to try to reduce cognitive load. When creating, users receive clear but informative responses when adding actions or when encountering an error. When editing, Convo plays back each step of the program in natural, non-jargon-filled phrases, helping users remember how their programs are structured and made.

Convo also tries to be flexible in its natural language understanding in order to reduce cognitive load on the user. Convo understands certain synonyms and variations in understood commands, allowing the system to be more forgiving when users forget the exact word or phrase needed to invoke certain intents. For example, users can tell Convo to make a program by saying "create a procedure", but if users forget the verb "create" or the article "a", they can simply say the command with a synonym like "make a procedure" or "generate procedure". Thus, users can spend less time trying to remember the exact words and referencing documentation, and more time with the actual program, thereby reducing cognitive load.

3.2 User Experience

Here, we illustrate the user experience of using the current version of Convo with a girl named Lily. She is a novice programmer who wants to make a game for her little brother Eli. Eli loves animals, but because he is young, he has trouble matching animals to the sounds they make. Lisa decides to make a game that helps Eli learn his animal sounds.

3.2.1 Creating a Program

When Lily first connects to Convo, Convo introduces itself and asks Lily what she would like to do.

> **Convo:** `Hi, I'm Convo! What would you like to do? To get started, you can create a new procedure by saying "Create a new procedure". If you want to run a procedure, say "Run" and then the name of the procedure.`

Lily asks to create to procedure and Convo proceeds to direct her towards making a game for Eli. She consults the documentation for Convo for examples and commands she can say to Convo.

> **Lily:** Hi Convo, I want to create a procedure.
>
> **Convo:** `What do you want to call the procedure?`
>
> **Lily:** Animal Sounds
>
> **Convo:** `What do you want to do first in the procedure? You could make me say something. See the documentation for more options.`
>
> **Lily:** I want a loop.
>
> `Convo: I didn't quite catch that. What action did you want me to add?`

Convo still requires some form of structure for commands, so Convo does not recognize initially that Lily wanted to create a loop. Lily would go through the documentation for examples and find the correct phrase. In a future version of Convo, Lily will be able to directly ask Convo how to make a loop and other questions.

Lily: I want to make a loop.

Convo: `What's the condition?`

Lily: Until I say stop.

Convo: `What do you want to do first in the loop?`

Lily: Listen for user input and save it as animal.

Convo: `What do you want me to say when asking for input?`

Lily: Say an animal.

Convo: `Added action to the loop! Anything else? If yes, what's next? If no, say 'close loop'.`

Lily: If animal is dog, play the dog sound.

Convo: `Anything else if condition is true? You can say 'done' to move on to adding actions when the condition is false. You can say 'close' if you are entirely finished.`

Lily: Close.

Convo: `Added action to the loop! Anything else? If no, say 'close loop'.`

Lily: If animal is cat, play the cat sound.

...

Lily: Close loop.

Convo: `Do you want to do anything else in the procedure?`

Lily: Done.

Convo: `I finished creating the procedure. You can say, 'Run Animal Sounds' to play it.`

In summary, Lily is able to create a program that listens for input and if the input is an animal, it will play a sound corresponding to the animal. Convo allows Lily to add a custom prompt when listening for input. The procedure repeats these steps with a loop until Lily or someone else (like Eli) tells Convo to stop. Inside the loop, Lily adds a check for each animal; she decides to add five animals for now - dog, cat, horse, cow and bird. One important design consideration shown in this conversation was the feedback by Convo. When creating the loop and adding steps inside the loop, Convo lets Lily know what direction she can take after each input, whether that being continuing to add to the procedure or loop or stopping using the specific stop words that Convo gives Lily. Once Lily indicated she was done creating the procedure, Convo also lets her know to run her new program.

3.2.2 Running a Program

Excitedly, Lisa shows Eli her new game "Animal Sounds". Because Convo provided Lisa with the phrase that she can say to run her program, Lisa quickly starts the game. Through the execution of the procedure, Lily tells Eli to give an animal from the list of animals that she had added. Convo is able to successfully run through the procedure and stop whenever Lily says stop.

Lily: Convo, run Animal Sounds.

Convo: `Procedure started running.`

Convo: `Say an animal.`

Lily (to Eli): Eli, say an animal.

Eli: Cow!

Convo: `*Plays sound of a cow mooing*`

Convo: `Say an animal.`

Lily (to Eli): Eli, say another animal.

Eli: Bird!

Convo: `*Plays sound of a bird tweeting*`

Convo: `Say an animal.`

Eli: Horse!

Convo: `*Plays sound of a horse neighing*`

Convo: `Say an animal.`

...

Eli: Wow!

Lily: Okay Convo, stop.

Convo: `Procedure finished running.`

3.2.3 Editing a Program

Here, we illustrate an example of how a user like Lily can go back and edit their programs in Convo. Convo provides a simple interface to go through a program and edit each step.

Lily: Convo, run Animal Sounds.

Convo: `Procedure started running.`

Convo: `Say an animal.`

Eli: Cricket.

Convo: `*Does nothing*`

Convo: `Say an animal.`

Lily: Okay Convo, stop.

Convo: `Procedure finished running.`

Eli wants Convo to play a sound of a cricket but Lily did not add cricket to her "Animal Sounds" procedure. However, through Convo, she can go back and add support for a cricket sound without much difficulties.

Lily: Convo, edit Animal Sounds.

Convo: `I am on the first step of the procedure, where I am doing 6 actions in a loop until you say stop.`

Lily: Step into loop.

Convo: `I am on the first step of the loop, where I am listening for input and saving it as animal.`

Lily: Go to the last step.

Convo: `I am on the last step of the loop, where I am playing the horse sound if the value of animal is horse.`

Lily: Convo, add a step.

`Convo: What action do you want to add to the loop?`

Lily: If animal is cricket, play the cricket sound.

Convo: `Anything else if condition is true? You can say 'done' to move on to adding actions when the condition is false. You can say 'close' if you are entirely finished.`

Lily: Close.

Convo: `I am on the last step of the loop, where I am playing the cricket sound if the value of animal is cricket. What do you want to do? You can say 'done' if you are finished editing.`

Lily: Done.

Throughout the editing process, Convo makes sure Lily knows exactly where she is at and what she is doing. This is also part of Convo's design consideration to give ample feedback to the user. If Lily had to pause and go do something else, she can come back and ask Convo to tell her where she was at when editing, with questions like "Where am I?" or "Which step am I on?". Now that Lily successfully added a cricket sound to her procedure, she is able to run it again and Eli will be able to hear a cricket sound.

Lily: Convo, run Animal Sounds.

Convo: `Procedure started running.`

Convo: `Say an animal.`

Lily (to Eli): Eli, you can say cricket now.

Eli: Cricket!

Convo: `*Plays sound of a cricket chirping*`

Eli: Yay!

Lily: Okay Convo, stop.

Chapter 4

Technical Implementation

This chapter details the technical implementation of all of Convo from Convo VUI to different parts of Convo Core.

4.1 Overview

On a high level, Convo can be separated into main components, Convo VUI (Voice User Interface) and Convo Core (see Figure 4-1). Convo Core can further be separated into three components - the natural language understanding (NLU) module, the dialog manager, and the program manager. Convo VUI lives on a Node JS server. Most of Convo Core lives on a Flask Python server. The NLU module contains two NLUs, a semantic regex-based NLU and an ML-based Rasa NLU. The Rasa NLU currently requires a separate Python server. Each component plays an integral part in allowing Convo to provide a stable conversational programming experience for the user.

4.1.1 Ways to Communicate with Convo

While Convo is a voice-first conversational programming system, the system allows users to interact with Convo using both voice and text inputs. With voice input, users interact with Convo VUI which in turn interacts with Convo Core (Figure 4-2). If users want to interact with Convo using text, they directly communicate with Convo

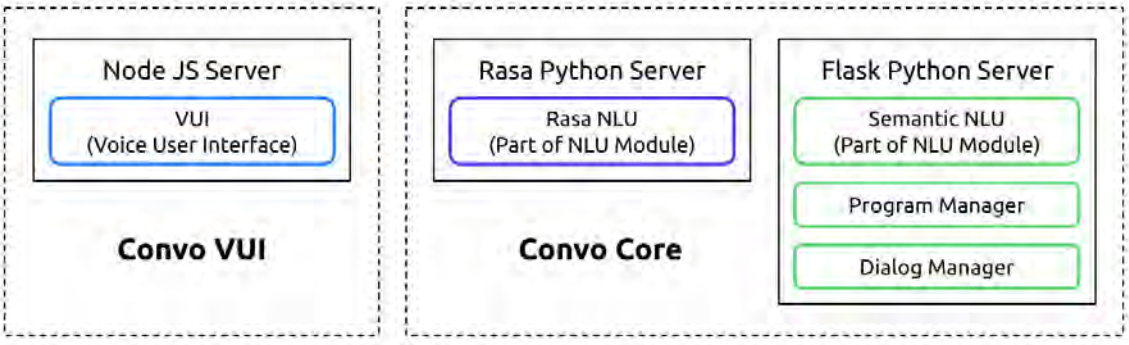


Figure 4-1: A diagram showing the servers and components that make up Convo. In total, there are three servers and four main components.

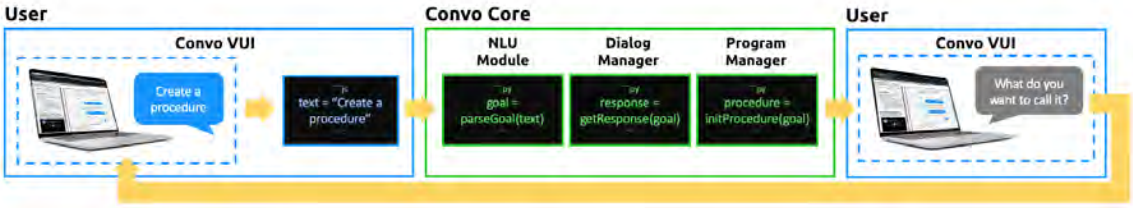


Figure 4-2: The typical path that a voice-based input from a user takes through Convo. The input is sent to Convo VUI first which is then sent to Convo Core where it processes the input and responds back to the user through Convo VUI.

Core. This provides an experience similar to the experience of interacting with a typical chatbot.

4.1.2 WebSocket Connections

Instead of the standard HTTP connections, WebSocket connections are used for the communications between the user and Convo and between Convo VUI and Core. WebSockets play an integral part in running Convo. Unlike HTTP connections, a WebSocket connection is a permanent, bi-directional communication channel between a client and the server, where either one can initiate an exchange [10]. WebSockets allow Convo to asynchronously send messages to users without needing the user to send a message first. In addition, WebSocket connections can be used to stream audio or byte data as in the case of Convo VUI (Section 4.2)).

A WebSocket connection is defined as permanent because once established, the connection remains available unless either the client or server disconnects from the

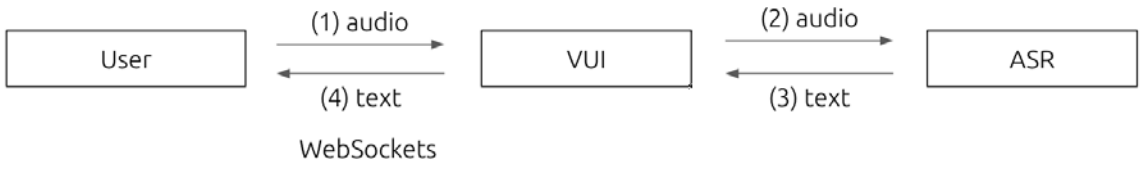


Figure 4-3: A diagram shows how the audio input from the user is streamed to the VUI, transcribed by the ASR service and finally sent back to the user as text. This text is sent to the dialog manager (not shown) which will respond back audibly to the user.

other. This helps Convo track which users are online and using Convo and which users disconnected. When a user connects, their procedures are retrieved from the database and stored in memory. Knowing when the user disconnects allows Convo to free up resources.

Instead of defining routes that allow users to send HTTP API requests, Convo defines WebSocket event handlers that allow users to send WebSocket events. For example, Convo has an event handler `message` to receive messages from clients. The event handlers function similarly to HTTP routes but the protocol further enables the client to have event handlers as well. Therefore, Convo can send messages to the user using the client-defined events. We defined a client event `response` to allow users to receive responses and messages from Convo.

4.2 Voice User Interface

The voice user interace (VUI) is the typical interface between Convo and its users; it is what users interact with and speak to. The main functions of the VUI are to receive and transcribe audio input to text and to respond and utter back an appropriate voice-based response determined by the dialog manager. Responses range from answering user questions to asking for further input from the user.

Convo requires the user's utterance to be transcribed into text before it can be further processed and used by the system. This is achieved using automatic speech recognition (ASR) technology. Figure 4-3 shows how a user's utterance is processed by the VUI from speech to text. The audio input is first streamed from the user

(typically through the browser) to the VUI via WebSockets. The VUI processes the audio data and sends it to the ASR. The ASR returns the transcribed text back to the VUI where it may undergo further post-processing before it is sent back to the user. The user will then send the transcribed text to Convo's Core where the dialog manager is waiting to process the input.

To transcribe speech to text as accurately as possible, Convo's ASR is handled by Google's Cloud Speech-to-Text API, as Google's language models have achieved some of the lowest word error rates (WER) [8][12]. Word error rate is a common metric of the performance of a ASR system with a low WER signifying an accurate transcription [32]. Once the utterance is transcribed, the VUI sends it to Convo Core, specifically the dialog manager, where the message is processed. After the dialog manager sends back a response, the VUI takes the response and presents it as audible speech back to users. Responses are voiced back to users using Google's Speech Synthesis API [11].

Even with Google's Speech-To-Text API, mis-recognition of words still occur, especially with accents or small voices. One can improve the accuracy of the transcription results Convo gets from Cloud Speech-to-Text API by using speech adaptation, a feature provided by the API. Speech adaptation allows the API to recognize specific words or phrases more frequently and to improve the accuracy of words and phrases that occur frequently. In Convo's case, words and phrases related to programming like "variable" or "loop" are subjected to speech adaptation. Even with speech adaptation, certain words are still frequently mistranscribed. For example, "done" is frequently mistranscribed to "dumb". In certain situations and based on the context of the phrase and the conversation, these pairs of words are regarded as "synonyms". The VUI will replace the mistranscribed word with the intended word.

The VUI lives on a Node server, separate from Convo Core living on a Python server. The decoupling allows users to interact with Convo through text-based utterances in situations where voice-based utterances cannot be used. For example, because Convo relies on an external ASR service, if for any reason the service is not available, Convo will still be able to function and reliably process user utterances

through text inputs. This also allows us to conduct the user studies detailed in Chapter 5.

4.3 Natural Language Understanding

Natural language understanding (NLU) allows Convo to interpret and comprehend the meaning behind what the user says. NLU systems typically achieve this by reducing text or transcribed speech into a structured ontology. Statistical or machine learning methods and models are then applied to this ontology to extract relevant information like meaning, intent, sentiment and context.

Convo uses natural language understanding to extract useful information and to recognize user intent from the transcribed message from the user provided by the VUI. The system's NLU is able to differentiate between syntactically similar yet semantically different phrases or commands like *"create a variable"* and *"create a procedure"* while recognizing semantically similar phrases or commands like *"create a variable"* and *"make a variable"*. Extracted information from NLU can include user intent and associated arguments. As an example of intent recognition, if Convo receives the utterance *"create a variable called foo"*, it is able to extract the intent to create a variable and the additional information that the variable should be named "foo". While the main responsibility of the NLU is intent recognition, the NLU is also able to capture other forms of information from user inputs including value placeholders and conditions that are used in procedures as components (see Section 4.5.1).

4.3.1 Approaches

Convo performs natural language understanding using a combination of a semantic-based approach and a machine-learning (ML)-based approach. While Convo's NLU support capturing other forms of information, the discussion about the two NLU approaches will primarily be focused around recognizing intents. When a user utterance is received, the semantic-based approach is utilized first. Convo uses the ML-based

approach only if the system is not able to extract an intent or usable information with the former approach. The extracted semantic information is provided to the dialog manager in the form of a goal (see Section 4.4.3). The dialog manager receives and processes the goal, performs actions based on the information given and returns an appropriate response to the user back through the VUI.

Semantic Regex-based NLU

The semantic-based method of natural language understanding uses regular expressions (regex) to parse user utterances that adhere to specific string patterns [29]. There is a regex pattern defined for every supported user intent in Convo. For example, the "create a variable" intent uses the following regex pattern.

```
(?:create|make)(?: a)?(?: (.+))? variable(?: called| named)?
(?:(?: (.+))? and set(?: it)? to (.+)| (.+))?
```

The pattern allows Convo to match different variations of utterances that still capture a user's intent to create a variable, ranging from simple phrases like "make variable" to more complex utterances containing more information like "create a variable called foo and set it to 5". In the complex utterance, the user provides the name and the initial value of the newly created variable as arguments. Parentheses in the pattern (like `(.+)`) capture the text matched by the regex inside them into a numbered group that can be referenced and retrieved later. This allows Convo to extract additional arguments that some intents require.

While the semantic regex-based approach to NLU allows for quick recognition and extraction of information, it does not scale well if we want Convo to recognize more utterances corresponding to the same intent. This would require either trying to use multiple regexes for the same intent or create a single unreasonably long pattern that can be hard to decipher. In addition, misspellings of words or slight variations in words may lead to mismatches or even no matches. This constrains Convo, for users need to say specific phrases to trigger certain intents. The ML-based approach to NLU helps loosen this constraint on the system.

Rasa NLU

Convo's ML-based approach to natural language understanding is through the integration and use of Rasa. Rasa is an open-source conversational AI framework for building context assistants and conversational software [5]. Rasa consists of Rasa Core, a dialog manager, and Rasa NLU, Rasa's open-source natural language understanding framework. Convo integrates the Rasa NLU as part of its own NLU along with its semantic regex-based NLU. Although other NLU solutions similar to Rasa are available, Rasa was chosen because of its ability to be highly configurable. In addition, Rasa NLU's performance compares favourably to various closed-source solutions [6].

