Programming by Voice: Exploring User Preferences and Speaking Styles

Sadia Nowrin
snowrin@mtu.edu
Michigan Technological University
Houghton, Michigan, USA

Keith Vertanen
vertanen@mtu.edu
Michigan Technological University
Houghton, Michigan, USA

ABSTRACT

Programming by voice is a potentially useful method for individuals with motor impairments. Spoken programs can be challenging for a standard speech recognizer with a language model trained on written text mined from sources such as web pages. Having an effective language model that captures the variability in spoken programs may be necessary for accurate recognition. In this work, we explore how novice and expert programmers speak code without requiring them to adhere to strict grammar rules. We investigate two approaches to collect data by having programmers speak either highlighted or missing lines of code. We observed that expert programmers spoke more naturally, while novice programmers spoke more syntactically. A commercial speech recognizer had a high error rate on our spoken programs. However, by adapting the recognizer's language model with our spoken code transcripts, we were able to substantially reduce the error rate by 27% relative to the baseline on unseen spoken code.

CCS CONCEPTS

• **Human-centered computing** → **Accessibility**.

KEYWORDS

Voice Programming, Speech Recognition, Voice User Interfaces, Accessibility

ACM Reference Format:
Sadia Nowrin and Keith Vertanen. 2023. Programming by Voice: Exploring User Preferences and Speaking Styles. In *ACM conference on Conversational User Interfaces (CUI '23), July 19–21, 2023, Eindhoven, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3571884.3597130

1 INTRODUCTION

Software development is a text input and text editing intensive activity typically performed using a keyboard and a mouse. Such reliance can present a significant barrier for individuals with motor impairments who want to learn to program or pursue careers in technology-related fields. This can be especially discouraging for novices who are trying to enter the field of programming. Additionally, even experienced software engineers can develop motor impairments such as a repetitive stress injury (RSI) due to the prolonged use of these input devices. Using voice to create code can be an alternative approach to traditional text-based programming, potentially improving the accessibility and efficiency of programming for individuals with motor impairments.

One possible architecture for a voice programming system would consist of two components. The first component would be a speech recognizer that converts a user's code utterances into the literal words spoken (e.g. "increment num words by one"). The second component would be a machine translation model that converts the recognized text into the target language and preferred coding style (e.g. "`numWords++;`"). This translation model could additionally be guided by a model that is aware of the target programming language grammar and any constraints introduced by the current location in the program (e.g. which variables are in scope). This two-part architecture has the advantage that different machine translation models could be swapped in for different target programming languages. We anticipate the speech recognition component would require minimal changes for different programming languages since the acoustic properties and vocabulary used may strongly overlap between languages. This may be particularly true when users speak code naturally (i.e. without explicitly dictating the literal characters needed by a given language). We focus on the first component in this paper.

Classically, a speech recognizer works by first converting a user's spoken sounds into possible words using an acoustic model. It then searches for the most probable sequence of words guided by a language model. Modern neural speech recognizers may use a more end-to-end approach, taking sound as the input and directly outputting letters or words. However, neural recognizers often still incorporate a language model to rescore hypotheses since a language model can be trained on large amounts of just text. While there is a wealth of data to train accurate language models for tasks such as writing emails, no such data exists for speaking code. Collecting a large amount of data for a new domain such as spoken code is challenging due to the variability in how programmers speak code and the complexity of programming syntax.

As a first step to supporting flexible and robust code input by voice, we explore variations in spoken code, aiming to understand different speaking styles and the potential ambiguities that can arise. Our primary goals were to 1) capture a range of user variability, and 2) develop a dataset of spoken code in order to train a language model to improve speech recognition accuracy. We wanted to better understand if programmers speak code naturally or in a literal manner, whether they skip symbols, spell things out, or explicitly denote cases. For example, consider the Java statement: "`items[i] = 5;`". Would programmers explicitly speak symbols such as square

CUI '23, July 19–21, 2023, Eindhoven, Netherlands

ACM ISBN 979-8-4007-0014-9/23/07.
https://doi.org/10.1145/3571884.3597130

brackets or semicolons? Would they speak in a more pseudo-code style? For example, the programmer could just say "assign items at location i to five". The latter is attractive as it might allow learning a shared vernacular for common statements in different languages, allowing programmers to more easily learn or switch between languages.

While large language models (LLMs) [5, 6, 11, 17] have the ability to generate a code block from a text prompt, they may not always be accurate or correct [21]. This is because LLMs heavily rely on the training data used to build them. If the training data is not representative of the types of code being generated or is limited in scope, the LLM may not perform well in accurately generating code. For instance, a general-purpose programming language-trained model may struggle to generate accurate code for specialized domains like machine learning or cryptography. In such cases, it may be necessary to manually correct individual lines or sections of the generated code. While LLM-generated code can be a powerful tool for programmers, it may not align with programmers' preferred workflow or coding style. Depending on the task at hand, a programmer may prefer to write code line-by-line or by breaking a task down into smaller, more manageable pieces. Additionally, programmers may not always write code in a linear fashion. They may jump back and forth between different sections of code or may modify previously written lines of code. Speaking code line-by-line may allow programmers to have more control over the code they are creating and ensure that it meets their specific requirements.

LLMs can help automate routine or repetitive coding tasks, freeing up programmers' time to focus on more complex or creative aspects of coding. However, for novices, relying too heavily on LLMs to generate a whole code block could limit their ability to learn coding concepts and develop problem-solving skills. For learners, it may be more beneficial to learn to code line-by-line, building up their knowledge gradually and reinforcing their understanding of each individual concept before moving on to more complex topics. Additionally, for experts, relying too heavily on LLMs to generate code could lead to a reduction in their coding skills and ability to write code manually. This could be detrimental to their long-term career prospects and could also limit their ability to identify and fix errors in auto-generated code. In this work, we aim to investigate how programmers speak individual lines or small segments of code.

A secondary goal of our user studies was to compare how participants spoke missing lines of code (the target line of code was blank) versus how they spoke highlighted lines of code (the target line of code was specified). Speaking missing code is closer to the actual cognitive task of programming by voice while speaking highlighted lines would likely be quicker and easier but could result in a fundamentally different speaking style. The purpose of evaluating these two scenarios was to see which one would be most effective for scaling up the data collection process. To understand if our collected data can enhance the speech recognition system's accuracy, we performed offline experiments using a commercial speech recognizer (Google Cloud Speech-to-Text). We adapted the language model on our collected spoken code, which helped reduce the word error rate (WER) by 27% relative. This shows the promise of improving recognition accuracy via changes to the speech recognizer's underlying language model.

Our work makes the following contributions:

(1) Our primary contribution is that we provide novel perspectives into how diverse programmers speak code (i.e. without teaching them a specific grammar). This can help inform the design of more effective and robust voice-based systems.
(2) We conduct the first comparison of two possible approaches in collecting spoken code. These approaches involved speaking a missing line of code and speaking a highlighted line of code. We further identify which approach is more effective in capturing the necessary variations and developing accurate language models for spoken code.
(3) We demonstrate how our collected data can be used to improve the accuracy of a speech recognizer, highlighting the importance of improving recognition accuracy in this domain.
(4) We release the first data set containing individual lines of code and the transcripts of how programmers spoke those lines. This provides a valuable resource for the research community in developing accurate and robust language models for spoken code.

2 RELATED WORK

Creating a voice programming system is challenging as it is much more than simply dictating lines of code. Existing voice programming systems such as Talon[1] and Vocola[2] require memorizing a set of voice commands. Such an approach can be useful for dictating programming statements, but it may become challenging for complex programs, as the number of commands increases. Using natural language for voice-based programming could be less restrictive as it enables the use of conversational language which may be learned and spoken more easily. In a previous study [14], we conducted an interview with seven motor-impaired programmers to understand their perceptions regarding programming by voice. During the interview, programmers expressed their frustration with current systems that require them to memorize a large number of commands. Instead, they expressed a desire to speak code in a flexible and natural manner with one programmer stating: "I guess it would be more similar to the experience of pair programming with someone." The use of natural language may also reduce the learning curve required to understand spoken programming language, making it easier for novices to learn code by voice.

2.1 Command-based Voice Programming

Arnold et al. [1] designed a command-based voice programming system called VocalGenerator. VocalGenerator takes a Context Free Grammar (CFG) and a voice vocabulary for a programming language as input and generates a programming environment in which users can write programs by voice. The system is no longer being developed.

Maloku and Pllana [12] developed HyperCode which enables coding in Java with voice commands in IntelliJ IDEA, a commercial Java Integrated Development Environment (IDE). In addition, HyperCode allows users to create their own custom voice commands. In a user study, coding with a combination of keyboard, mouse, and

[1] https://talonvoice.com/
[2] http://vocola.net/

voice using Hypercode was faster (average 46 seconds) compared to only keyboard and mouse (average 65 seconds) and only voice input (average 84 seconds).

Rosenblatt et al. [18] conducted a Wizard of Oz study to explore the commands programmers might use to write code. They also developed a working prototype named VocalIDE. They evaluated VocalIDE with participants having upper limb motor impairments. The efficiency of the system was limited by inadequate speech recognition accuracy.

Mancodev [13] is an IDE designed to enter JavaScript by voice. The system was evaluated with motor-impaired programmers, all participants mentioned that speech recognition was slow and error-prone.

Wagner et al. [24] developed Myna to make block-based visual programming language accessible. Myna is a voice-driven interface designed to enable motor-impaired children to learn to program in Scratch[3]. In an evaluation, Myna took 15.6 seconds less time on average than the mouse and keyboard [23]. However, non-native English speakers made more errors while using Myna compared to native English speakers.

Okafor and Ludi [15] also explored programming by voice in a block-based visual programming language. They evaluated the use of Google's Blockly[4] language by people with upper limb motor impairments. According to their evaluation, the system was easy to use but users found it hard to learn the predefined commands. The authors concluded that more accurate speech recognition is required as short commands like "in" or "up" were misrecognized 70% of the time.

Previous studies have focused on developing command-based systems for voice programming, where programmers were required to learn specific grammar. Our approach differs in that we aim to explore how programmers naturally speak code without being constrained by grammar rules. We aim to understand the diverse ways in which programmers speak code and did not want to impose any limitations on their language usage.

2.2 Natural Language-Based Voice Programming

Researchers have also investigated natural language systems for voice programming. Price et al. conducted a Wizard of Oz study to explore how people would use a voice interface for programming [16]. The authors observed that novice programmers struggled to describe their programs. But they were really excited about the natural language programming interface.

Desilets [8] conducted a study to understand the challenges involved in programming by voice such as dictating punctuation and variable names with abbreviated words and items in mixed case. The author later developed a tool named VoiceGrip that enabled users to speak code using a pseudo-code syntax that was then translated into native code. The system's capabilities were restricted to a manually created database containing mapping from native symbols (e.g. "<") to pseudo symbols (e.g. "`less than`") and some predefined rules to understand programming constructs.