Rasa NLU trains a language model for intent recognition and entity extraction on training data provided in Markdown format. For example, training examples for the "create a variable" intent look like

```
## intent:create_variable
- create a variable
- can you create a variable
- make a varible
- create a variable
- i want to make a variable
- make a variable
```

Rasa NLU allows users to customize the pipeline that is used for training, allowing one to change different components like pre-trained embedding sources, tokenizers, featurizers and intent classifiers. Typically, a Rasa NLU pipeline consists of three main parts: tokenization, featurization and intent classification.

Convo uses the following NLU components:

1. `HFTransformersNLP`: The `HFTransformersNLP` component allows the use of pre-trained language models gathered from HuggingFace's Transformers Python library [33]. In machine learning, pre-trained models are models developed and trained with large amounts of data to solve a particular problem. For a similar

problem, instead of starting from scratch, using the pre-trained model as a starting point in a process known as transfer learning can improve accuracy and training time immensely. The component allows Rasa to leverage existing state-of-the-art pre-trained language models like BERT, GPT-2 or XLNet for use in training and relevant downstream NLP tasks like intent classification [9][23][34].

In its NLU, Convo utilizes the BERT model for its task of recognizing user intent. Released by Google in 2018, BERT was the first language model to apply the bidirectional training used in the popular attention model Transformer [31] its own training. This led to BERT obtaining state-of-the-art results on various NLP tasks including question answering and language inference. Training and using a model through transfer learning from the pre-trained BERT model allows Convo to achieve the best possible results for intent recognition.

2. `LanguageModelTokenizer`: The tokenizer splits input text and creates tokens using the pre-trained BERT model specified in the `HFTransformersNLP` component.

3. `LanguageModelFeaturizer`: The featurizer uses Convo's pre-trained BERT model to extract similar contextual vector representations for the input text.

4. `LexicalSyntacticFeaturizer`: The featurizer creates features for entity extraction by moving a sliding window over the tokens created by the tokenizer and extracts lexical and syntactic features.

5. `CountVectorsFeaturizer`: The featurizer creates bag-of-words representations of input text and intent for features using scikit-learn's `CountVectorizer` [21].

6. `DIETClassifier`: The Dual Intent and Entity Transformer (DIET) classifier is used to classify the intents that Convo supports. It uses a sequence model that takes word order into account, which offers better performance than Rasa's former bag-of-words model. According to Rasa, DIET parallels large-scale pre-

trained language models in accuracy and performance, improves upon current state of the art, is six times faster to train [19].

Rasa NLU requires a separate running Python server to use a language model that it has trained. The Rasa server is run locally in the same environment as the Convo Core server. Once running, Convo can request intent recognition predictions from the trained model through HTTP POST requests using the `/model/parse` endpoint.

4.4 Dialog Manager

The dialog manager handles the responsibility of receiving user messages from the VUI and sending it to the NLU to get user intent and the associated goal. The dialog manager also works closely with the program manager to provide users with a stable environment. When a user starts using Convo, a new dialog manager is created and assigned to them. It handles

- processing of user goals from the VUI
- conversation and context tracking
- question answering
- interactions with the program manager

4.4.1 Dialog Context

The dialog context is the brain of the dialog manager, maintaining and controlling user- and system-level information needed for dialog management. The dialog context contains information on

- **dialog state**: The dialog context stores the dialog state used by the state machine (Section 4.4.2).
- **goals in progress**: Any goals in progress can be found in the dialog context.

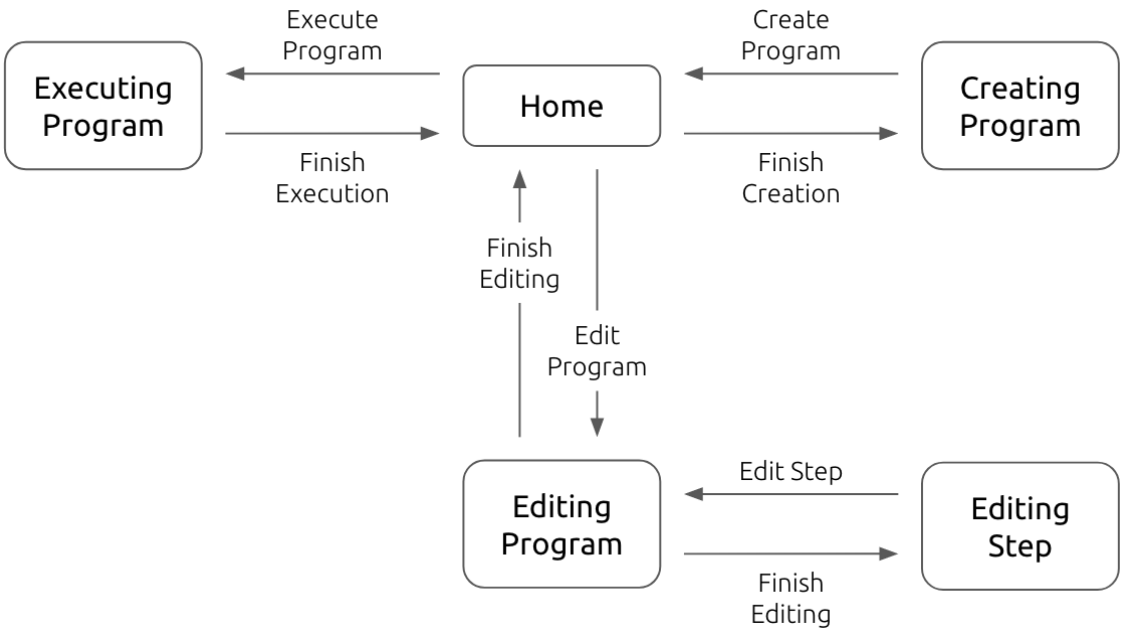


Figure 4-4: The diagram depicts the state machine in the dialog manager and the transitions between the states. Users first start in the Home state and transition to the other shown states with different intents.

- **editing contexts**: (Section 4.5.2) and whether a program is currently executing (Section 4.5.3). The dialog manager can check if a program is being edited or executed through its dialog context.

- **all of the user's procedures**: All procedures retrieved by the program manager from the database can be found in the dialog context.

- **conversation history**: All user messages and Convo responses are stored in the context.

4.4.2 State Machine

The dialog manager uses a state machine as part of its processing workflow and stores a dialog state in the dialog context. A state machine is a system that has a set of states and defines transitions between those states and allowed actions within those states. Together, the state machine and the dialog state allow Convo to direct conversations and prevent potential unintended system behaviors by restricting certain intents to

certain states. Certain intents can transition the dialog state to another state. The state machine also prevents invalid transitions between states. We do not want users trying to delete a program while still editing the program or creating a new program while the current program is still running.

Figure 4-4 shows the state machine used in the dialog manager and the transitions. When using Convo, users will always start in the Home state before transitioning to one of the three other main states. When users are in each state, there will be certain intents that are not allowed by the system. The dialog manager will respond appropriately when users provide an intent that is not allowed, providing information on why it is not allowed and providing examples of intents that are allowed in the current state.

4.4.3 Goals

Goals are the main way to effect changes and perform actions within the system, and they are handled by the dialog manager. Changes occur when these goals are completed and goals help produce the responses that users receive. In general, there are two main types of goals, user goals and system goals. User goals represent a single user intent and any of its required arguments while system goals represent system actions that will be taken or changes that will be effected. Each goal can have a list of sub-goals or nested goals that each need be completed before the goal can be completed. See Table 4.1 for a list of the goals that Convo uses.

Goals can also be grouped based on how they impact or change the system and dialog context. Groupings help include or exclude user intents in different states. The goal groups consist of the following:

Action goals: These user goals, once completed, adds a corresponding action to a procedure when creating or editing the procedure (see Section 4.5.1). Examples include `CreateVariableActionGoal`, `SayActionGoal` and `GetUserInputActionGoal`. They are only allowed in the "Editing Program" or "Creating Program" state.

Goal	Description
CreateProcedureGoal	Creates a procedure.
RenameProcedureGoal	Renames a procedure.
DeleteProcedureGoal	Deletes a procedure.
EditGoal	Edits a procedure.
ExecuteGoal	Runs a procedure.
GetProcedureActionsGoal	Gets actions from user that will be added to the procedure.
GetLoopActionsGoal	Get actions from user that will be added to a loop.
GetConditionalActionsGoal	Get actions from user that will be added to a conditional.
GetInputGoal	Gets input from the user during system-level operations.
GetUserInputGoal	Gets input from the user during the execution of a procedure.
GoToStepGoal	Jumps to a step of a procedure during editing.
DeleteStepGoal	Removes a step of a procedure during editing.
AddStepGoal	Adds a step of a procedure during editing.
ChangeStepGoal	Changes a step of a procedure during editing.
ActionGoal*	Adds an action to a procedure.

Table 4.1: This lists all of the goals that are used in Convo to support system- and program-level operations. *The names of specific action goals are the same as the actions they add, suffixed with `-Goal` so only the base class `ActionGoal` is listed here. See Table 4.2 to view the list of actions currently supported in Convo.

Get-Actions goals: These system goals assist Convo in adding actions from the user. These goals have action goals as sub-goals. These goals are only completed when there are no remaining action sub-goals and the user indicated that they are done adding actions. Examples include `GetProcedureActionsGoal`, `GetConditionalActionsGoal` and `GetLoopActionsGoal`.

Home goals: These user goals are involved with program management. Examples include `CreateProcedureGoal`, `ExecuteGoal` and `EditGoal`. They are the only goals allowed in the "Home" state.

Edit goals: These user goals are involved with editing programs. Examples include `AddStepGoal`, `ChangeStepGoal` and `GoToStepGoal`. They are only allowed in the "Editing Program" state.

Input goals: These system goals are involved with getting inputs from the user. Examples include `GetInputGoal`, `GetUserInputGoal` and `GetConditionGoal`.

Arguments

Goals usually have required arguments where a value must be assigned to each of them. Each argument has its own name that can be referenced easily through the goal. Required arguments in user goals are given by the user through their intent. If an user provides an intent with none of its required arguments, Convo uses slot-filling, a common approach used in conversational interfaces [24]. Convo asks the user for the arguments one-by-one until all arguments are provided. To do this, `GetInputGoal`s (one for each argument) are added as sub-goals of the user goal. Each `GetInputGoal` is completed as each valid argument is provided.

Most arguments contain some form of validation when slot-filled. If the user provides an argument that is invalid, Convo provides feedback to the user on why the given argument is invalid. The goal will fail and be discarded. To try again, the user will have to provide the intent again. An example can be observed with the goal `ExecuteGoal`, which is created when the user wants to run a program. This particular goal requires the name of the program as an argument. If a user provides a

program name that does not match any of the existing programs associated with the user, Convo will respond back "The procedure, [program name], hasn't been created, so we can't run it".

Completing Goals

Each goal has a set of conditions that need to be met for it to be deemed complete. By default, this set of conditions includes not having any incomplete sub-goals and having no errors. For user goals, this is why all required arguments need to be provided before they are deemed complete. Once a goal is completed, it performs a series of post-completion steps before being discarded by the system. These post-completion actions are defined in each goal and they include system-level actions like transitioning the dialog state, adding a newly created program to the database and executing a program.

Some goals can already be completed the moment they are created because they had neither incomplete sub-goals nor errors. They are swiftly handled by the dialog manager. For example, if an user says "edit hello world" where "hello world" is the name of a program the user created, an `EditGoal` is created. The only argument for `EditGoal` is the name of the program which the user already provided in their original message, meaning the goal was completed the moment it was created. Therefore, once the dialog manager receives the goal, it will simply perform the post-completion steps defined in the goal. In this case, the dialog manager opens the program "hello world" for editing and transitions the dialog state to "Editing Program".

4.4.4 Handling Inputs and Goals

When the system receives an utterance or input from the user, it passes through a series of checks so it knows how to best handle the new input. The checks are performed in the following order

1. If the input messsage is "reset", it is processed by the reset handler `handle_reset`.

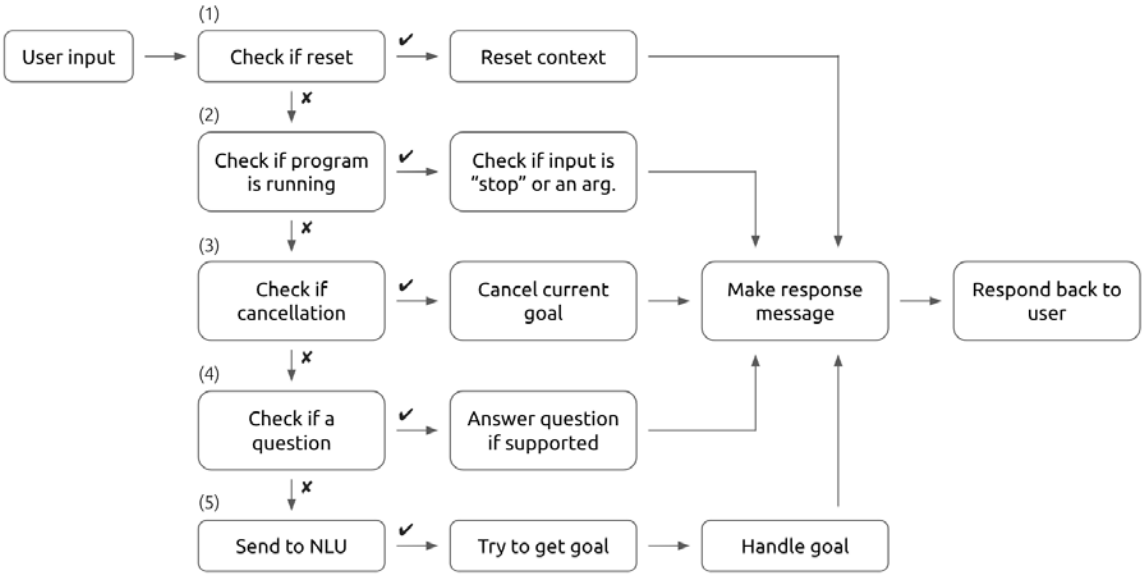


Figure 4-5: The flowchart shows how the dialog manager handles user input and what paths can the input take. The dialog manager will check, in the following order, (1) if user is asking for a reset; (2) if a program is running; (3) if user wants to cancel the current goal; or (4) if user is asking a question. If any of them is true, the input goes to the immediate right block in the flowchart shown by the corresponding checkmark. If none of checks pass, the input goes through the default path (indicated by the crossmarks), which is goal handling. The input will either be an argument for a current goal or be a new goal. All of the possible paths will end with a response back to the user.

2. If the input is received while Convo is still running a program, it is processed by the execution handler `handle_execution`.

3. If the input message relates to cancellation, it is processed by the cancellation handler `handle_cancel`.

4. If the input is a question, it is processed by the question-answering handler `handle_question`.

5. Lastly, if it is none of the above, the input is sent to the NLU for intent recognition and goal creation.

Figure 4-5 roughly shows how the dialog manager handles a new input with the checks. The series of checks are enumerated on the figure from (1) to (4), with (5) being the default step if none of the checks passed. If any of the checks pass, the input

is processed by a corresponding handler function. On the figure, this is represented as the arrow pointing to the right of the box containing the corresponding check.

Handling Resets

If for any reason, users need to restart or refresh their dialog context, they can do so by saying "reset". Resetting will cause Convo to remove any saved messages from the previous conversation and bring the dialog state back to "Home" state. If any program is running, it will be stopped. Any goals that were in progress are discarded. Any procedures that users have created will remain. This was particularly useful for some participants of the user study when they wanted to reset the current task they were trying to complete.

Handling Input While Program is Running

Users can communicate with Convo while a program is running in the background, but they are limited to what they can have Convo do. When a program is running, Convo will be in the "Executing Program" state. If an input arrives during this state, Convo checks for two possible cases

1. If the user message is "stop" or "cancel", Convo will stop the program and respond to the user with "Procedure has been stopped". This case usually occurs when the user wanted to prematurely stop the program or because the program had an infinite loop.

2. The program could be temporarily paused as it is waiting for an input from the user. Once the user provides the input, the program consumes the input and resumes execution.

If it is neither of the two cases, Convo will reject the input and respond back to the user with "Procedure is still executing." Convo will be able to receive other inputs once the program finished running and once it has transitioned back to the "Home" state.

Handling Cancellations

Users can cancel what they are currently doing by saying "cancel". When users cancel, the most immediate goal that is not a type of input goal is cancelled and removed. As an example, let's have the current goal be `Goal1`. `Goal1` has a sub-goal `Goal2`, and `Goal2` has a sub-goal `Goal3`. If `Goal3` is an input goal, `Goal2` is cancelled and removed, meaning `Goal3` is also removed. If `Goal3` is not an input system goal, only `Goal3` will be cancelled and removed. Cancelling may involve further system-level actions. For example, if the user wants to cancel creating a procedure, Convo needs to take further steps to remove the procedure and transition the dialog state from the "Creating Program" state back to the "Home" state.

Handling Question Answering

Convo has a limited question-answering system that users can use. The current module uses regex parsing to check if the user is asking a question and to check for supported questions. Currently, only a couple of specific questions can be answered by Convo.

- Users can ask about their procedures in the "Home" state. They can ask what procedures are currently saved. This question uses the regex `what (.+)?procedures` which allows for slight variations ilke "What are my procedures?" or "What procedures do I have?"

- When editing, users can ask which step in the procedure they are editing at that moment. This helps users reorient themselves if they forget their position during editing. This question uses regex `(?:what|which) step(?:.+)?|where am i`. This allows users to ask questions like "Where am I?" or "What step am I on?" during editing. Convo will respond back with the step number as well as a description of the action at that step.