[3]https://scratch.mit.edu
[4]https://developers.google.com/blockly

Brummelen et al. [22] conducted a user study where participants completed novice and advanced programming tasks using only voice input, using only text input, or using a combination of both. They found that novices appreciated the use of natural language to enter the program, supporting our belief that developing a system allowing users to speak code naturally is a worthwhile goal.

Begel and Graham [3, 4] conducted a study to investigate how programmers read Java code written on a piece of paper. The authors found that all programmers spoke in a similar way despite their programming experience but their speaking style varied a lot depending on the programming construct. Based on their study, they designed a system called Spoken Java using a rule-based approach to recognize spoken code. The system is based on a lexical analyzer, which breaks down spoken commands into tokens, and a semantic analyzer, which uses contextual information to determine the meaning of the tokens. The author expressed concerns that their approach might not reflect all variations in spoken code as people might speak differently when they dictate code from scratch as opposed to reading a pre-written bit of code aloud. This motivated our investigation of collecting spoken code via two methods: one where the speaker sees the code while speaking and another where the speaker speaks code without seeing it.

Very few studies have been conducted on the natural language use of programming by voice and no robust system currently exists. One of the challenges with natural language-based programming is that the system needs to be able to interpret a wide range of spoken language variations. This is important because not everyone speaks code in the same way, and a rigid system would limit the number of people who could effectively use the spoken programming language. We focus on a data-driven approach, allowing users to speak code in a flexible and natural manner, similar to how you might speak code in pair programming. Additionally, we investigate two data collection approaches aiming to capture enough variability to train a better language model for spoken code. We believe our work is a first step toward creating a robust and accessible voice programming system.

2.3 Speech Recognition in Domains with Strict Syntax

Advancements in speech recognition technology have made it applicable in a wide range of domains that require special symbols or strict syntaxes, such as mathematics, SQL queries, and the legal field. One recent study by Song et al. proposed a novel architecture named "SpeechSQLNet" that can directly translate human speech into SQL queries without the need for external Automatic Speech Recognition (ASR). Their research showed that the end-to-end architecture outperformed the cascaded style of speech to SQL, which first converts speech signals into transcripts with an ASR system and then conducts downstream text-to-SQL conversion.

Several software programs exist in the mathematics domain that provides accessibility for motor-impaired users to dictate mathematical equations and formulas using speech recognition technology, such as Math Speak & Write [9], MathSpeak [20], TalkMaths [25]. However, few studies have explored the use of voice-based transcription in the legal domain [19]. Programming languages have strict syntax and grammar rules that must be followed in order for

the code to be executed correctly. For people with disabilities who may have difficulty typing or using a keyboard, speech recognition technology can potentially make programming more accessible by allowing them to dictate code instead of typing it.

3 USER STUDY

To collect examples of people speaking programs to a hypothetical system, we conducted a remote user study. Participants completed the study using their own computers and microphone. Novice programmers received extra credit in a course while expert programmers received a $20 Amazon gift card for completing the study.

3.1 Procedure

We created two sets of programs, each consisting of 20 identical Java programs. In the first one, the odd-numbered programs had missing lines(s) and even-numbered programs had highlighted line(s). In the second one, we reversed this. Participants were randomly assigned to the first version or the second one. Out of the 20 Java programs, 16 had single missing or single highlighted lines, while four had multiple missing or highlighted lines of code. The single lines of code included various code constructs, such as function calls, if-else statements, loops, input-output statements, arrays, comments, decrement operations, mathematical calculations, and variable declarations. The multiline code snippets included an if-else block, a for-loop block, a multiline comment, and a function body. Table 1 shows an example of a Java program with a missing line and highlighted line as used in our study. On average, the single-line and the multi-line programming statements were 42 and 53 characters in length respectively. All 20 programs were different and ranged from 6 to 16 lines.

Using a web application, participants first signed a consent form and filled out a demographic questionnaire. At the beginning of the experiment, participants were instructed as follows:

> a) "Imagine you are a programmer who has an injury. Typing on the keyboard is difficult for you. How would you speak code to an intelligent computer program that could convert your speech into code?"
>
> b) "There are no rules in how you speak code."

For each program, participants recorded themselves speaking either the missing line(s) or highlighted line(s) of each program. They did not receive any feedback while speaking but could play back their recording afterward. Participants could re-record the audio for a given program as many times as they wanted; we only kept the last recording. Finally, they completed a questionnaire that asked about their experience during the study.

3.2 Participants

We recruited 12 novice programmers (7 female, 5 male) from introductory Java courses. We recruited 12 expert programmers (2 female, 10 male) through word-of-mouth. Experts were required to have at least four years of programming experience and be familiar with Java. Experts' programming experience ranged from five to 23 years.

All participants in both studies were native English Speakers. As for their usage of speech interfaces, 8% of novices and 18% of experts strongly agreed or agreed that they frequently used speech interfaces. We asked participants how frequently they wrote programs. 58% of novice participants and all of the expert participants strongly agreed or agreed that they frequently wrote programs. The exact questionnaire we used is available in our supplementary materials.

3.3 Data Analysis

We manually reviewed all collected recordings. Novice participants often submitted empty recordings for the multi-line tasks which we discarded. In total, we collected 224 audio files (192 for single lines, 32 for multi-lines) from the novices and 240 audio files (192 for single lines, and 48 for multi-lines) from the experts. We listened to each audio file and typed a verbatim word-by-word transcript of what the person said including spoken symbols, words, and spaces. For example, "string very large string two equals quotation mark world end quotation mark semicolon". For both single and multi-line programming statements, each recording is transcribed into a one-line transcript. The human transcripts and the associated programming statements are available in our supplementary materials.

4 USER STUDY RESULTS

Novices completed the experiment on average in 48 minutes (SD = 8.8) while experts took 45 minutes (SD = 26.2). We split the participants into novice and expert groups to analyze two measures: speaking style and speaking rate in both the missing and highlighted conditions. Additionally, we investigated the variations of speaking different programming constructs as well as the ambiguity in spoken code. Finally, we analyzed participants' subjective feedback.

4.1 Speaking style: Natural versus literal

The two authors independently judged the speaking style of each utterance in the transcripts and categorized each as either natural or literal. Prior to judging, authors discussed the criteria for judgment and the definition of natural and literal utterances. Utterances were considered natural if participants spoke most parts of the code using natural phrases (e.g. "start a comment") instead of literal adherence to the required characters (e.g. "forward slash forward slash"). The two judges did not see each other's ratings beforehand. Inter-rater reliability was very high (Cohen's kappa = 0.98), indicating an almost perfect agreement between the raters. To ensure consistency, the authors then discussed their judgment and resolved any disagreements.

A mixed analysis of variance (ANOVA) design was conducted to investigate the effects of experience level (novice versus expert) and conditions (missing versus highlighted) on the use of natural language in code dictation. The results revealed no significant interaction between experience level and condition ($F(1, 22) = 3.04$, $p = 0.09$), indicating that the effect of condition did not differ significantly between expert and novice programmers. Furthermore, there was no significant main effect of experience level on the use of natural language ($F(1, 22) = 1.82$, $p = 0.19$), with expert programmers using natural language slightly more often than novices (62% versus 45%, respectively). However, there was no significant main

Java program with a missing for-loop at line 8

```
1  //Calculates the sum of first n
2  //natural numbers using a for loop
3  public static void main(String[] args) {
4      int n, i;
5      int sum = 0;
6      Scanner scan = new Scanner(System.in);
7      n = scan.nextInt();
8
9          sum = sum + i;
10     }
11     System.out.println("Sum=" + sum);
12 }
```

Java program with a highlighted line at line 6

```
1 //This program prints numbers from 10 to
      1.
2 public static void main(String[] args) {
3     int largeNumCounter = 10;
4     while (largeNumCounter >= 1) {
5         System.out.println(largeNumCounter
              );
6             largeNumCounter -;
7     }
8     return 0;
9 }
```

Table 1: Example of two Java programs from our user study. The left program has a missing line. The right program has a highlighted line.

effect of condition on the use of natural language ($F(1, 22) = 0.21$, $p = 0.65$). These findings suggest that the use of natural language in code dictation is not significantly influenced by experience level and that both novice and expert programmers are similarly affected by the missing or highlighted conditions.

4.2 Verbalization by Programming Construct

We found wide variations in how certain programming constructs and some specific parts of code were verbalized. Most variations occurred when speaking method declarations, user-defined names, assignment operations, elements of an array, comments, abbreviated words, and punctuation.

4.2.1 Method signature and method call. The majority of the expert programmers verbalized different parts of the method such as return type, method name, and a parameter list naturally (e.g. "declare function public static return type integer name cube parameter int num"). In contrast, all but one novice programmer spoke methods in a literal way (e.g. "public static integer cube open paren int num close paren").

4.2.2 User-defined names. While dictating user-defined names such as variable and method names, two expert programmers mentioned naming conventions (e.g. "camel case very large string") while one expert spelled them out. Five other experts and three novices verbalized capitalization (e.g. "large capital n num capital c counter"). We also observed that expert programmers occasionally said the term "variable" while dictating a variable name.

4.2.3 Comments. As comments are written in natural language, we wanted to see how participants switched between the syntax required to denote a comment and the comment itself. All experts but only 30% of novices started a comment by speaking "open comment", "header comment" or "comment". The other novice participants spoke comments by verbalizing "slash slash" or "forward slash forward slash". 80% of the experts explicitly mentioned if their intent was a single-line or multi-line comment.

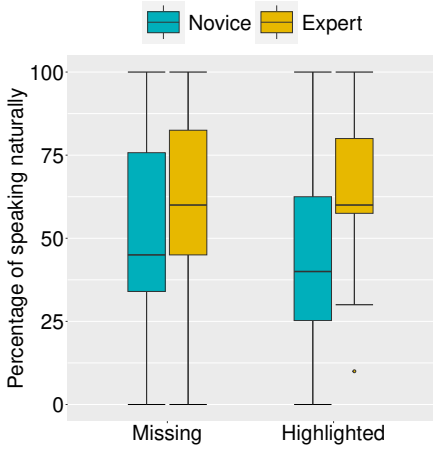


Figure 1: Comparison of speaking naturally in the missing and highlighted conditions.

4.2.4 Abbreviated words. Abbreviated words were either spoken as full words or spelled out. 80% of experts and novices verbalized the function `sqrt` as "square root" while others spelled it out. Two expert programmers verbalized the full form for an abbreviated variable name, for example, saying "number" instead of "num". Additionally, 90% of the experts and 40% of the novices verbalized the function `println` naturally as "print line" while others spoke it as "print l n".

4.2.5 Assignment operation. 70% of experts used phrases such as "is assigned" or "becomes" or "set" instead of verbalizing the "equal" symbol. for instance, one expert uttered the variable assignment `i = 1` as "i is assigned one". None of the novices used such natural phrases for assignment operation.