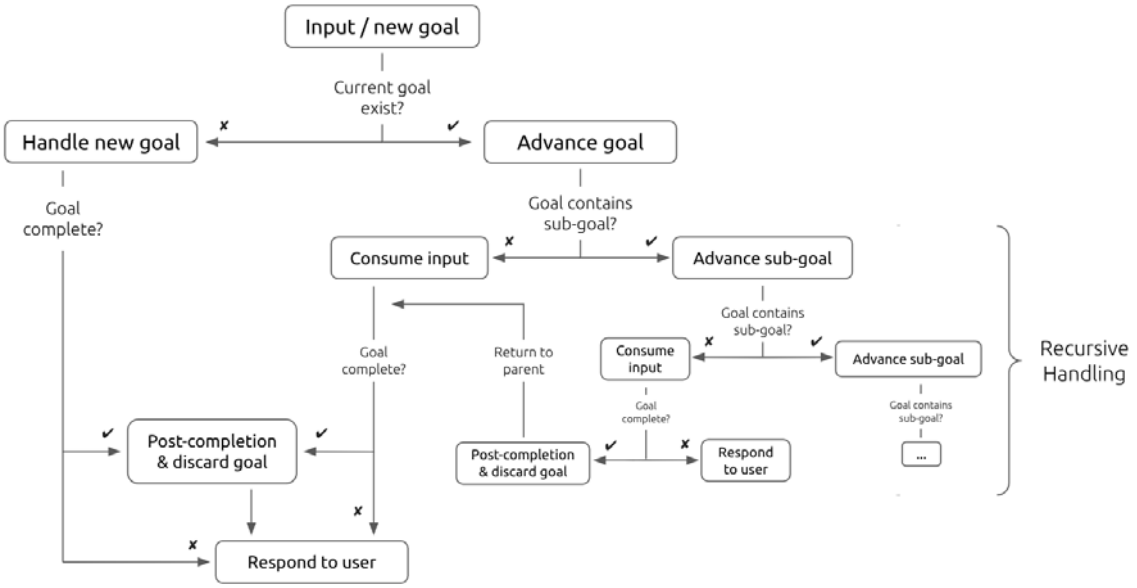


Figure 4-6: The flowchart shows how the dialog manager processes a goal whenever there is a new input or goal. How the dialog manager handles the new input or goal first depends on whether a goal is already in progress. Recursive handling on the right side of the diagram happens when a goal contains a sub-goal. What is not shown is the error handling when an input or goal is invalid or becomes invalid during any point in the process shown. In case of error, the system simply returns a response to the user with feedback on the error.

Handling Goals

If none of the checks apply, the input message will be handled normally, where it can be an input for an existing goal in progress or an entirely new goal. In either case, the input is first sent to the NLU where the NLU will attempt to create a goal based on the intent of the message.

If the NLU successfully returns a goal and there are no goals in progress, the dialog manager processes the new goal. If the goal is already complete, the dialog manager performs any post-completion steps and waits for the next input. If the goal has an error, the goal is discarded and a message is sent back to the user about the error. If there is a current goal in progress, Convo "advances" the current goal.

Advancing means a goal tries to make progress towards satisfying its completion conditions. Advancing happens recursively; if the current goal has a sub-goal, advancing the current goal involves advancing the sub-goal. This recursive advancing

continues until Convo advances a goal that has no sub-goal. Once this is the case, the goal advances by attempting to use the current input to satisfy its completion conditions. If the NLU returns a goal, the advancing goal will add it as a sub-goal if appropriate. If the advancing goal is a type of input goal, the goal will validate the input and try to consume it. This will either complete the goal or produce an error response.

Once advancing is done, Convo checks if the goal is complete, starting from the deepest goal or sub-goal and moving up. If completed, Convo performs its post-completion steps, discards the goal, and moves up to check its parent goal for completion. If the goal is not complete, Convo creates a response message and sends the message to the user and waits for the next input. The response message is usually generated by the incomplete goal because the incomplete goal provides context to the user on why it is incomplete and what needs to be done to complete it. This process of advancing and completing goals is how Convo handles goals. Figure 4-6 details the process in a diagram.

Example: For a concrete example, we have a user creating a procedure, which involves the `GetProcedureActionsGoal`. This goal does not complete until the user signifies that they are done adding new actions through a stop word like "done". The user says "create a variable", creating a `CreateVariableActionGoal` as a sub-goal of `GetProcedureActionsGoal`. The user does not supply two required arguments, a variable name and initial value. Therefore, two `GetInputGoal`s for each argument are created and added as sub-goals of `CreateVariableActionGoal`. Convo starts slot-filling and responds to user by asking the user for one of the arguments, the variable name. Once the user supplies the variable name, Convo advances the current goal `GetProcedureActionsGoal`. This advances its sub-goal `CreateVariableActionGoal` and `CreateVariableActionGoal`'s sub-goal `GetInputGoal`. The `GetInputGoal` is completed by consuming the variable name that the user provided. Now, Convo performs its post-completion steps, which involves setting the name argument of the `CreateVariableActionGoal`, and then checks if `CreateVariableActionGoal` is complete. It is not complete because `CreateVariableActionGoal` has

another `GetInputGoal`. Now, Convo responds to user, asking the user for the initial value and the process repeats. Once `CreateVariableActionGoal` is complete, the action is added to the procedure and Convo waits for the next input.

4.5 Program Manager

The program manager is currently responsible for procedure creation, editing and storage. It interacts mainly with the dialog manager, receiving actions from completed action goals to be added to procedures. In return, the program manager provides the list of created procedures, an interface with the database to store and retrieve procedures and other program-related information.

4.5.1 Components

This section details the four components that can appear in procedures

- Actions
- Conditions
- Value placeholders
- Lists

Actions

Actions are the main essential components of procedures. They represent steps or actions that will be performed during the execution of a procedure, like creating a variable or making Convo say something to the user. All actions are all implemented as subclasses of the Python class `Action`. See Table 4.2 for the full list of actions and their brief descriptions. Actions stores any necessary arguments required for use during execution. The behavior of each action during execution is defined in the `Execution` (Section 4.5.3). Behaviors include sending an audio message to the user or manipulating variables during runtime.

Action	**Description**
CreateVariableAction	Creates a variable with an initial value.
SetVariableAction	Sets the value of an existing variable.
AddToVariableAction	Increments the value of a variable by a specified value. Can only be used with variables storing numerical values.
SubtractFromVariableAction	Decrements the value of a variable by a specified value. Can only be used with variables storing numerical values.
SayAction	Says a specified message or phrase.
ConditionalAction	Creates a conditional (if-else).
LoopAction	Creates a while or until loop.
CreateListAction	Creates an empty list.
AddToListAction	Adds a value to a list.
GetUserInputAction	Waits and listens for input from user.
PlaySoundAction	Plays a supported audio or sound file.

Table 4.2: This lists all of the actions that can be added to procedures on Convo and brief descriptions of each one.

Actions are added to procedures through action goals, a group of goals specifically used when adding actions. Each action has associated and unique action goal. When the user sends a message that is recognize to be an intent to add an action, the goal is created and handled by the dialog manager. When the goal is complete, the corresponding action is added to the procedure as part of the goal's post-completion steps.

Certain actions like `LoopAction` and `ConditionalAction` that represent loops and conditionals, respectively, store a list of actions themselves. This allows Convo to define scope in procedures, to support nested loops and conditionals and to be able to conditionally execute the nested actions based on the condition defined by the loop or conditional. If the condition is not satisfied, the actions stored in the `LoopAction` will not be executed. `ConditionalAction`s store two lists of actions to support executing actions conditionally if the condition is true or false.

Value Placeholders

A value placeholder is a component that helps access values of variables during procedure runtime or execution. It defined as `ValueOf` in Python. The component is especially useful for accessing variables with values that are determined only at runtime. For example, a user creates a procedure that first listens for input, saves the input into a variable "input" and have Convo repeat the input value back to the user. Because the value of variable "input" is not determined until runtime, the value placeholder "holds" the place of the missing value until it is defined. The user can tell Convo to make a value of placeholder by using the phrase "value of" before a variable name and Convo will recognize that the user wants a value placeholder for that specific variable.

The value placeholder component helps reduce ambiguity in conversations when dealing with variables. If the user wants to have Convo say the value of the variable "animal", the user might say "say the value of animal". If Convo was not able to recognize "value of" as a value placeholder, it might misinterpret the user's intent to be saying the phrase "the value of animal" instead of saying the value of the variable

"animal". The component can be used anywhere in the procedure that takes in a value. For example, a user can create a variable "variable one" with the same initial value as another variable "variable two" by saying "create a variable called variable one and set it to the value of variable two".

Conditions

Like in traditional programming languages, users can use conditions for loops and conditionals (if-else statements) in Convo. The condition is represented by the `Condition` class. Convo supports conditions based on equality (i.e. "is", "is not", "equal to", "not equal to") and inequality (i.e. "less than", "less than or equal to", "greater than", "greater than or equal to").

As a design choice, equality and inequality conditions are implemented as separate classes, `EqualityCondition` and `ComparisonCondition` respectively, in Convo. This is because, unlike equality conditions which can be used to compare both numeric and string-based values, currently inequality conditions (e.g. "less than", "greater than or equal to") can only be used to compare numeric values. In the future, Convo will support comparing string-based values lexicographically, like in current programming languages. Both types of conditions have validators during the execution of a procedure that tell users if two values cannot be compared because of differing or, in the case of inequality conditions, unsupported types. Convo also supports a special condition `UntilStopCondition` that can be used exclusively for loops. The condition enables the user to have the loop run forever until the user says "stop", equal to a `while True` loop in Python.

Lists

Convo currently has limited support for the use of lists in procedures. Users can create a list in a procedure using the `CreateListAction` and add elements to the list using the `AddToListAction`. In the next version of Convo, we will be adding more support for lists as we imagine them to be an integral part of procedures for Convo in the future. Lists can be used for a multidude of applications like in procedure for

grocery lists or procedure to set reminders.

4.5.2 Editing Programs

Once users finish creating their procedures, they are able to go back and update them by saying "edit [name]". This transitions them to the "Editing Program" state and an editing context `EditContext` is created. Editing contexts are stored in a stack. The editing context maintains necessary information to support the editing process like

> **Variables**: New variables could be declared and current variables could be removed. Convo needs to keep track so it can inform users when they perform invalid additions of variable-related actions.
>
> **Actions**: The actions that are being edited in this context.
>
> **Current step**: Where the user is currently editing.
>
> **Current scope**: Whether the user is editing actions at the top level of procedure or editing actions within a loop of the procedure.

Users can step forward or backwards through the procedure to edit as needed. Certain actions like loops contain nested actions. Users can step into loops to edit the actions nested within the action. When stepping into a loop, a new editing context is created and added to the stack. When stepping out, the editing context for the loop is popped from the stack, and the user will be placed back into the previous editing context. A current limitation of the editing system is the inability to step into conditionals but this will be resolved in the next version of Convo.

Users have the ability to add a step after the current step, remove the current step or change the current step. At an earlier version of Convo, adding a step would require users to use two utterances: "Add step" then a utterance to the desired action. To improve user experience and efficiency, we removed the requirement to first say "Add step". In the current version, users can add actions in the same way users add actions when creating a procedure. In addition, users can jump to a specific step and can ask

Convo to describe the current step. Convo will use natural language to describe the action instead of simply stating the name of the action and its arguments.

4.5.3 Running Programs

Convo allows users to run the procedures they create. The `Execution` class represents a procedure execution. When a user provides an intent to run a procedure to Convo, Convo instantiates an instance of this class and attaches it to the user's dialog context and transitions to the "Executing Program" state.

The `Execution` object receives the list of actions from the desired procedure and runs through executes each action based on the implemented behavior of the action. All actions contain the properties and information needed during execution. Each `Execution` object also contains a running list of defined variables and their values, which certain actions (e.g. `CreateVariableAction`, `SetVariableAction`) can add to and or modify. Certain actions like `SayAction` asynchronously sends an audio message to the user when executed. This asynchronously message is sent by WebSockets, using the same emitting event as the standard response.

After the `Execution` is initialized for the procedure, Convo starts the execution of its actions in a separate child thread. This allows Convo to continue receiving and processing user messages and utterances in the main thread. Execution in the child thread allows the user who called for the procedure execution to interrupt and stop the execution when it is running. This also allows the user to stop infinite loops in procedures and incorrectly performing procedures. While the procedure is running and not asking for user input, the user cannot utter commands other than *stop* to prevent issues that can result in undesired behaviors.

When a procedure requires further input from the user during execution when reaching a `GetUserInputAction`, the execution is "paused" and the execution state is stored in the dialog manager's context. The execution state contains the information necessary to "resume" execution from the point from which the execution was "paused. This includes any initialized variables and already-executed actions. Once the user provides the necessary input, the execution will "resume" with the creation

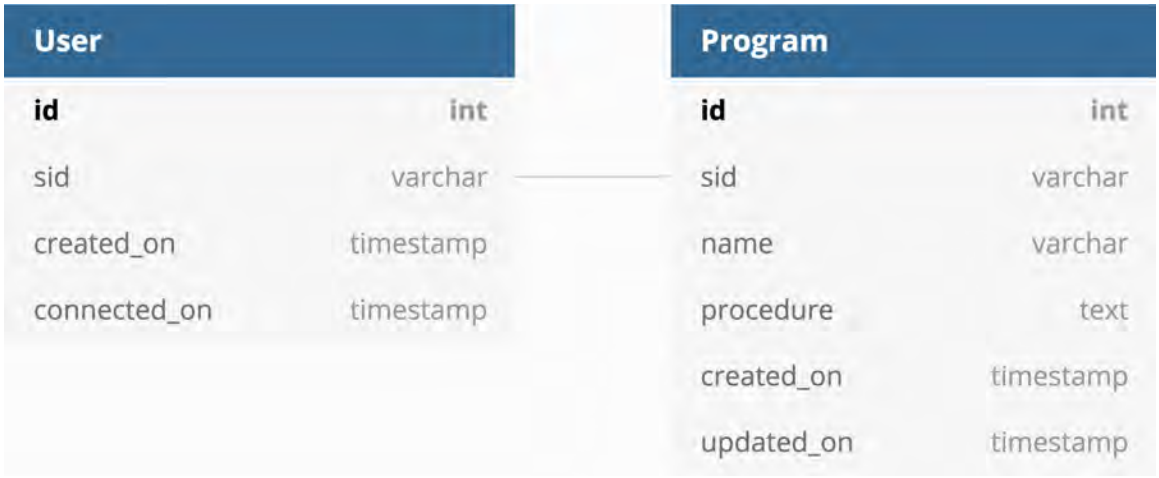


Figure 4-7: The database schemas for the two tables, User and Program, used in Convo to store user and program information. The relation between the two tables is achieved through the `sid` column, which contains the unique IDs of each user using Convo.

of a new `Execution` object using the stored execution state, allowing it to start at the point where the previous `Execution` stopped its progress.

4.5.4 Database and Storage

Procedures and user information are stored in a SQLite database. Unlike most other SQL databases, SQLite does not require a separate server as the database is contained in a single file that can be easily modified [4]. Convo's database file is stored on the server on which Convo is running. Currently, two tables are used for Convo (see Figure 4-7): User and Program. The User table contains the IDs of all users who have connected to and used Convo. The Program table contains all procedures that users have made and associated with users in the User table through a foreign key. When a user connects to Convo, Convo retrieves all procedures associated with the user's ID.

To interact with database and its tables through Python, we use SQLAlchemy, a object relational mapper (ORM) for SQL databases [3]. An ORM associates user-defined Python classes with the database tables and instances of the classes with rows in the corresponding tables. In other words, these user-defined classes define

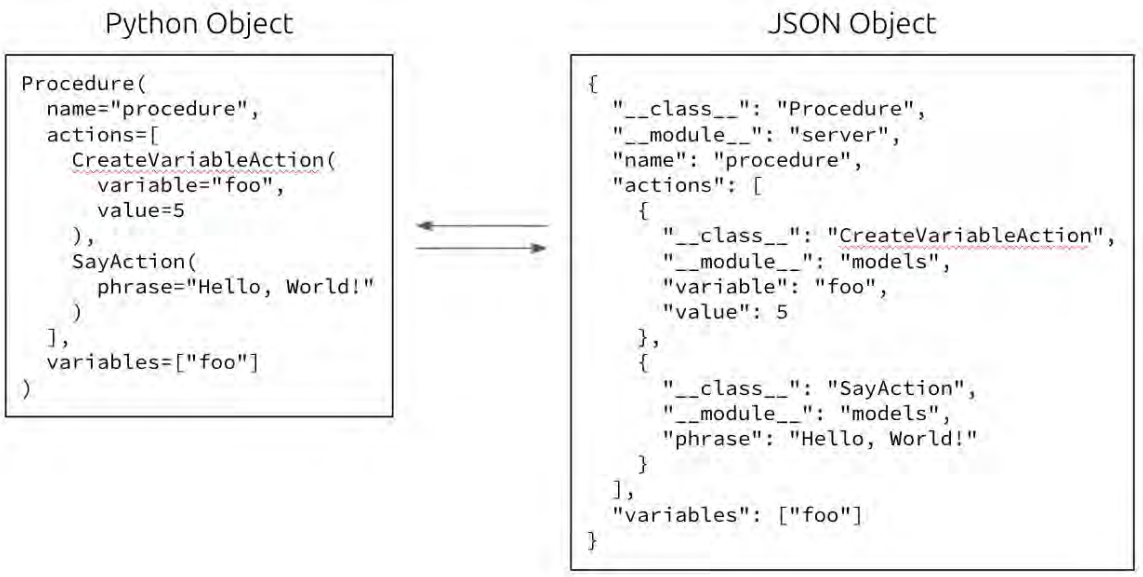


Figure 4-8: An example procedure with its Python object representation and JSON object representation. The additional metadata fields in the JSON object are necessary for the conversion back to a Python object.

the schema of the corresponding database table. This allows us to add, remove or modify rows to database tables by simply creating or modifying objects in Python, removing the need to create and use SQL queries. In Convo's case, a User and a Program class are defined for their respective database tables and helper functions are implemented to store clients as User objects and Procedure objects in Program objects. The database-related classes and functions are handled by a single file `db_manage`.

We cannot store custom Python classes like Procedure and Action in the database without first converting them into a database-friendly format, like strings. To be able to store procedures and actions as strings, we convert them into JSON objects which can be encoded as strings. The encoded string is stored as a column of the Program table.

To convert procedures into JSON, we need to convert the Python object into the Python dictionary representation of the object, which closely resembles a JSON object and the Python library contains the required tools to convert between the two. For procedures and the actions that are defined in each procedure, we take the public

properties of the objects and store them as key-value pairs in the dictionary. Certain metadata like class name and module name are also added to the dictionary to allow conversion from JSON object representation back to Python object representation. Figure 4-8 shows a simplified example procedure and its conversion between its Python object representation and its JSON object representation.

4.6 Deployment

Convo is deployed using Docker on a server hosted by MIT App Inventor. The deployment through Docker is managed by Docker Compose and a configuration file defining the Docker containers and services that should be built and deployed. Convo's deployment involves three main Docker containers, with each container serving a separate part of Convo. Convo's VUI, Convo Core and the Rasa server containing the trained NLU model are each served in an individual container. The database is located on the same container as Convo Core. To secure the server and enable HTTPS, we use Let's Encrypt, a free and automated HTTPS certificate authority, and its accompanying software CertBot, living on another Docker container, to automate deployment with HTTPS. [1].

Chapter 5

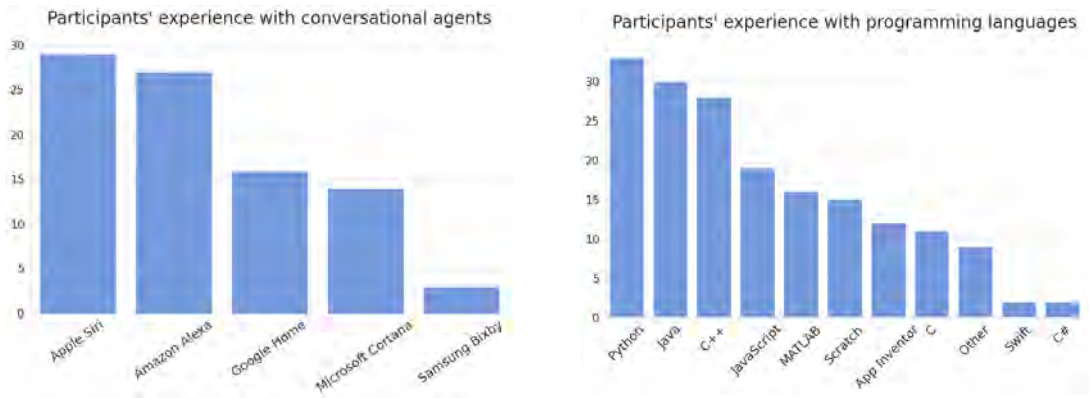
User Study

We conducted a user study to evaluate the effectiveness of Convo as a conversational and NL-based system and to identify additional requirements and user needs for systems like Convo to provide the best possible user experience [7]. The study was conducted across two days on Feburary 4th and 5th of 2020. This chapter details the design and methodology of the user study.

5.1 Participants

For the user study, we recruited forty-five participants (Male = 27, Female = 18) through posters around MIT campus, through certain mailing lists and through word of mouth. Participants included students from a local high school, students from MIT and other local universities and local community members. Based on results of a demographic survey, the age of the participants ranged from fourteen years old to sixty-four years old, with the average age being about twenty-five years old.

Twelve participants self-identified as novice users and thirty-three participants self-identified as advanced users. Novice users have little to no programming experience or have experience in block-based programming languages (e.g. Scratch, App Inventor). Advanced users have experience in object-oriented programming (e.g. Java, Swift) or completed a course or project using a text-based programming language (e.g. Python). All but four participants have experience in some programming



(a) Conversational agents

(b) Programming languages

Figure 5-1: The amount of conversational agent experience and programming language experience as reported by the participants in the demographic survey at the beginning of the user study.

language and all but five participants have interacted with a conversational agent. The conversational agents that participants had the most prior experience with were mainly Apple's Siri and Amazon's Alexa (see Figure 5-1a). The programming languages that participants had the most prior experience with were mainly Python, Java and C++ (see Figure 5-1b). All forty-five participants were able to complete at least one part of the user study. For compensation, participants were given an Amazon gift card worth $20 for novice users or $30 for advanced users. The difference in value was due to the fact that advanced users had to complete an additional stage, requiring more time to complete the user study.

5.2 Methodology

Before participants started the user study, an informed consent form was provided to and signed by them. An additional consent form to be signed by parents or guardians was sent to any participating minors before the day of the user study. We asked participants to bring their own laptops and earphones but we brought additional laptops and earphones in the event that participants did not bring their own or did not want to use their own.

When participants first visited the user study website, they read detailed informa-

tion on what to expect during the study and watched a video on how to maneuver and interact with the different text, voice and voice-or-text systems. Once participants have a grasp of how the study will proceed, they were asked to fill out a demographic survey.

After filling out the demographics survey, the participants proceeded onto the actual tasks for the study. The user study had three stages: practice, novice and advanced. In each stage, participants needed to complete three tasks, one for each of the three different systems using Convo: text-based, voice-based and voice-or-text-based. Participants were shown what task completion looked like through a video at the beginning of each task. Each task consists of creating a procedure with certain actions. Participants could not advance until they successfully completed the current task. Novice users only had to complete the practice and novice stages; advanced users had to complete all three stages. The practice stage was designed to familiarize participants with the systems and the environment, while the other two stages were more involved.

In each stage, the tasks varied slightly but were similar enough so that participants produced procedures with similar actions. The tasks were randomly assigned to each of the three input systems (i.e. voice-based, text-based, voice-or-text-based). We introduced slight variations in the tasks and randomized the order of systems participants used in order to reduce possible learning effects. The general tasks for each stage are detailed below.

1. For the practice stage, we asked participants to create a simple procedure that makes Convo audibly say "hello world", equivalent to the standard of having users print out "hello world" in text-based programming languages as an introduction.

2. For the novice stage, we asked participants to create a procedure where Convo will listen for user input and play a corresponding animal sound if the input matches one of two specified animals (e.g. *If I say cat, play the meow sound. If I say dog, play the bark sound*). After creating the procedure, we asked

Stage	Instructions
Practice	Say: 1. "Create a procedure called hello world" 2. "Say hello world" 3. "Done" 4. "Run hello world"
Novice	Say: 1. "Create a procedure called pet sounds" 2. "Get user input and save it as pet" 3. "If the value of pet is **'cat'**, play the **cat** sound" 4. "Done" 5. "No" 6. "If the value of pet is **'dog'**, play the **dog** sound" 7. "Done" 8. "No" 9. "Done" 10. "Run pet sounds"
Advanced	Create a program that does the following: 1. Use a while loop 2. Listen to user input **5** times 3. Every time it listens, if the user input is **'dog'**, play the **dog** sound. If the user input is **'cat'**, play the **cat** sound.

Table 5.1: This table shows instructions that participants may have seen during each stage. Participants followed step-by-step instructions in the practice and novice stages and general instructions in the advanced stage. The bolded words in the instructions for novice and advanced stages are varied and randomized for each participant and task. The pair of animal sounds for these instructions were dog and cat.

the participants to run the procedure. The stage required participants to use conditionals and variables to complete the tasks. We varied the pair of animals that participants should check for and their corresponding sounds among the tasks. The possible pairs of animals were (dog, cat), (cow, horse) and (bird, cricket).

3. For the advanced stage, we asked participants to create a procedure where Convo will continuously listen for user input for a specified number of times. The stage builds on top of the novice tasks by adding a loop. In a loop, Convo asks for input and blocks, waiting for the user to provide the input. When the user provides an input, Convo will need to play the corresponding animal sound if the input matches one of the two specified animals. After playing the sound, Convo asks for input again, repeating this set of actions for a specified number of times. We varied the number of times Convo will listen for input, from three times to five times. Just like in the novice stage, we also varied the pair of animals that the participants should check for. The same pairs of animals from the novice stage were used in the advanced stage.

In the practice and novice stages, participants followed step-by-step instructions; the next step was only shown when current step was successfully completed. If participants failed to follow the current step at any point, the system reset back to the previously completed step. This is to remove any potential issues arising from an incorrect action performed by the participants. In the advanced stage, participants were given more freedom with the instructions being given all at once and more high-level. The advanced tasks are deemed complete when the procedures matched the expected behavior. If not, participants are free to edit their procedures or reset their progress and try again. See Table 5.1 for example instructions used for each stage.

In all stages, participants were given as much time as they needed to complete all tasks and if necessary, participants were able to restart a task from the beginning. Participants were allowed to ask questions by raising their hand. For any technical questions, we followed a protocol designed and written beforehand so participants all

Question Type	Question
Likert	I found it difficult to complete the goal with the [input]-based system.
	I found programming with the [input]-based system difficult to use.
	I am satisfied programming with the [input]-based system.
	I found programming with the [input]-based system efficient to use.
Free-form	What did you like about programming with the [input]-based system?
	What was frustrating about programming with the [input]-based system? How could we make it less frustrating?
	What did you wish you could say to the agent? What didn't the agent understand?
	What features can we add, change, or remove to make the system better?

Table 5.2: The questions and their types, Likert or free-form, in the questionnaire answered by participants after completing the task with [input]-based system where input is either voice, text or voice-or-text.

receive the same level of advice and answering.

After completing all three tasks in the novice and advanced stages, participants filled out a questionnaire about their experience with the specific system (see Table 5.2). After completing all of their required stages, participants also filled out a final questionnaire. The final questionnaire had the participants compare the three systems with which they interacted (see Table 5.3). All of the aforementioned questionnaires consisted of 5-point Likert scale questions and free-form short answer questions [27]. After completing the final questionnaire, participants are directed to a page where they can input their emails to receive their Amazon gift cards. For anonymity and separation from the study results, only email and programming level (novice or advanced) were recorded in the last page.

Question Type	Question
Likert	Preference between text-based system and voice-based system.
	Preference between voice-based system and voice-or-text-based system.
	Preference between voice-or-text-based system and text-based system.
	I think programming with just voice is easier than programming with just text.
	I think programming with just text is easier than programming with both voice and text.
	I think programming with just voice is easier than programming with both voice and text.
	I think programming with just text is frustrating or hard.
	I think programming with just voice is frustrating or hard.
	I think programming with both voice and text is frustrating or hard.
	I enjoyed the process of trying to complete the tasks.
	I think being able to program using voice is useful.
	I like being able to program using voice.
	If I could, I would continue to learn how to program using voice.
	I am a programmer.
Free-form	We want the agent to eventually be able to explain things about how it works. If you were to ask the agent any question, what would you ask it? Please list as many questions as you can think of.
	What challenges did you run into while interacting with the agent?
	What questions do you have about the system?

Table 5.3: The questions and their types, Likert or free-form, in the final questionnaire answered by participants after finishing all required tasks.

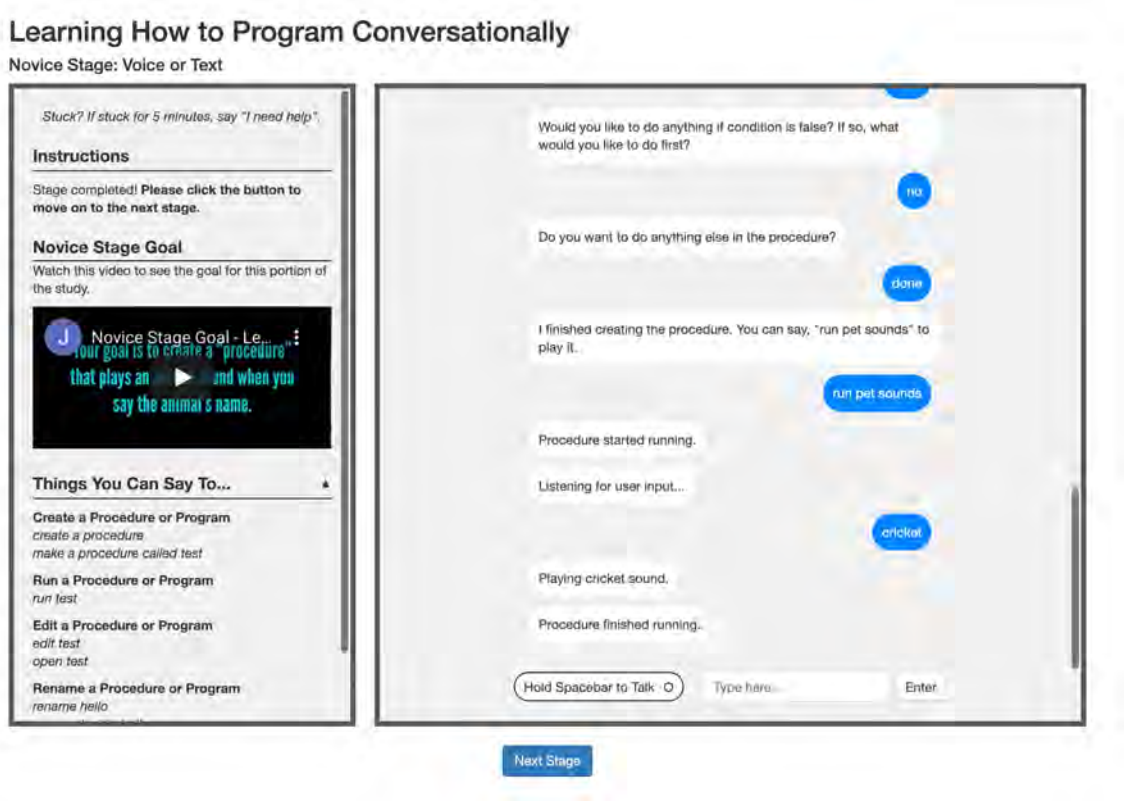


Figure 5-2: This screenshot shows one of three tasks in the novice stage of the user study that participants had to complete. The objective here was to complete the task using the voice-or-text-based system.

5.2.1 User Study Web Interface

The user study web interface is served by the same Node server serving Convo VUI (see Section 4.2). The majority of the web interface consists of HTML pages for the tasks, task instructions and questionnaires and surveys and JavaScript files needed to connect to Convo VUI's Speech-to-Text service and to Convo Core on the relevant pages. See Appendix A for a full list of screenshots of the user study website. A simple username-and-password authentication service was implemented to prevent participants from visiting the website before and after the study. The username and password combination was changed for each day of the study.

The pages for each study task all contain two sections (see Figure 5-2). The left section is the sidebar containing various information regarding the task. The instructions for the current task is at the top of the sidebar. Next on the sidebar is the

video explaining the task and showing what success looks like for this task. Lastly, the sidebar contained a section documenting examples of utterances that participants can say to express certain intents to Convo. The examples are updated as participants interact with Convo to show what intents are allowed at that moment in their conversation with Convo.

The right section visually showed the entire conversation history between participants and Convo, with participants' messages showing up in blue bubbles and Convo's messages showing up in white bubbles. Below the conversation history section was the input modality, corresponding to the system selected for the current task. A text-box was shown when working with the text-based system; a microphone button was shown when working with the voice-based system. Both text-box and button were shown when working with the voice-or-text-based system. After participants complete a task, a "Next" button appeared at the bottom of the screen taking them to a questionnaire.

We accounted for two potential system-related technical issues that could arise and prevented participants from advancing through the user study tasks. The two issues were the following:

1. In tasks using the voice-based system, the automatic speech recognition (ASR) service is a single point of failure. The ASR can fail because of connection issues with Google's API, and heavily-accented or fast speech from participants can result in continuous inaccurate results which can be frustrating for the user. To try to solve the latter reason, we asked asked participants to speak more slowly and loudly. If the inaccurate transcription continues to be a problem for specific phrases, we implemented a hidden JavaScript function that can be invoked by us in the console to show a text-box that the participants can use to type. If the hidden text-box was shown, we asked the participants to keep using the voice-input as much as possible to complete the task.

2. We tried to resolve as many bugs as possible before the user study but new bugs were encountered the user study. A bug had prevented a few number of par-

ticipants from completing a task. We implemented a second hidden JavaScript function to show a button, allowing them to skip the bugged task and move on. This only happened a couple of times.

5.2.2 Data Collection

Various types of data were collected throughout the user study. As mentioned in Section 5.2, participants filled out a demographic survey at the beginning of the study. The questions asked in the survey were:

1. **How old are you?**

2. **What best describes your gender?** We included the option to self-describe and the option to not provide a gender.

3. **Is English your first language?** Participants who are non-native English speakers may see poorer speech recognition results compared to participants whose English is their first language. This is because the accents in their speech can lead to under-performance of the ASR system [16].

4. **Would you consider yourself a novice or advanced programmer?** We included definitions of a novice and an advanced programmer so participants can self-identify.

5. **What programming languages have you used before?** We included block-based programming languages (e.g. Scratch, App Inventor), traditional text-based programming languages (e.g. Python, JavaScript) and an option for participants to provide additional programming languages.

6. **What type of conversational agents have you interacted with before?** We included device-based conversational agents (e.g. Amazon Alexa, Google Home), virtual voice assistants (e.g. Microsft Cortana, Apple Siri, Samsung Bixby) and an option for participants to provide additional conversational agents.

Participants answered questions after each completed task in the novice and advanced stages. These questions gathered participants' feedback and opinions on various aspects on each of the three systems like usability, efficiency, satisfaction and preferences. Participants also answered questions after completing all of the required stages. These questions focused on having participants compare the three different input modalities and their preferences between them. We also asked participants more general system-related feedback like their opinions on using voice to program, what questions they had about the system, and what other features they would like to see in a conversational programming system like Convo.

Aside from collecting data from questionnaires, we were also collecting data as participants complete their tasks. These data help provide quantitative results from the user study. We collected data on

- **Conversation**: All text- and voice-input transcripts, all audio data and the type of each input (text or voice) were collected. For privacy reasons, this was properly informed to the participants through the consent forms they signed before the user study.

- **Assistance needed**: The number of times participants raised their hand and asked for help.

- **Resets**: The number of times participants reset the progress on a task or refreshed the page.

- **Time to completion**: The total duration of time for a participant to complete each task.

- **Bugs and issues**: The number of times participants needed to skip a task because a bug prevented participants from completing it.

- **Instances of poor speech recognition**: The number of times a text-box was shown in tasks using the voice-based system; the number of times participants repeated phrases; the number of phrases participants needed to progress through

each step in the novice stage; the number of words replaced by the VUI because of inaccurate ASR (see Section 4.2).

Chapter 6

Results and Discussion

This chapter details the qualitative and quantitative results of the user study and the discussion and conclusions drawn from the results including feedback we received on what users may want in conversational programming systems.

6.1 Qualitative Results From Open Coding

To analyze the free-form short-answer responses we received from the participants, we used a qualitative analysis approach known as open coding, part of the grounded theory method. [28]. Open coding is a method by which we iteratively parse through the short answer responses, coming up with a set of themes and concepts ("codes") to which we can attribute each short-answer. Through open coding, we identified fourteen different design themes which we separated into two different categories, positive feedback and recommendations. In all, we coded 651 occurrences of the design themes with representative responses below [7].