4.2.6 Multi-line code. We had programmers speak four multi-line programs. Our aim was to explore how participants might verbalize a block of code including, for example, specifying the transition to a new line. The majority of experts uttered phrases like "new

Metric	Condition / Speaking Style	Participant		Mixed ANOVA
		Novice	Expert	
Speaking rate (wpm)	missing	84.0±10.13 [55.4, 111.6]	95.2±16.72 [44.5, 137.7]	$F(1, 22) = 1.20$, $p = 0.29$
	highlighted	**92.7**±10.63 [74.3, 123.0]	**105.8**±16.32 [59.4, 148.1]	highlighted > missing, $p = 0.0003$
	natural	**98.9**±14.31 [66.8, 133.9]	99.5±17.62 [48.0, 136.4]	$F(1, 18) = 0.181$, $p = 0.68$
	literal	85.1±10.28 [60.0, 118.5]	89.1±12.97 [61.2, 135.3]	novice: natural > literal, $p = 0.02$
Proportion of speaking naturally (%)	missing	51.1±21.22 [0, 100]	40.8±20.69 [0, 100]	$F(1, 22) = 3.04$, $p = 0.09$
	highlighted	58.2±20.11 [0, 100]	36.7±17.29 [0, 100]	no significant pairs

Table 2: Numerical results from the user study including the statistical test details. Result format: mean ± 95% CI [min, max]

line", "enter", "begin body", or "start for loop body" to transition to a new line. None of the novices explicitly uttered any term for a new line, instead, they dictated the entire block of code as if it were on a single line (e.g. "static int cube int num end parenthesis curly bracket return num times num times num curly bracket").

4.2.7 Symbols and punctuation. We found significat variations in spoken punctuation. Experts who spoke naturally used some natural phrases for punctuation, for instance, two experts uttered "end line" instead of "semicolon". In addition, eight experts and three novices dictated array elements as "items at location i", "items at index i" or "items sub i" while the other participants dictated them in a literal way (e.g. "items open square bracket i close square bracket").

Participants used a variety of terms to refer to the quote symbol, including natural terms like "character" or "string" as well as more specific terms like "quote", "single quote", "opening quote", "end quote", "quotation marks". Additionally, participants spoke parentheses in varied ways such as "paren", "left parenthesis", "open parenthesis", and "close parenthesis". Variation also occurred in speaking brackets or braces, e.g. "left curly brace", "open curly brace", "close curly brace", "curly bracket" or "bracket".

In a few cases, participants uttered the same punctuation differently even in the same line. Participants who spoke in a literal way had a tendency to omit punctuation in both highlighted and missing conditions. We considered all single-line literal utterances and calculated the proportion of spoken punctuation in the transcript to actual punctuation in the target code. It is noteworthy that only participants who spoke syntactically at least 50% of the time in both conditions were included in the analysis, which consisted of eight novices and three experts. Overall, participants spoke parentheses 83% of the time in the highlighted condition versus 72% of the time in the missing condition. Interestingly, participants verbalized quotation marks 100% of the time in the highlighted condition but only 42% of the time in the missing condition. In the case of semicolons, participants spoke fewer in the missing condition (60% of the time) compared to the highlighted condition (69% of the time). This suggests two potential explanations for the differences in punctuation use. First, it is possible that participants struggled to balance out the parentheses or quotes when they could not see the line of code, leading to a decrease in their use of punctuation. Second, it is possible, participants anticipated that an intelligent voice programming system would auto-complete the missing punctuation, leading them to rely less on their own use of these punctuation marks.

4.3 Correctness and Semantic Ambiguity

We suspected participants might sometimes speak incorrect code (i.e. code that does not achieve the program's stated objective). This could occur especially often when participants could not see the line. The two authors independently categorized each spoken single line of code as either correct or incorrect. Utterances were considered incorrect when the spoken code was incomplete, incorrect, or ambiguous. Inter-rater reliability was high (Cohen's kappa = 0.88), indicating close agreement between the raters. To ensure consistency, we reviewed our ratings and resolved any disagreements.

We calculated the proportion of participants' incorrect spoken code. Overall, novices spoke incorrect lines 16% of the time while experts spoke incorrect lines 7% of the time. As might be expected, participants spoke more incorrect code in the missing condition than in the highlighted condition. Novices spoke incorrect lines 28% of the time in the missing condition but only 4% in the highlighted condition. Similarly, experts spoke incorrect lines 12% of the time in the missing condition but only 2% in the highlighted condition.

We observed that the incorrect spoken programs in the highlighted condition were a result of unclear or ambiguous speaking patterns. We felt a voice programming system might have difficulty accurately transcribing such speech. For example, a few participants spoke the line of code "`result=Math.sqrt(x+y)/z`" as "result equals math dot square root x plus y divided by z". Without mentioning the order of arithmetic operations, the system might interpret this as "`x+y/z`". While some participants mentioned the order explicitly, for example saying "result equals math dot square root open paren x plus y close paren divided by z".

Some participants failed to specify whether a line of spoken code included a digit or a character. This particularly occurred in the conditional statement "`c <= '9'`" in which participants simply spoke it as "if c less than or equal to nine". We also observed a few participants mistakenly spoke "backslash" while dictating a comment instead of "forward slash". Additionally, we observed that some participants did not specify whether a line of spoken code was a comment, but instead spoke just the comment's text. Such ambiguity or lack of context may lead to an inaccurate machine translation of the spoken code to its target code.