Positive Feedback

Themes that fall into the positive feedback category are

> **Efficient** (49/651): *"I liked how quick it was. Having to just speak to program is far quicker than typing [...]"*

Usable (48/651): *"Easy to use just had to talk." "I liked that I could just tell it what needs to be done."*

Accessible (32/651): *"It was super fast and I was able to type out shorter commands while speaking the longer ones"*

Effective coding features (9/651): *"I liked that I wasn't beholden to strict grammar (didn't complain about missing commas that would likely give good context)" " It was very speedy for simple actions and I had an idea of how it was working under the hood"*

Interesting (6/651): *"It's pretty cool that I was able to construct a program with my voice!" "It feels cool to do this - I can imagine coding while driving or doing housework."*

Recommendations

Themes that fall into the recommendations category are

Increase agent interaction (91/651): *"It would be interesting to be able to ask the agent for information on the code I already wrote." "I wish it could provide me some guidance when I am making mistakes."*

Add visualization (72/651): *"Maybe some sort of visualization of the function being built up as interaction progresses."*

Improve efficiency (30/651): *"It also seems quite inefficient to figure out the right way to express a statement in actual words that otherwise can be typed in a programming language [...]"*

Reduce cognitive load (12/651): *"I can't see my program and I have to remember what's going on, that will become infeasible very quickly."*

Increase transparency (25/651): *"How do you register what I'm saying?" "What kind of voice recognition is used?"*

Reduce ambiguity (12/651): *"Being unable to ask the AI for clarifications" "Please specify exactly what you need?"*

Convey system purpose (9/651): *"How is this system going to be implemented? Where would you use this system?"*

Improve speech-to-text (190/651): *"It seems like if speech recognition worked well, it would be a better choice, but having this is useful (especially in a noisy environment)."*

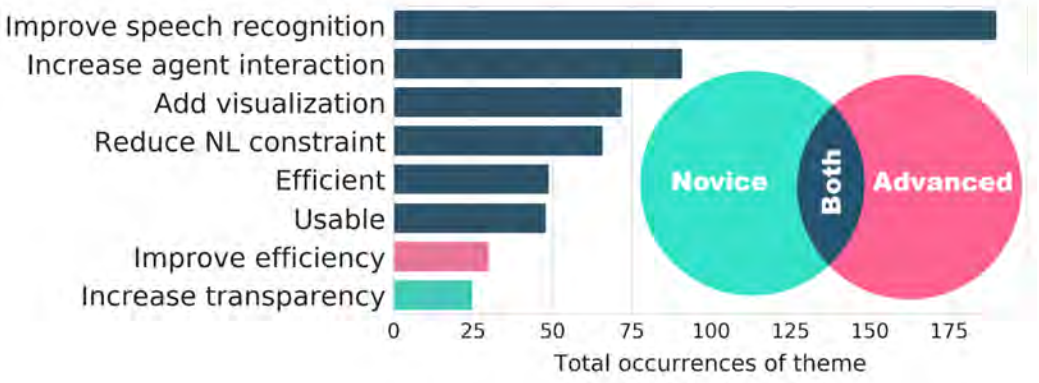
Reduce NL constraints (66/651): *"It would be great if I don't have to explain in a language of programming but I can talk as I talk to someone else."*

6.1.1 Thematic Comparisons Between Novice and Advanced Users

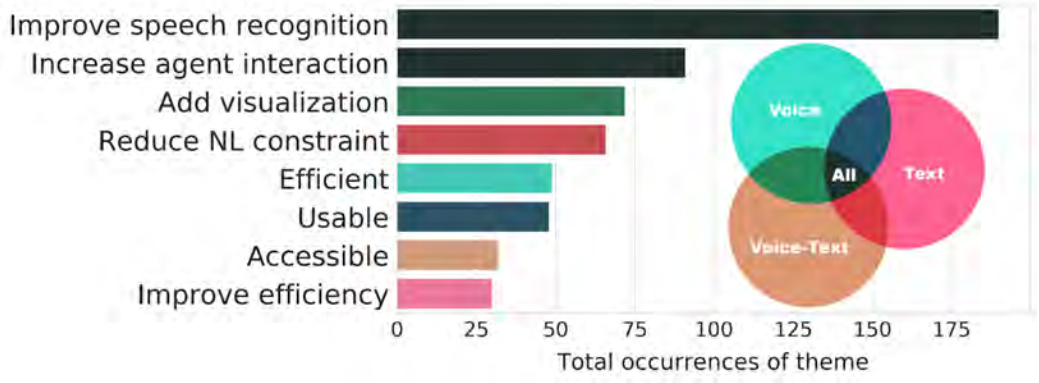
Figure 6-1a shows comparisons between occurrences of the top seven themes identified from responses of novice users and responses of advanced users. Novice and advanced users share six of the top seven occurring themes, with the top four all being from the Recommendations category. The only theme in top seven themes that differ between novice and advanced users is between improving efficiency and increasing transparency. Novice users emphasized increasing transparency over improving efficiency and advanced users were the opposite. We suspect that this is because advanced users may already have an underlying understanding of or experience on how the programming system works while novice users will still be curious. If advanced users were to use Convo, they would likely want to improve the efficiency more instead.

6.1.2 Thematic Comparisons of Input Modalities

Figure 6-1b shows comparisons between occurrences of the top five themes identified from users responses to each of the three different input systems (voice-input, text-input, voice-or-text) that users utilized during the user study. Responses to all three



(a) The total number of occurrences for the top seven themes from advanced users' responses and top seven from novice users' responses.



(b) The total number of occurrences for the top five themes from responses on each of the systems that users used in the user study.

Figure 6-1: Comparisons of the top themes from responses between novice and advanced users and from responses on each of the different input systems. The colors represent which user group(s) or system(s) from which each of the top theme came. The Venn diagrams match the color to the label.

input systems emphasized the need to improve speech recognition and increase conversational agent interactions. Responses to both voice-input and voice-or-text systems emphasized adding visualizations which will help reduce cognitive load. In responses to text-input and voice-or-text systems, reducing the natural language constraints was a top theme. Showing the potential for voice-input systems, a top theme from responses to voice-input systems was efficiency. Responses to the text-input system emphasized improving efficiency, possibly due to the fact that typing out commands in natural language will usually be longer than commands in traditional text-based programming languages. Responses to the voice-or-text system complimented on its

	S. Preferred - 1	Preferred	Neutral	Preferred	S. Preferred - 5
Novice	1 (Voice)	0	1	3	6 (Voice-or-Text)
	6 (Text)	3	1	1	0 (Voice)
	2 (Voice-or-Text)	4	1	1	3 (Text)
Advanced	0 (Voice)	0	1	7	14 (Voice-or-Text)
	12 (Text)	5	2	2	1 (Voice)
	3 (Voice-or-Text)	6	4	2	7 (Text)

Table 6.1: The preferences of novice and advanced participants between each pair among the three possible input modalities. Each column shows the number of participants who selected that preference for the input modality. Generally, both user groups preferred the text-based and voice-or-text-based systems over the voice-based system while having mixed-preferences between the voice-or-text-based and text-based systems.

accessibility due to the fact that users can choose input modalities depending on the situation.

6.2 Quantitative Results

From the 5-point Likert scale responses (Tables 5.2 and 5.3) and the quantitative data that we collected during the tasks, we performed various between-subject analyses using analysis of variance (ANOVA) and various within-subject analyses using repeated measures ANOVA [26]. The between-subject conditions were the novice and advanced stages of the study and the within-subject condition was the input modality type.

6.2.1 Preferences and Difficulties Among Input Modalities

The first three Likert scale questions (Table 5.3) from the final survey of the user study asked participants their preferences between each possible pair of the three input modalities. Between text-based and voice-based systems, both novice and advanced participants strongly preferred the text-based system over the voice-based system. A

possible bias towards text-based system may exist among the participants because of the fact that most traditional programming languages are text-based. From the system surveys, participants also felt it was more difficult to complete the tasks with the voice-based system based on the Likert responses in the system surveys. Novice participants made significantly more incorrect utterances with the voice-based system ($M = 17.38$) compared to the text-based ($M = 1.38, p = 0.0001$) and voice-or-text-based systems ($M = 4.77, p = 0.0017$). The large number of incorrect utterance is most likely due to the inaccuracies of the ASR service, suggesting that speech recognition has room for improvement. However, no significant differences were observed with advanced participants.

Between voice-or-text-based and voice-based systems, both novice and advanced participants strongly preferred the voice-or-text-based system over the voice-based system as well. Between voice-or-text-based and text-based systems, there was a mixed result in preferences. For advanced participants, there was an equal mix of participants who preferred one over the other and vice versa. However, advanced participants did perceived the voice-or-text-based system ($M = 2.94$) to be more difficult to use compared to the text-based system ($M = 3.5, p = 0.02$). For novice participants, there was a slight preference towards the voice-or-text-based system but there was no significant difference in the level of difficulty between the different systems as we observed with advanced participants.

Based on Likert results from the system surveys, novice participants were more satisfied with the voice-or-text-based system ($M = 2.61$) than the voice-based system ($M = 3.44, p = 0.0003$). They also found the voice-or-text-based system ($M = 2.66$) more efficient to use than the voice-based system ($M = 3.47, p = 0.0006$). We saw no significant difference in satisfaction and efficiency between systems among the advanced participants. The preference towards the voice-or-text-based systems compared to the other two systems suggests that participants see value in a mixed-input system. This is supported by the fact that from our qualitative analysis of the responses, a top design theme on the voice-or-text-based system was accessibility or the ability to choose which input to use at a given situation (See Section 6.1.2 and

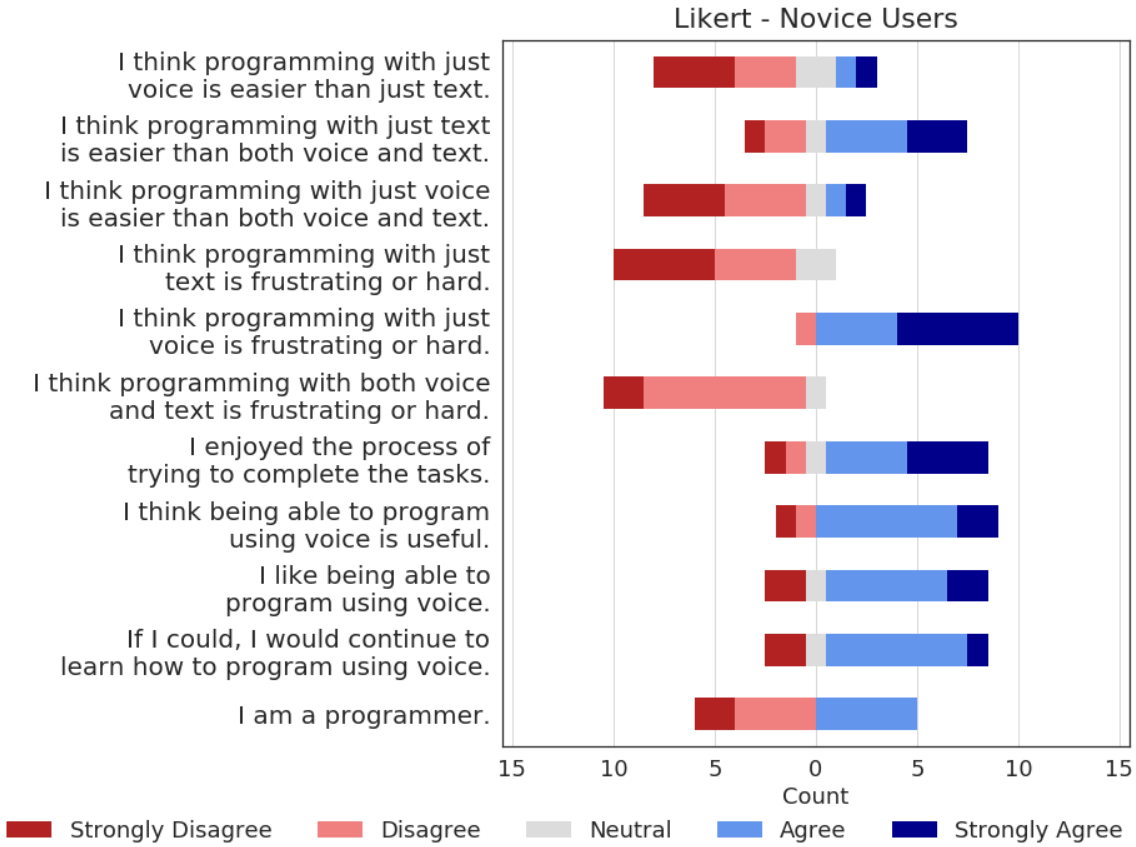


Figure 6-2: Novice user responses to Likert scale questions from the final survey. In general, novice participants found voice-based programming to be more difficult than the other two systems. However, the majority of novice participants saw future potential for voice-based programming given the responses.

Figure 6-1b).

6.2.2 Cognitive Load Effects

To investigate for any possible effects from cognitive load, we analyzes the data we collected on the number of resets by a user, time to goal completion and the number of times participants asked for help. We excluded the data from novice stages because we provided step-by-step instructions in the novice stage, so there is minimal cognitive load on the user. We examined the number of resets because participants had mentioned during the study that they would sometimes reset the task if they forget what step or what actions they had already added to their procedure. We found no significant difference in the number of resets participants made between the three

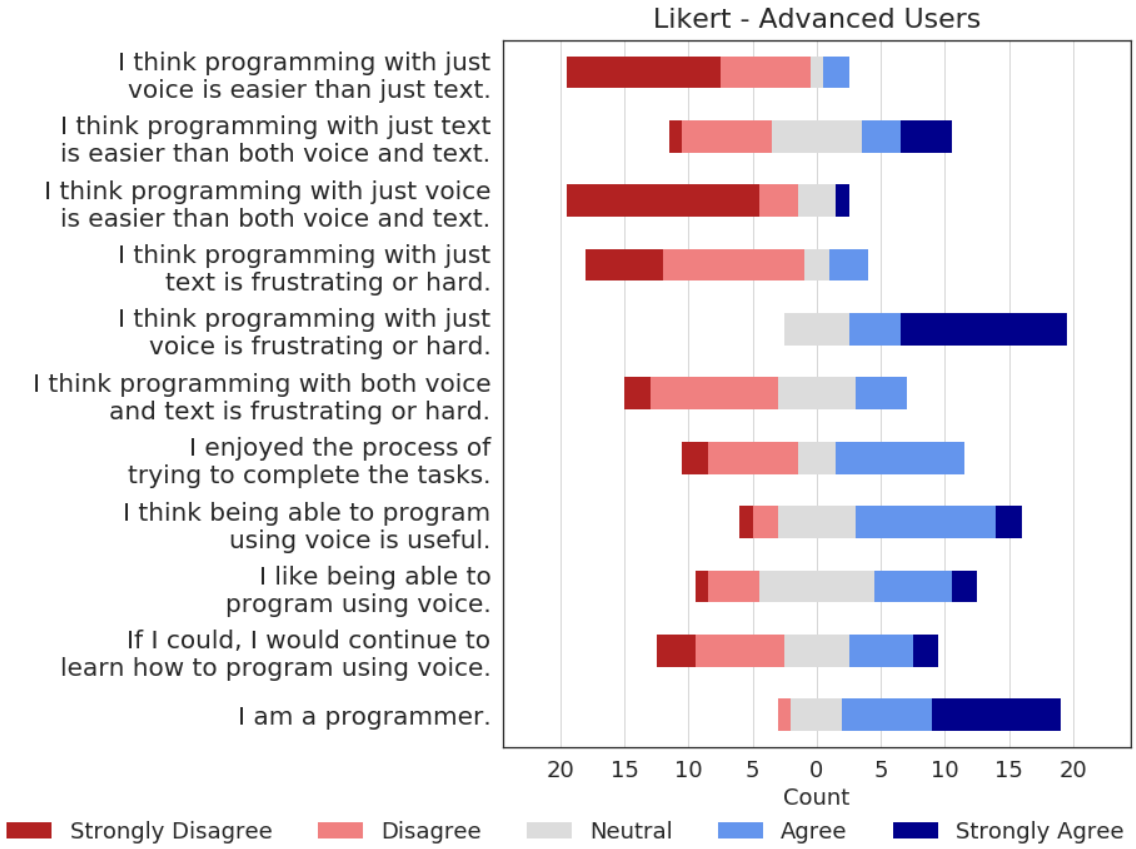


Figure 6-3: Advanced user responses to Likert scale questions from the final survey. Advanced participants were generally less favorable towards voice-based programming than novice participants.

differnet input modalities. We also observe no significant difference in the number of times participants asked for help or in the time participants took to complete the advanced tasks.

6.2.3 Potential for Programming Using Voice

While results have shown that most participants displayed preferences towards text-based systems, we also observe that there is a large potential for voice-based programming. From the Likert scale responses in Figure 6-2, we can observe that most novice participants strongly agree that programming with voice can be useful and enjoyable. In fact, most novice participants also strongly agree that if they could, they would continue to learn programming using voice.

As shown in Figure 6-3, there were more mixed opinions among advanced participants. While many advanced participants strongly agree that being able to program using voice is useful, more advanced participants would not continue to learn how to program using voice. This suggests that while voice-based programming may not be preferred for experienced programmers, it can be used to teach novice programmers or people with little to no programming experience computational thinking and programming skills and concepts. The potential for voice programming is especially great for people who cannot learn these skills through traditional methods.

6.3 Design Recommendations for Future Conversational Programming Systems

Based on our qualitative and quantitative results, recommendations can be made that will be useful for Convo and future conversational programming systems.

6.3.1 Be Flexible and Accessible

Our results suggest that conversational programming systems should be flexible and accessible, having options for voice-based and text-based inputs. User choices can

vary based on preferences and experience, so conversational programming systems should be configurable and have options for users to tailor it to their needs or task. We observed in our results that novice users tend to find voice-based inputs useful and more enjoyable than advanced users. To support this, we examined the number of text and voice utterances that participants used during the study. We observed that while there was no significant difference between the overall number of voice utterances and text utterances, participants in the advanced stage tended to type (text-input) rather than speak (voice-input) ($p = 0.003$). Advanced users also seem to find conversational programming potentially cumbersome and unnecessarily verbose. This is likely due to the fact that advanced users have programming experience in traditional text-based languages and were used to the conciseness of those syntax-restricted languages.

While advanced users do prefer text-based systems, both user groups found value when having both input modalities and the ability to choose. Many participants had responses similar to "I liked being able to use the voice for longer commands, and the text for shorter commands or misunderstood commands." These responses were supported by the significant difference in number of characters ($p = 0.004$) and *words* (p=0.003) per voice utterance over text utterance, meaning voice utterances tended to be longer than text utterances. In addition, the ability to switch between voice and text inputs is useful if either system becomes insufficient for programming. Even with current speech recognition technologies, accents and imperfect conditions can lead to inaccuracies which will degrade the user experience with voice inputs. Many participants had responses related to this like "Sometimes it had problems understanding my speech, so I resorted to typing things."

Overall, having a conversational programming system be flexible and accessible can improve the user experience. We will be working on this with Convo as well.

6.3.2 Reduce Cognitive Load

When designing conversational programming systems, developers should take caution not to introduce too much cognitive load and develop ways to reduce cognitive load. In our qualitative analysis, participants had responses related to the increased cognitive

load like “I found it quite challenging to figure out the logic of the program entirely in my head”.

Including visualizations is simple yet effective recommendation to reduce cognitive load, especially for systems that use voice and for sighted users. We observed this desire for visualizations in our free-form responses as a recurring response and theme. This is supported by the fact that adding visualizations is a top occurring theme among both novice and advanced users as well as a top occurring theme for the systems that supports voice input. We see users asking questions like "Will there be added functionality where voice is translate to running text so that one does not need to ‘hold’ the program in memory?" and "What does my function currently look like?" We also see users having challenges like "being unsure what [they] had done so far" or "being unable to visualize the commands completed so far." For those who need it, including visualizations as an option will undoubtedly help reduce the cognitive load when designing and creating programs in a conversation-based system.

Other design features to potentially reduce cognitive load include improving speech recognition and natural language understanding. One method is to reduce the constraint on the NLU so that users do not have to remember specific phrases to invoke specific intents. In addition, having a robust question answering system may help as well. If users forget at any point or have any outstanding questions, the system should be able to help them. These are features that we are working to include and improve in Convo as well.

6.3.3 Improve Speech Recognition and Natural Language Understanding

Improving speech recognition and reducing NLU constraints are two of the top reoccurring themes that came out of our thematic analysis (Figure 6-1). This shows that speech recognition and natural language have ample room to improve. Currently, Convo uses Google’s state-of-the-art ASR system, yet it still resulted in inaccuracies that made programming with voice frustrating. One potential avenue to mitigate

these issues is training a custom ASR model that incorporates corpus of widely used phrases in conversational programming. However, we would have to be careful to balance between generalization and training on specific phrases, as this could increase NL constraint instead of reducing it. Participant responses like “It’s a very cool idea, and with expanding the dictionary it could work better” and "More options to say things" show that reducing NL constraint should also be considered when designing conversational programming systems. With research in the speech recognition and NLP space maturing and improving with breakthroughs like BERT, we look forward to the improving upon Convo as well see other conversational programming systems enter the space.

Chapter 7

Conclusion

In this thesis, I introduced Convo, a voice-first conversational programming system developed with the ultimate goal to to empower more people to develop programming skills and concepts and computational thinking as well as another outlet for people to express their creativity. I presented an overview of Convo, an example scenario of how a user might use Convo, and Convo's implementation details. Next, I presented the methodology and details of the user study that we conducted using Convo to learn more how people will perceive and use conversational programming systems as well their effectiveness. Lastly, I presented the results and discussion from conducting the user study with over forty participants. In the discussion, I also presented the design recommendations that we came up with based on the results of the study for Convo and other conversational programming systems to consider.

With Convo and the results of the user study, we have shown that conversational programming systems have a huge potential to be the starting points of numerous people's path towards learning how to program. Convo takes advantage of the increasing prevalence of voice-based technology to give people who do not have access to traditional programming experience and learning methods a way to develop programming skills. We hope that Convo and future conversational programming systems will help pave the path towards using conversational programming to empower more people and democratize computational thinking and programming.

Chapter 8

Future Work

Because Convo can still be considered a conversational programming system in its infancy, there are numerous potential avenues for future work. First and foremost, we should look towards applying the design recommendations that were conceived in the discussion of the user study and its results. This includes reducing cognitive load and improving speech recognition and natural language understanding.

Reducing cognitive load in Convo will be important. One feature is adding visualizations of programs to Convo as mentioned in our design recommendations. While Convo is a voice-first programming system, adding visualizations, when able to, can help reduce cognitive load, especially for sighted users. Another feature of Convo that can be improved is its question-answering system. Currently, it is very limited in scope, but we should be able utilize current language models like BERT to implemented a NLP-based question-answering system. Having a robust QA system will allow Convo to be more transparent as well, a desire expressed by many of the participants in our user study.

For improvements towards speech recognition and natural language understanding, one avenue would be to try to train a separate custom ASR model to use that incorporates common conversational programming phrases. Another avenue would be to improve the ML-based Rasa NLU of Convo. Currently, the NLU only support extracting a single intent from an input message. If provided a message like "create a variable and add one to the variable", Convo would most likely only recognize that

the user wants to create a variable. Supporting recognition of multiple intents can help reduce NL constraints as well, another recurring theme from participants. In addition, improving the NLU's argument extraction will also be beneficial for Convo.

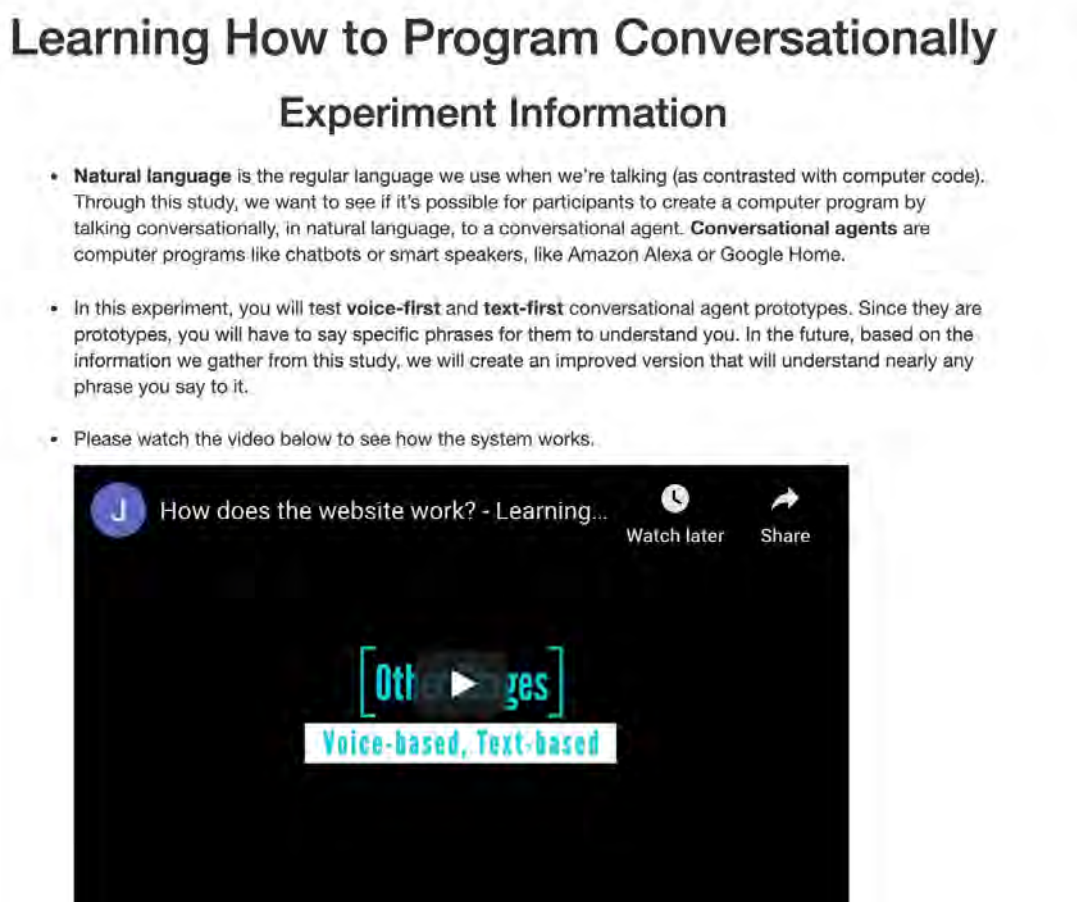
Another effort can be made in extending Convo's programming capabilities. For example, work can be done in adding support for classes, objects, events and other features found in traditional object-oriented programming languages. These features were planned but was not high priority. In addition, Convo's modularity allows us to easily support new actions in programs written in Convo. For example, as machine learning and AI are getting more popular, ML-based actions like sentiment analysis action or image classification action can be supported. We believe that Convo can develop into a full fledged conversational programming system and once that happens, curriculum can be developed around Convo to teach computational thinking and programming concepts using voice and natural language.

With results showing that users with little to no programming experience see voice-first programming as a potential way to learn programming, we can also make an effort towards developing a curriculum around conversational programming. Before that, we can allow Convo to mature more through new features and more accurate speech recognition and better natural language understanding. We would also conduct user studies beforehand to gauge how effective Convo and conversational programming can be in teaching programming skills and concepts and computational thinking.

Appendix A

User Study Web Interface

A.1 Home Page



- The experiment consists of:
    1. a **practice stage**,
    2. a **novice stage**,
    3. and an **advanced stage**, if you identify as an **advanced programmer**.
- For each stage, you will complete three tasks (all very similar tasks) in three different ways:
    - through speaking,
    - typing,
    - and speaking/typing (you can choose).

  The **order** in which you complete them will be **randomized**.
- At any point, if you're absolutely stuck (for example, no progress for 5 minutes), you can type/speak, "I need help", and follow the instructions on-screen.
- You will also complete **surveys** about yourself and your experience with the system. We want to improve the system as much as possible, so we appreciate any and all feedback you have!
- You should expect the experiment to last approximately 30-60 minutes.
- You will be able to view **examples/hints on a sidebar on the left side of the screen**.
- We are evaluating and comparing **the systems** (voice-based, text-based, and voice/text-based systems), **not your abilities**. Additionally, all results will be anonymized.

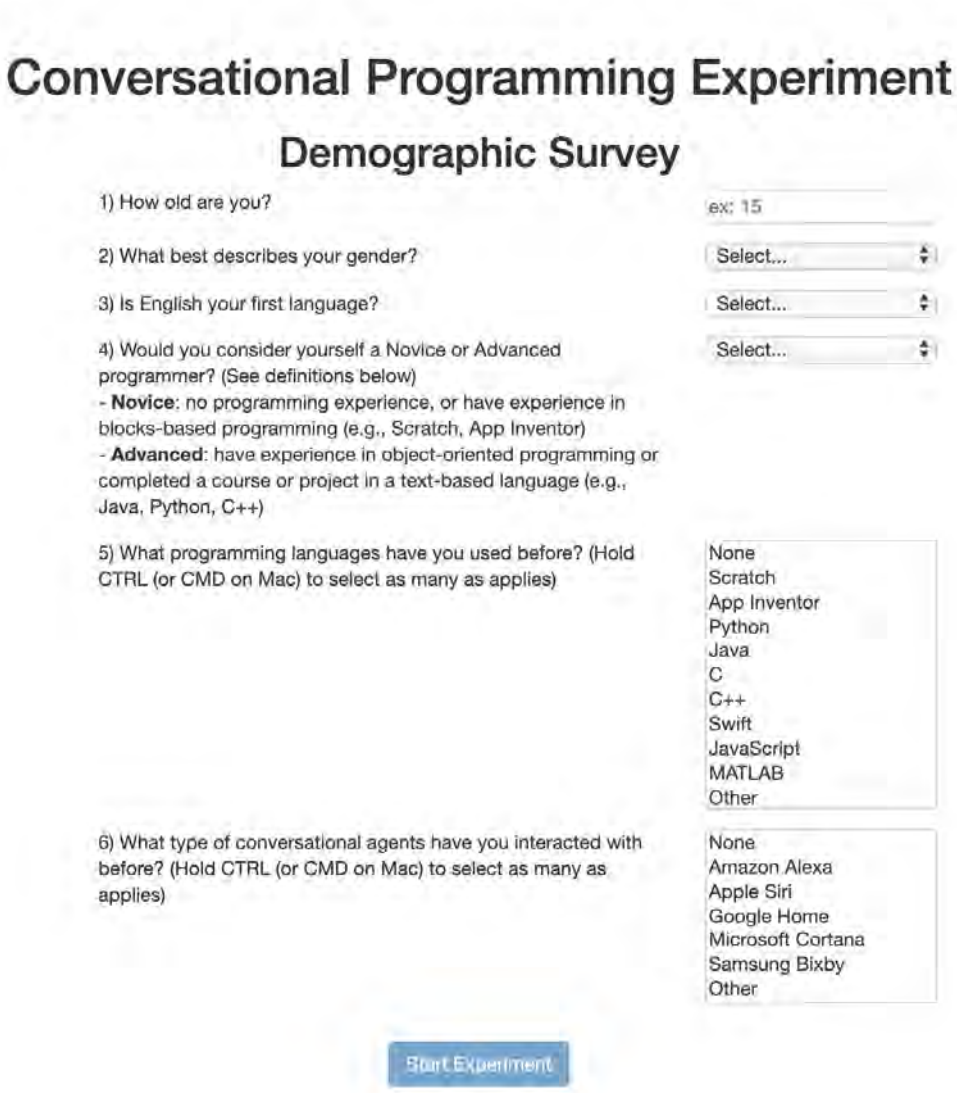
Survey

If you agree to participate in this experiment, please proceed to the next page to complete a brief survey.

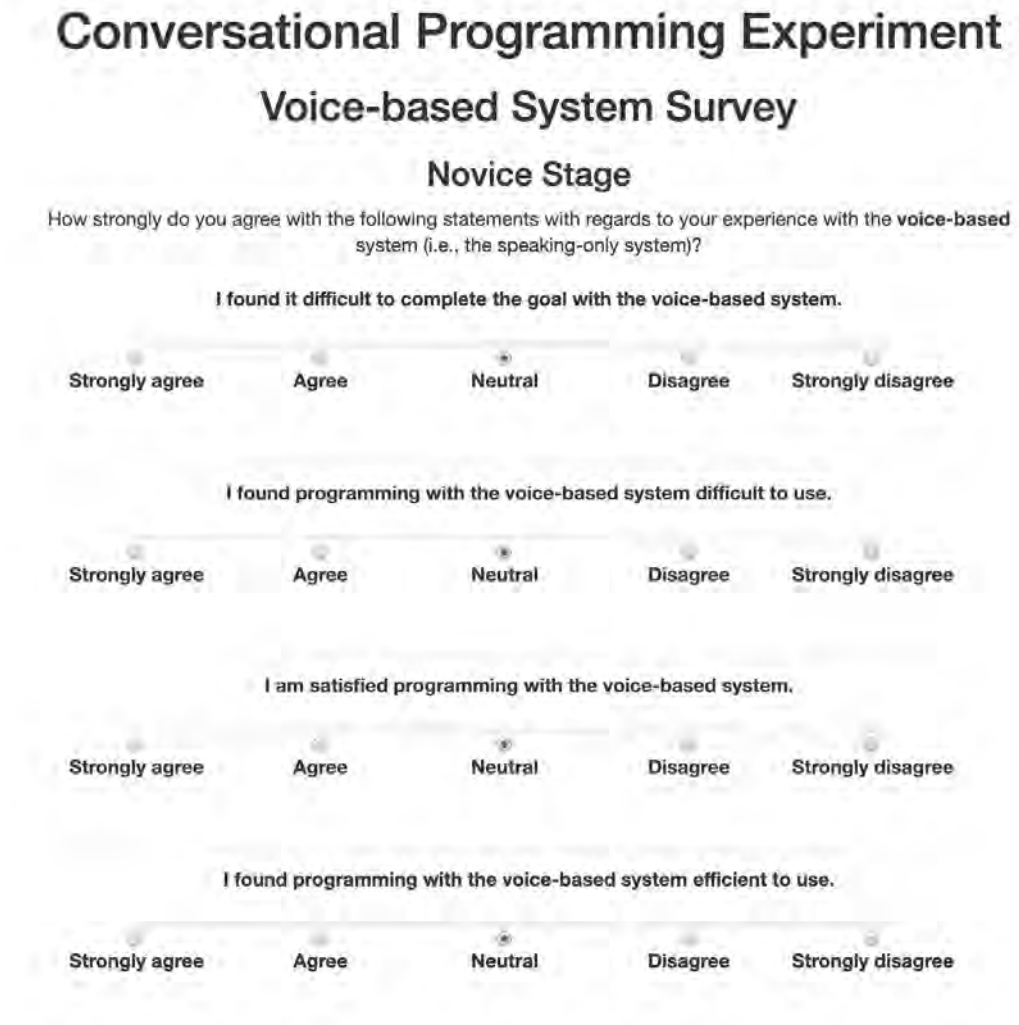
Begin Survey

A.2 Surveys

A.2.1 Demographic Survey



A.2.2 System Survey



Conversational Programming Experiment

Voice-based System Survey

Novice Stage

How strongly do you agree with the following statements with regards to your experience with the **voice-based** system (i.e., the speaking-only system)?

I found it difficult to complete the goal with the voice-based system.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I found programming with the voice-based system difficult to use.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I am satisfied programming with the voice-based system.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I found programming with the voice-based system efficient to use.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

Voice-based System Survey

Novice Stage

Answer the following questions with regards to your experience with the **voice-based** system (i.e., the speaking-only system).

What did you like about programming with the voice-based system?

I liked...

What was frustrating about programming with the voice-based system? How could we make it less frustrating?

You could improve the system by...

What did you wish you could say to the agent? What didn't the agent understand?