4.4 Speaking Rate

We trimmed silence from the start and end of the recordings and calculated the speaking rate of an utterance in words per minute (wpm). As we did not have enough multi-line code from novices,

Target code	Programmer	Human Transcript
`items[i] = scan.nextInt();`	novice	`items square bracket i end square bracket equals scan dot next int semicolon`
	expert	`items at location i is equal to scan dot next int`
`largeNumCounter−;`	novice	`large num counter minus minus semicolon`
	expert	`decrement large num counter`
`while (num >= 1)`	novice	`while parenthesis num is greater than or equal to one end parenthesis quotation`
	expert	`start while loop start condition num is greater than or equal to one end condition end while loop`
`total = calculate_sum(age,5);`	novice	`total equals calculate underscore sum open parenthesis age comma five close parenthesis semicolon`
	expert	`variable total is equal to method calculate sum where the first argument is variable age and the second argument is the number five`

`/**`
`*Calculate the area of a square`
`*@param s side of the square`
`*@return area of the square`
`*/`
	novice	`java doc calculates the area of a square at param s side of the square at return area of the square`

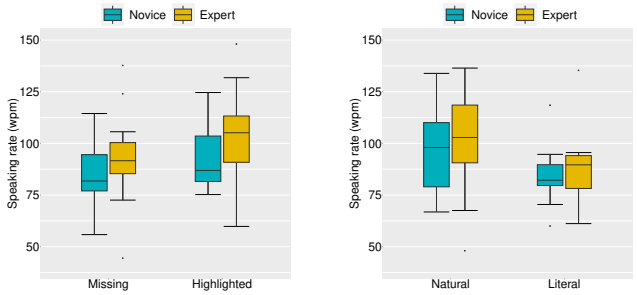
	expert	`start multiline comment new line calculates the area of a square new line at param s side of a square new line at return area of a square new line end multiline comment`

Table 3: Some examples of the variations in the speech of novice and expert programmers.

we analyzed only the single lines. In the transcripts, there were about 18 words per missing line and 19 words per highlighted line on average.

A mixed ANOVA analysis revealed no significant interaction between experience level and condition ($F(1, 22) = 0.18$, $p = 0.67$) on speaking rate. There was no significant main effect of experience level on the speaking rate ($F(1, 22) = 1.20$, $p = 0.29$). However, there was a significant main effect of condition on the speaking rate ($F(1, 22) = 18.87$, $p = 0.0003$). Posthoc pairwise comparisons with Bonferroni corrections revealed that experts spoke significantly faster in the highlighted condition than in the missing condition ($p = 0.004$). Similarly, novices spoke significantly faster in highlighted versus missing ($p = 0.002$) (Table 2). It might be the case that the increased cognitive demands of mentally visualizing the target line based on the surrounding code may have required additional time, resulting in a slower speaking rate when participants could not see the line.

We also calculated the speaking rate of novices and experts when speaking naturally versus when they spoke in a literal manner. We excluded participants who always spoke naturally (one novice and two experts) or always spoke in a literal manner (one novice). This resulted in a sample of ten novices and ten experts. We found no significant interaction between experience levels and speaking styles on the speaking rate ($F(1, 18) = 0.181$, $p = 0.68$). There was no significant main effect of experience level ($F(1, 18) = 0.071$, $p = 0.79$), but there was a significant main effect of speaking style on the speaking rate ($F(1, 18) = 9.231$, $p = 0.007$). Posthoc pairwise comparisons with Bonferroni corrections indicated that novices had a significantly faster speaking rate when speaking naturally compared to speaking in a literal manner ($p = 0.02$), while there was no significant difference in speaking rate between the two styles for experts ($p = 0.14$). This suggests that the effect of speaking style on speaking rate did not differ significantly between novice and expert speakers but the way in which novice participants spoke affected their speaking rate. It might be the case that following strict rules of grammar and syntax imposed additional cognitive demands on novices which slowed down their speaking rate compared to a more natural speaking style.



(a) Missing versus highlighted **(b) Natural versus Literal**

Figure 2: Comparison of speaking rate.

4.5 Subjective Feedback

Participants rated five statements about their overall experience with the study on a 7-point Likert scale (1=strongly disagree, 7= strongly agree). Experts' ratings showed a significant difference ($\chi^2(3) = 111.3$, $p = 0.00001$). Posthoc pairwise comparisons with Bonferroni adjustment revealed that experts found it significantly easier to speak a single highlighted line (mean = 6.2, SD = 1.4) compared to a single missing line (mean = 4.8, SD = 1.6) ($p = 0.0002$). There were no significant differences in their ratings for the ease of speaking a single missing line (mean = 4.8, SD = 1.8) versus multiple missing lines (mean = 3.5, SD = 1.7), between single highlighted line versus multiple highlighted lines ($p = 0.13$), or between single missing lines versus multiple missing lines ($p = 0.4$).

A significant difference was also found in novices' ratings ($\chi^2(3) = 121.7$, $p < 0.000002$). Posthoc pairwise comparisons revealed significant differences in their ratings for the ease of speaking a single highlighted line (mean = 6.2, SD = 1.4) versus a single missing line (mean = 4.8, SD = 1.6), as well as for the ease of speaking multiple highlighted lines (mean = 5.2, SD =1.5) versus multiple missing lines (3.4, SD = 1.4). However, there were no significant differences in their ratings for the ease of speaking a single missing line versus multiple missing lines ($p = 0.3$), or between a single highlighted line versus multiple highlighted lines ($p = 0.3$).