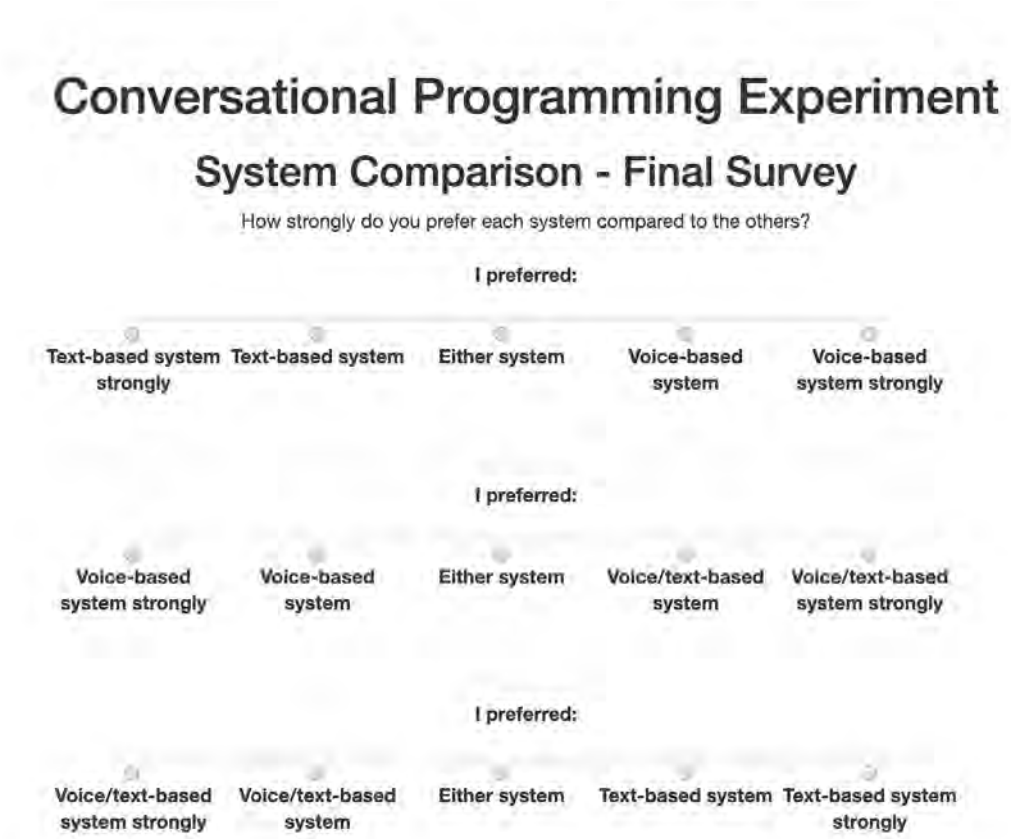
I wish I could have said things like...

What features can we add, change, or remove to make the system better?

You could add...

Next

A.2.3 Final Survey



How strongly do you agree with the following statements?

I think programming with just voice is easier than programming with just text.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I think programming with just text is easier than programming with both voice and text.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I think programming with just voice is easier than programming with both voice and text.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I think programming with just text is frustrating or hard.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I think programming with just voice is frustrating or hard.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I think programming with both voice and text is frustrating or hard.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I enjoyed the process of trying to complete the tasks.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I think being able to program using voice is useful.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I like being able to program using voice.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

If I could, I would continue to learn how to program using voice.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

I am a programmer.

Strongly agree | Agree | Neutral | Disagree | Strongly disagree

System Comparison - Final Survey

Answer the following questions with regards to your experience.

We want the agent to eventually be able to explain things about how it works. If you were to ask the agent any question, what would you ask it? Please list as many questions as you can think of.

I would ask the system...

What challenges did you run into while interacting with the agent?

Some things I found challenging were...

What questions do you have about the system?

I was wondering...

Next

A.3 Stages

A.3.1 Practice Stage

Practice Stage

- You're going to learn how to converse with the conversational agent and **program a simple *Hello World* program** in three slightly different ways: by speaking, by typing, and by speaking and/or typing (you choose).
- The **order** in which you complete the tasks will be **randomized** (you may be asked to complete the task using just voice first, voice/text first or just text first.)
    - In the **voice** system, you can click a button or press the spacebar to say commands to the agent.
    - In the **text** system, you can type out commands to the agent.
    - In the **voice + text** system, you can click a button or press the spacebar to say commands or type out commands to the agent.
- The following video shows the program you are going to create for the practice stage.

- **Guidance** will be shown on the **left side of the screen**. This will change depending on the context of the conversation.
- After each practice stage, you may play around with the system in a "sandbox mode" to see how it works.

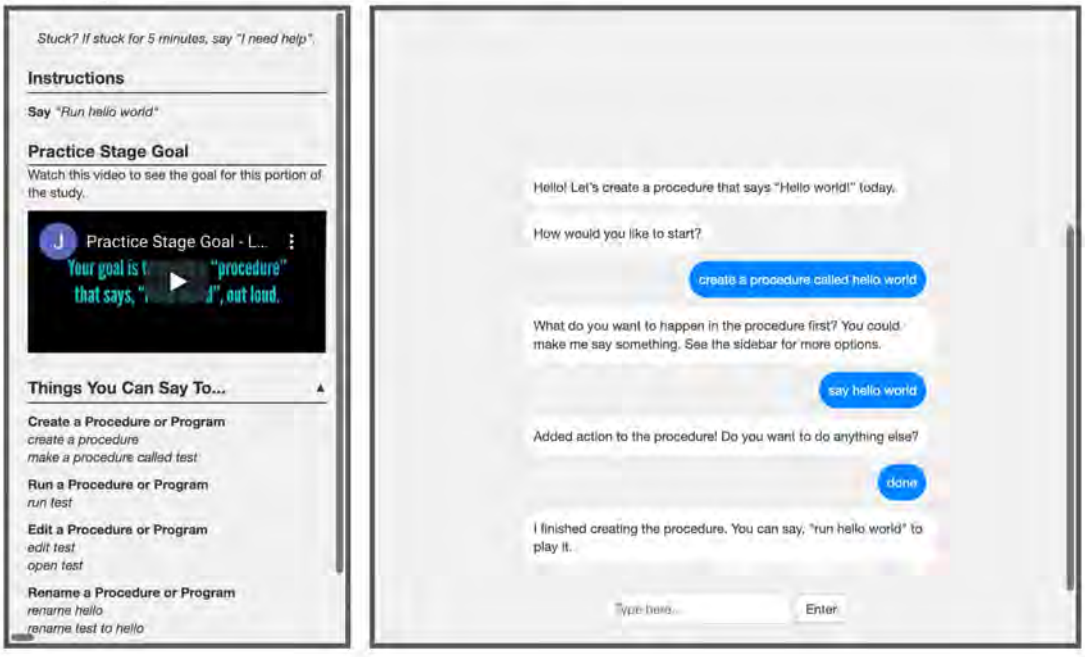
Begin Practice

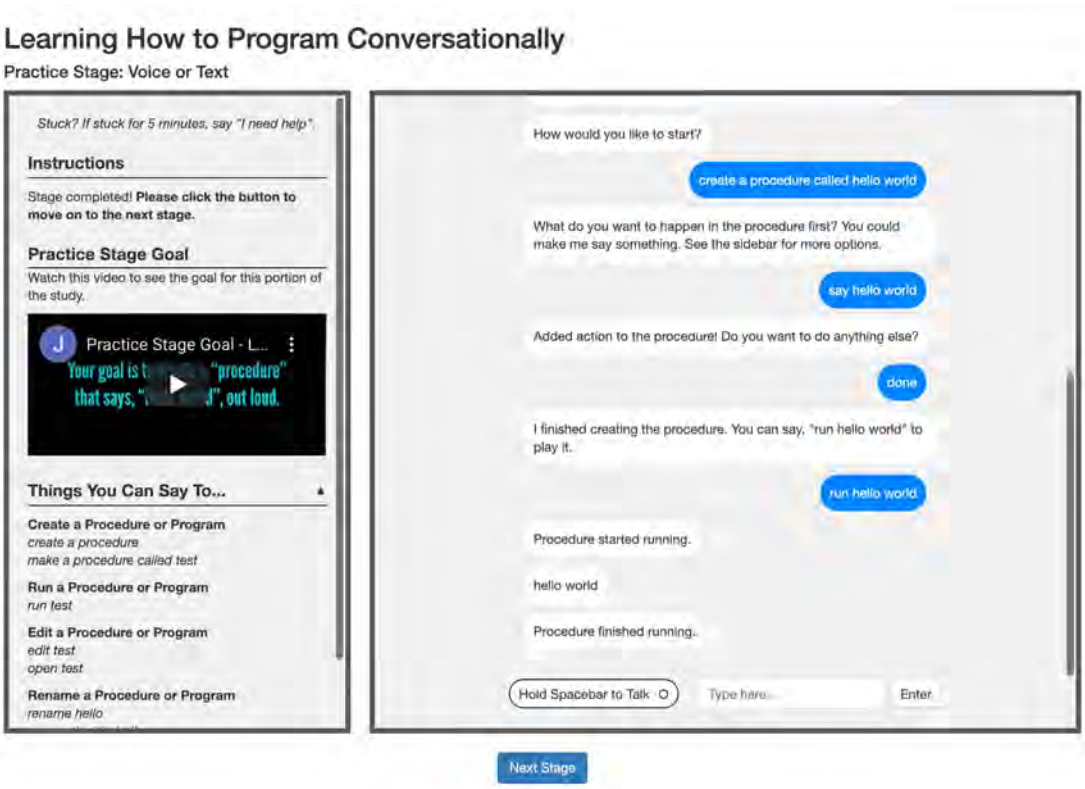
Learning How to Program Conversationally

Practice Stage: Voice

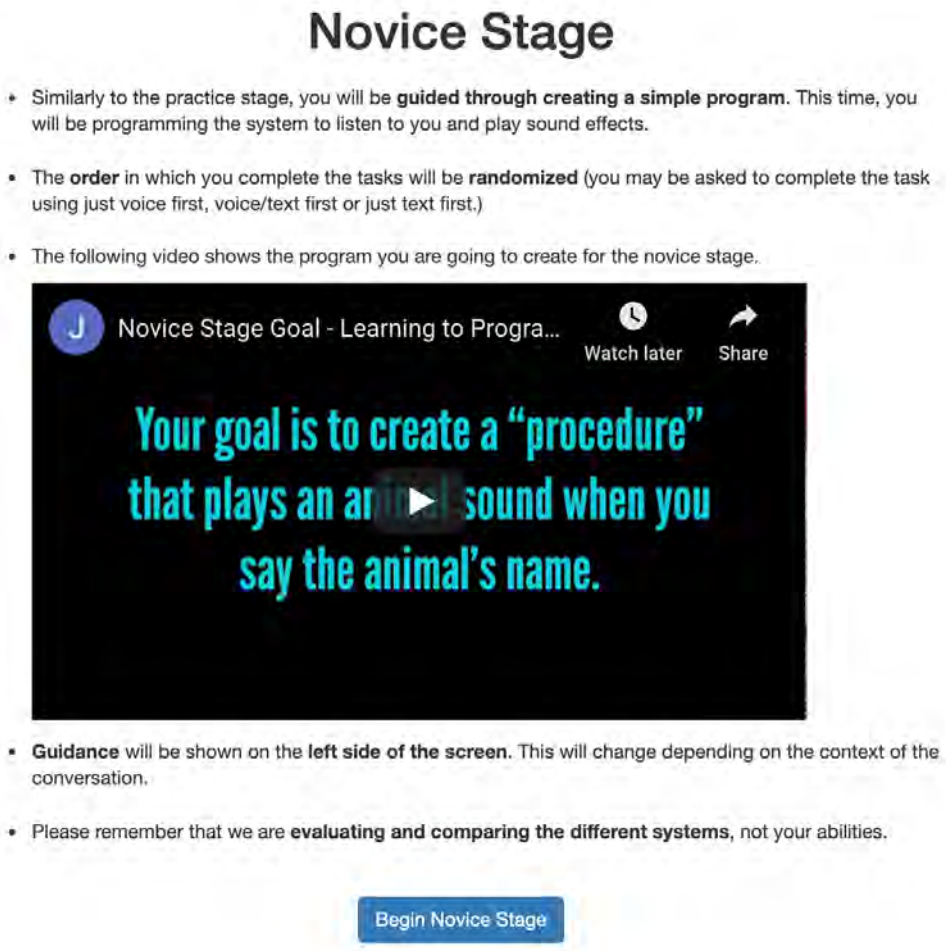
Learning How to Program Conversationally

Practice Stage: Text




Learning How to Program Conversationally
Practice Stage: Voice or Text
Stuck? If stuck for 5 minutes, say "I need help".
Instructions
Stage completed! Please click the button to move on to the next stage.
Practice Stage Goal
Watch this video to see the goal for this portion of the study.
Practice Stage Goal - L...
Your goal is t "procedure" that says, " ", out loud.
Things You Can Say To...
Create a Procedure or Program
create a procedure
make a procedure called test
Run a Procedure or Program
run test
Edit a Procedure or Program
edit test
open test
Rename a Procedure or Program
rename hello
How would you like to start?
create a procedure called hello world
What do you want to happen in the procedure first? You could make me say something. See the sidebar for more options.
say hello world
Added action to the procedure! Do you want to do anything else?
done
I finished creating the procedure. You can say, "run hello world" to play it.
run hello world
Procedure started running.
hello world
Procedure finished running.
Hold Spacebar to Talk
Type here...
Enter
Next Stage

A.3.2 Novice Stage

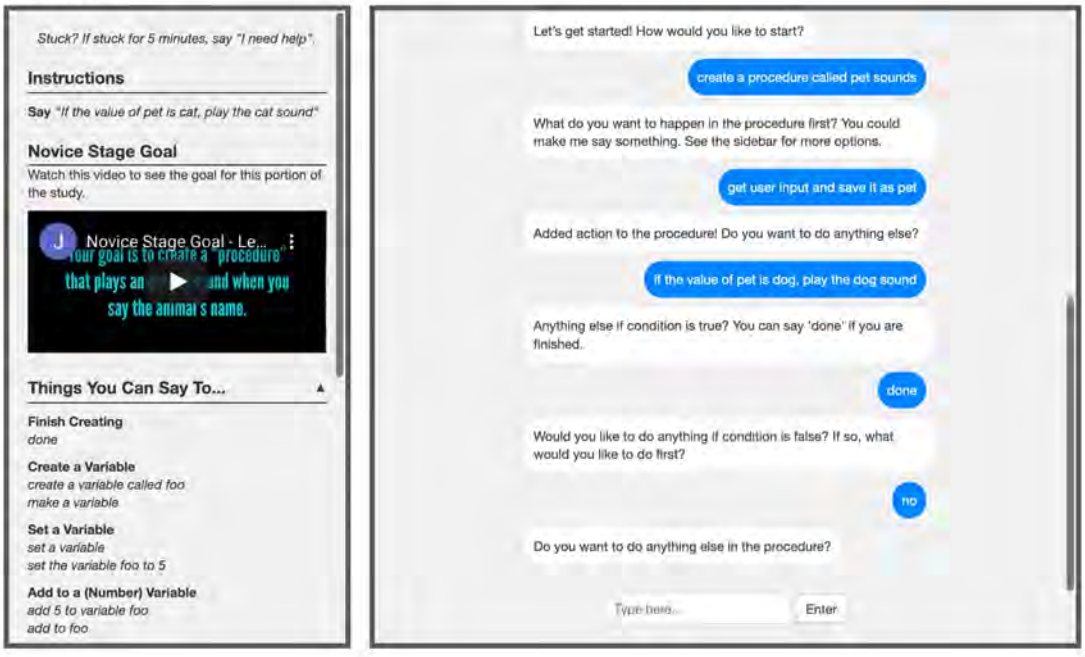


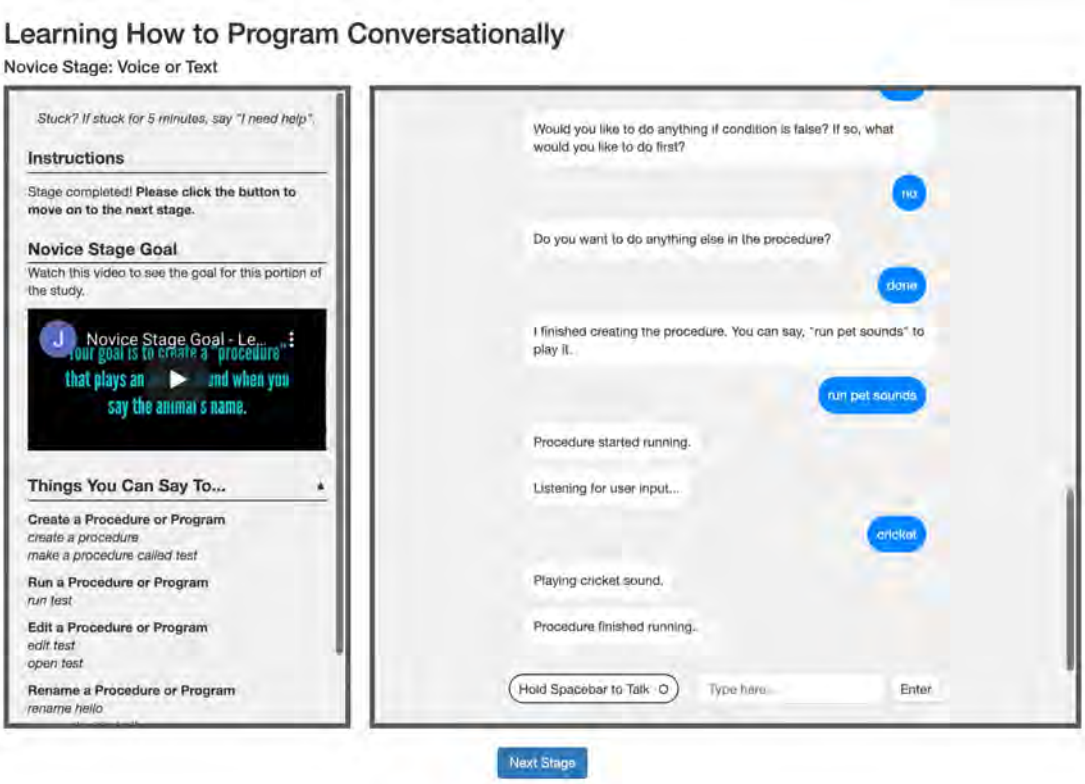
Learning How to Program Conversationally

Novice Stage: Voice

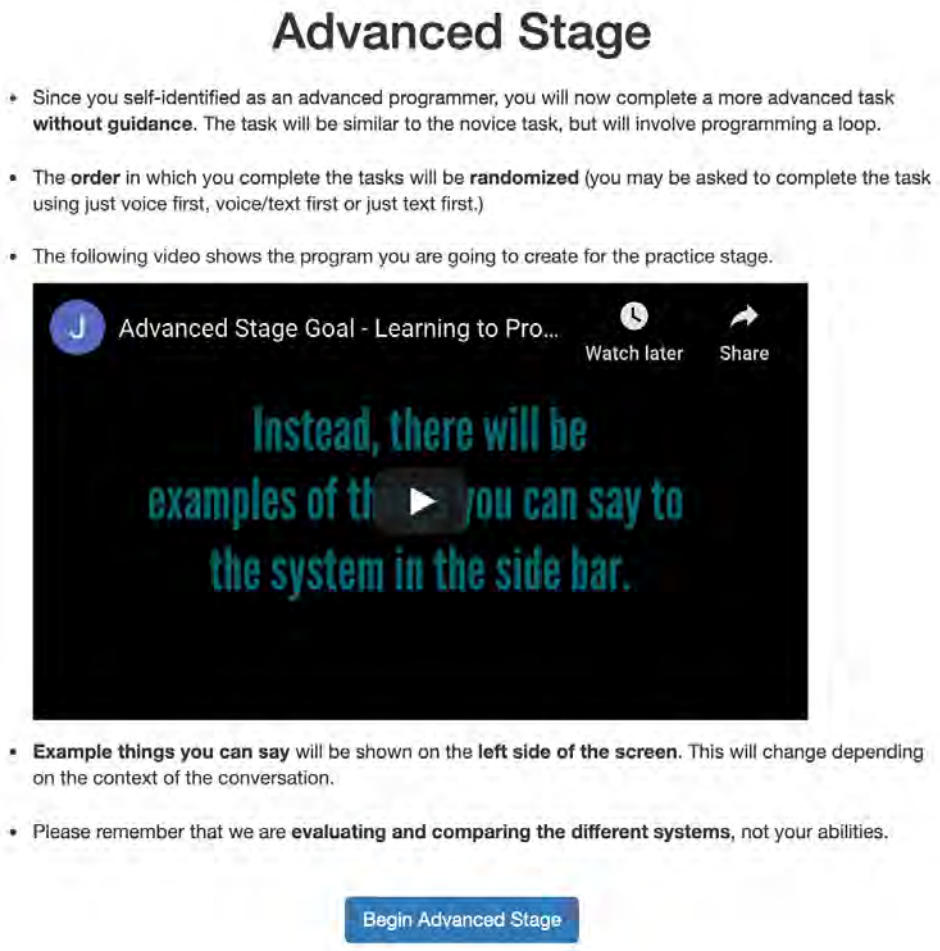
Learning How to Program Conversationally

Novice Stage: Text



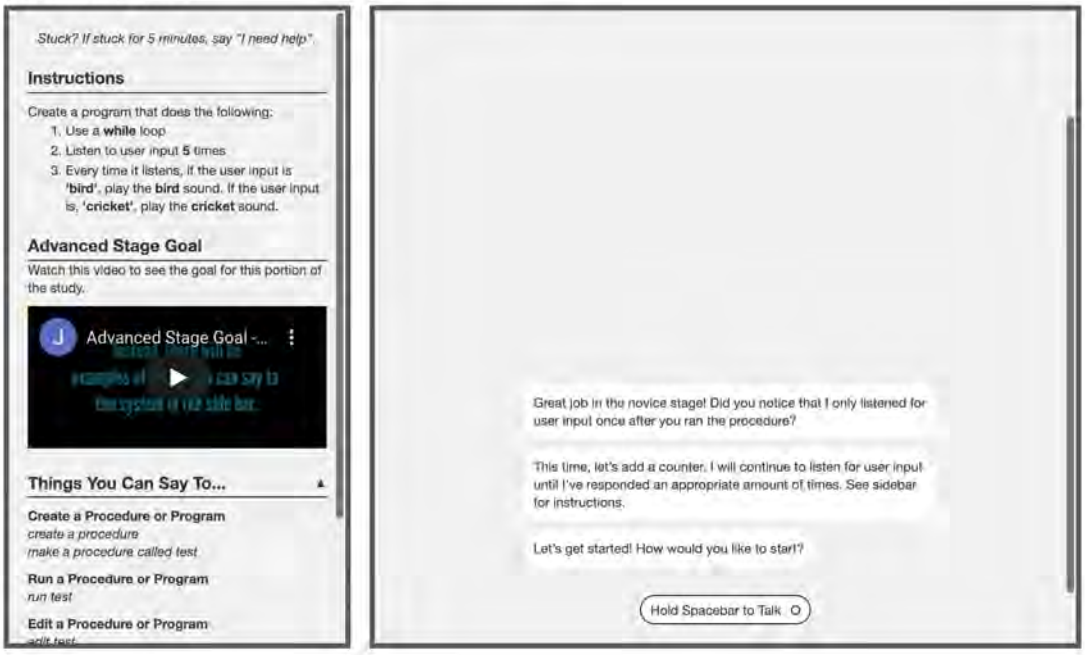

Learning How to Program Conversationally
Novice Stage: Voice or Text
Stuck? If stuck for 5 minutes, say "I need help".
Instructions
Stage completed! Please click the button to move on to the next stage.
Novice Stage Goal
Watch this video to see the goal for this portion of the study.
Novice Stage Goal - Le...
Your goal is to create a "procedure" that plays an and when you say the animal s name.
Things You Can Say To...
Create a Procedure or Program
create a procedure
make a procedure called test
Run a Procedure or Program
run test
Edit a Procedure or Program
edit test
open test
Rename a Procedure or Program
rename hello
Would you like to do anything if condition is false? If so, what would you like to do first?
no
Do you want to do anything else in the procedure?
done
I finished creating the procedure. You can say, "run pet sounds" to play it.
run pet sounds
Procedure started running.
Listening for user input...
cricket
Playing cricket sound.
Procedure finished running.
Hold Spacebar to Talk
Type here...
Enter
Next Stage

A.3.3 Advanced Stage



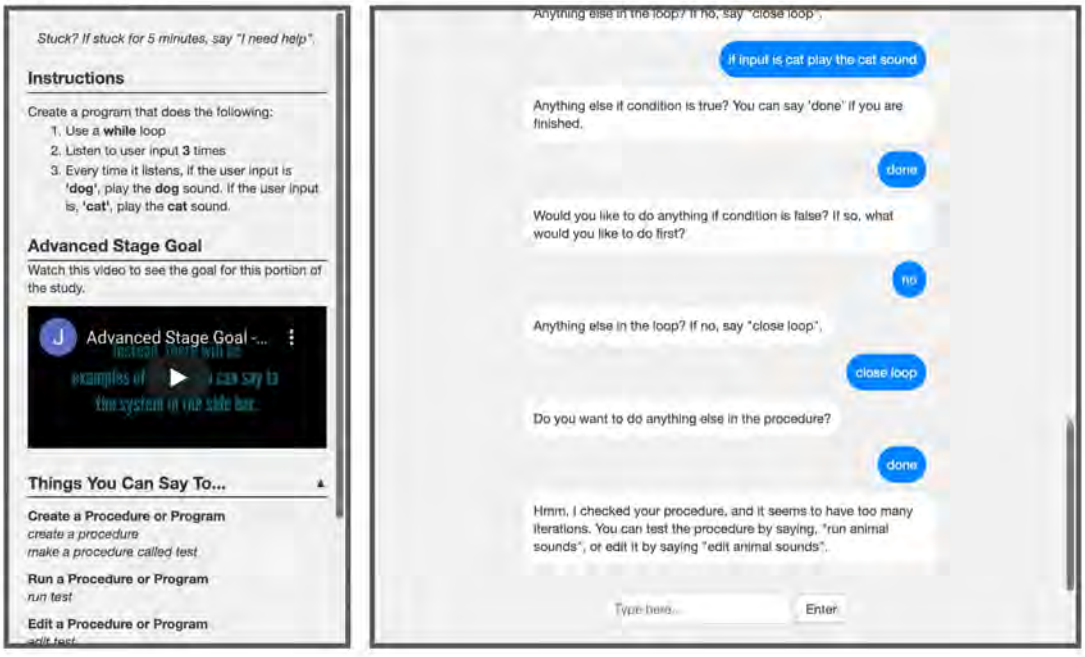
Learning How to Program Conversationally

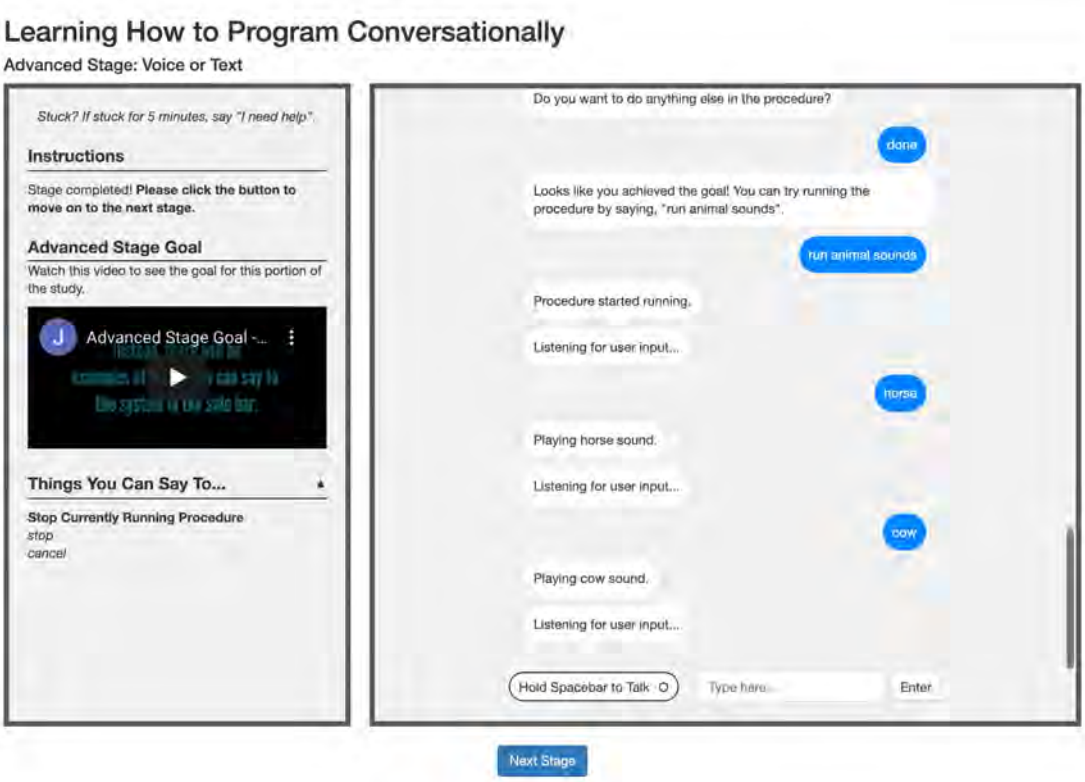
Advanced Stage: Voice



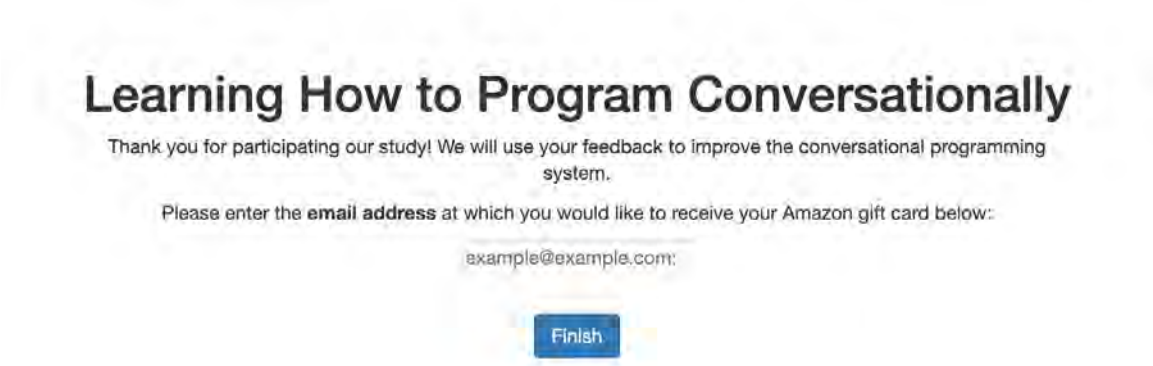
Learning How to Program Conversationally

Advanced Stage: Text




Learning How to Program Conversationally
Advanced Stage: Voice or Text
Stuck? If stuck for 5 minutes, say "I need help".
Instructions
Stage completed! Please click the button to move on to the next stage.
Advanced Stage Goal
Watch this video to see the goal for this portion of the study.
Advanced Stage Goal -...
Things You Can Say To...
Stop Currently Running Procedure
stop
cancel
Do you want to do anything else in the procedure?
done
Looks like you achieved the goal! You can try running the procedure by saying, "run animal sounds".
run animal sounds
Procedure started running.
Listening for user input...
horse
Playing horse sound.
Listening for user input...
cow
Playing cow sound.
Listening for user input...
Hold Spacebar to Talk
Type here...
Enter
Next Stage

A.4 Gift Card



A.5 Thank You

Bibliography

[1] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, et al. Let's encrypt: An automated certificate authority to encrypt the entire web. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2473–2487, 2019.

[2] Amazon. Alexa skill blueprints. `https://blueprints.amazon.com/`.

[3] Michael Bayer. Sqlalchemy. *The architecture of open source applications*, 2:291–314, 2014.

[4] ST Bhosale, T Patil, and P Patil. Sqlite: Light database system. *International Journal of Computer Science and Mobile Computing*, 4(4):882–885, April 2015.

[5] Tom Bocklisch, Joey Faulkner, Nick Pawlowski, and Alan Nichol. Rasa: Open source language understanding and dialogue management. *CoRR*, abs/1712.05181, 2017.

[6] Daniel Braun, Adrian Hernandez Mendez, Florian Matthes, and Manfred Langen. Evaluating natural language understanding services for conversational question answering systems. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, pages 174–185, Saarbrücken, Germany, August 2017. Association for Computational Linguistics.

[7] Jessica Van Brummelen, Kevin Weng, Phoebe Lin, and Catherine Yeo. Convo: What does conversational programming need? an exploration of machine learning interface design, 2020.

[8] Chung-Cheng Chiu, Tara N. Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J. Weiss, Kanishka Rao, Katya Gonina, Navdeep Jaitly, Bo Li, Jan Chorowski, and Michiel Bacchiani. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[10] Ian Fette and Alexey Melnikov. The websocket protocol. RFC 6455, Internet Engineering Task Force, December 2011.

[11] Google. Introduction to the speech synthesis api, 2014. https://developers.google.com/web/updates/2014/01/Web-apps-that-talk-Introduction-to-the-Speech-Synthesis-API, Last accessed on 2020-02-24.

[12] Google. Google cloud speech-to-text. https://cloud.google.com/speech-to-text, 2020. Last accessed on 2020-04-10.

[13] Google. Learn about conversation. https://designguidelines.withgoogle.com/conversation/conversation-design/learn-about-conversation.html, 2020. Last accessed on 2020-04-10.

[14] Herbert P Grice. Logic and conversation. In *Speech acts*, pages 41–58. Brill, 1975.

[15] Serenade Labs Inc. Serenade. http://serenade.ai/.

[16] Abhinav Jain, Minali Upreti, and Preethi Jyothi. Improved accented speech recognition using accent embeddings and multi-task learning. In *Proc. Interspeech 2018*, pages 2454–2458, 2018.

[17] Bret Kinsella. Nearly 90 million u.s. adults have smart speakers, adoption now exceeds one-third of consumers. Technical report, Voicebot, April 2020.

[18] Yaniv Leviathan and Yossi Matias. Google duplex: An ai system for accomplishing real-world tasks over the phone. https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html, May 2018.

[19] Mady Mantha. Introducing diet. https://blog.rasa.com/introducing-dual-intent-and-entity-transformer-diet-state-of-the-art-performance-on-a-lightweight-architecture/, 2020. Last accessed on 2020-04-10.

[20] Anna Nowogrodzki. Speaking in code: how to program by voice. https://www.nature.com/articles/d41586-018-05588-x, Jul 2018.

[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[22] Tina Quach. Agent-based programming interfaces for children: Supporting blind children in creative computing through conversation. Master's thesis, MIT Media Lab, May 2019.

[23] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[24] Rasa. Slot filling. `https://legacy-docs.rasa.com/docs/core/slotfilling/`, 2019. Last accessed on 2020-04-10.

[25] Viktor Schlegel, Benedikt Lang, Siegfried Handschuh, and André Freitas. Vajra: Step-by-step programming with natural language. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, IUI '19, pages 30–39. ACM, 2019.

[26] Lars St, Svante Wold, et al. Analysis of variance (anova). *Chemometrics and intelligent laboratory systems*, 6(4):259–272, 1989.

[27] Gail M Sullivan and Anthony R Artino Jr. Analyzing and interpreting data from likert-type scales. *Journal of graduate medical education*, 5(4):541–542, 2013.

[28] David R Thomas. A general inductive approach for analyzing qualitative evaluation data. *American journal of evaluation*, 27(2):237–246, 2006.

[29] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[30] Jessica Van Brummelen. Tools to create and democratize conversational articial intelligence. Master's thesis, MIT App Inventor, June 2019.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[32] Ye-Yi Wang, Alex Acero, and Ciprian Chelba. Is word error rate a good indicator for spoken language understanding accuracy. In *IEEE Workshop on Automatic Speech Recognition and Understanding*. Institute of Electrical and Electronics Engineers, Inc., January 2003.

[33] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[34] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.