In general, both novices and experts found speaking a missing line challenging, but novices faced more difficulty in speaking multiple missing lines. One expert said "It is more difficult to make code that is nontrivial from scratch as opposed to reading a line that already exists". Speaking missing lines of code seems practical and relevant as in the context of an actual voice programming system, people may need to describe code without being able to see it or may only have partial context. However, verbalizing missing code does require more cognitive effort. Although we provided participants with some context in the form of comments (e.g. "This program reads ten integers from standard input into an array named items") in both highlighted and missing conditions, it is worth noting that such context may introduce biases as participants might rely heavily on the provided information. One expert participant acknowledged that he was biased, "When reading comments I had a temptation to want to follow the comment that was provided". This suggests that an effective data collection methodology for spoken code needs to balance the benefits and drawbacks of speaking missing lines to ensure reliable data while also taking steps to avoid biasing the programmer to one particular solution.

We asked participants about the ease of the programs. All experts and all but one novice found the programs easy to understand. When asked about the parts of programs they were most uncertain about how to dictate, novices and experts had differing opinions. All experts and three novices indicated that dictating function declarations was the hardest. One expert said "I was most uncertain about method keywords, for example how to differentiate different parts of the method declaration". None of the novices but five expert programmers thought dictating variables was really challenging. One expert said "It's complicated to figure out how to speak variable names when considering issues like capitalization and whether to spell out an identifier".

Some participants were uncertain about whether to dictate punctuation, and they believed that an intelligent voice programming IDE should auto-complete punctuation, especially when it comes to balancing braces and parentheses. According to a novice programmer, "I was uncertain mostly what punctuation I needed to state directly and what could be auto-completed". This suggests further investigation is required to overcome the challenges in dictating difficult parts of code such as method declarations, variable names, and punctuation.

A few expert programmers shared additional comments on how an intelligent voice programming tool should work in general. One expert said "I started off very literal but pretty soon realized that would be an unmanageable way to code and started assuming a smarter model. For instance, typically Java style is to camel case variable names so I assumed that should be the default interpretation". Another expert programmer noted "There is a lot of nuance to simple code such as Math.sqrt(x+y). Although it's very short and simple to spell out, I found it really challenging to try to express it in a command-type way". These insights support our approach of collecting data by asking participants to speak code without imposing any rules, as it enables the system to account for the diverse ways in which people might speak code. For instance, an intelligent system should be able to recognize the function `sqrt` regardless of whether it is spelled out letter-by-letter or spoken naturally as "square root".

5 RECOGNIZING SPOKEN PROGRAMS

We conducted three offline recognition experiments on our collected data. The goal of the first experiment was to measure recognition accuracy on spoken code using a state-of-the-art commercial speech recognizer. The second experiment was conducted to compare data collected in the missing condition versus highlighted condition. The goal of the third experiment was to see whether accuracy could be improved by adapting the language model by gradually augmenting the number of transcripts of spoken code.

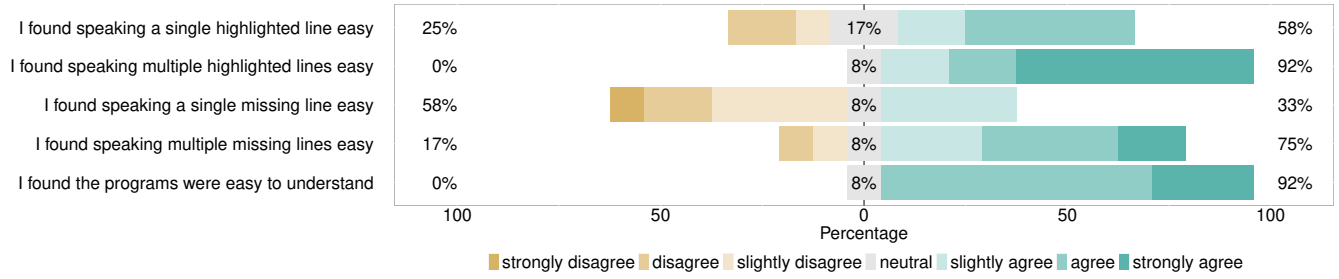
5.1 Recognition with Base Model

In the first experiment, we recognized our audio recordings with Google Cloud Speech-to-Text[5] service. We used a web API to recognize each audio file. As test data, we used 224 recordings from the novice study (after discarding incomplete recordings) and all 240 recordings from the expert study. We submitted audio using 16-bit linear encoding at a sampling rate of 16 kHz.

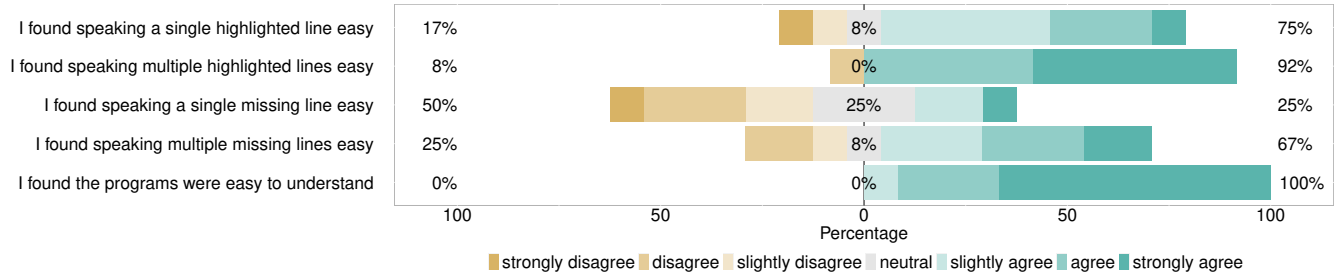
5.2 Language Model Adaptation

In the second experiment, we used Google's language model adaptation feature to test the recognition accuracy. We adapted the language model by incorporating transcripts from unseen programs spoken by unseen users, which were not part of the test data. In order to investigate recognition accuracy on unseen users' speech, we conducted a leave-one-participant-out cross-validation which is a modified form of k-fold cross-validation where each fold excluded one participant's data. We adapted the model using the unseen adaptation data from all participants except for one and tested it on that selected participant. The aim was to evaluate the adapted

[5]https://cloud.google.com/speech-to-text



(a) Feedback from novices



(b) Feedback from experts

Figure 3: Novice participant feedback (top) and expert participant feedback (bottom). The percentages on the left are the portion of participants who strongly disagreed, disagreed, or slightly disagreed with the statements. The percentages in the middle correspond to the portion who were neutral. The percentages on the right correspond to those who strongly agreed, agreed, or slightly agreed.

language model's ability to learn new programs spoken by new users. We repeated this process for each of the 24 participants.

5.2.1 Creating the adaptation dataset. To ensure that there is no overlap between adaptation and test data, we split the transcripts into two sets– the first half contains transcripts associated with the first ten programs and the second half from the last ten programs. Each set provided a sufficient number of each of our distinct programming constructs such as loops, if-else statements, expression statements, arrays, comments, and functions.

Google's model adaptation requires a set of unique phrases in order to improve the model's recognition of those particular words or phrases. To create this set, we generated different n-grams from the adaptation transcripts ranging from bigrams (e.g. "left paren") to 12-grams (e.g. for i equal to zero i less than five i plus plus). Bigrams to 12-grams were chosen for adaptation dataset creation as they met the limits of Google's speech-to-text adaptation API (5000 maximum phrases per request, 100,000 characters per request, and 100 characters per phrase). We found that the bigrams yielded the best recognition accuracy and selected it for further experimentation. We believe thew reason for this is that frequently repeated phrases or keywords like "system dot", "int i", "plus plus", and "open paren" are more common in spoken code transcripts than complete sentences. As a result, bigrams are more effective in capturing these repeated phrases and keywords compared to higher order n-grams which may detect more complex and infrequent patterns. Furthermore, the use of higher n-grams leads to an increase in the complexity and computational cost of the model, which can be impractical for larger datasets.

5.2.2 Comparing missing versus highlighted adaptation data. We wanted to investigate the impact on adaptation of collecting data in the missing and highlighted conditions. To do this, we adapted the model by taking transcripts from either the missing lines or the highlighted lines. Afterward, we conducted the leave-one-out experiments on each of these sets. For example, to recognize P1's first ten spoken programs, we adapted the model with missing line transcripts from the last ten programs of all participants except for P1.

5.2.3 Amount of adaptation data. We explored the effect of gradually increasing the amount of adaptation data from 25%, 50%, 75%, and finally, 100% on the improvement in recognition accuracy. For this, we created adaptation sets by randomly sampling 25%, 50%, and 75% missing transcripts combined with 25%, 50%, and 75% highlighted transcripts respectively, and finally 100% of the transcripts. We repeated the random sampling ten times and took the average.

5.3 Evaluation

We used simple heuristic rules to post-process the recognition results from Google's recognizer by converting all numbers and symbols into corresponding words. This was to ensure the recognition error rate could be fairly computed against our human transcriptions (in which our protocol was to spell out all numbers and symbols). We then calculated the Word Error Rate (WER) for the

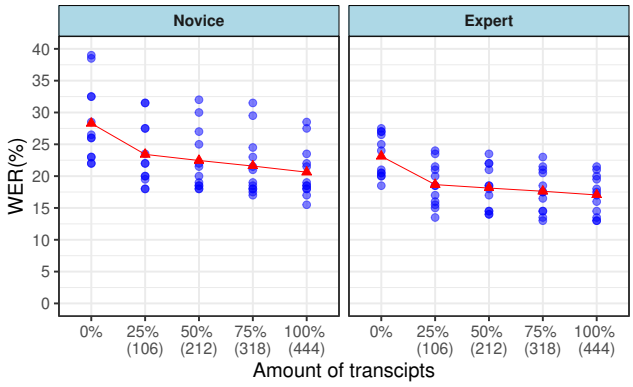


Figure 4: Comparison of Word Error Rate (WER) using increasing amounts of adaptation transcripts for the novices (left) and the experts (right). The mean value is marked as a red triangle. The x-axis shows the percentage of transcripts used followed by the exact number of lines in parentheses.

post-processed recognition results using our human transcriptions as the reference. WER was calculated by summing the number of insertions, deletions, and substitutions that occurred in the recognition result compared to the reference transcript, dividing by the number of words in the reference, and multiplying by 100.

5.4 Results

The baseline model had a high WER of 28.29%, for novices and 23.13%, for experts (Table 4). Adapting the language model on transcripts from just the highlighted condition provided similar improvements to using just transcripts from the missing condition. This suggests that both conditions provide enough variations in spoken code to facilitate the language model's learning.

Adding a progressive amount of transcripts yields a substantial reduction in error rate. As shown in Figure 4, the WER significantly drops with only 25% of the adaptation transcripts, 19% relative to the baseline. Adding in more transcripts provided a slight reduction in error rate, though gains began to diminish. It is possible that the model is already able to capture most of the relevant patterns and information from the data, so adding more data did not provide as much benefit. On average, across all participants, adapting with 100% transcripts reduced WER by 27% relative to the baseline.

Table 5 shows some examples of target lines of code, our human reference transcripts, recognition results using the base model from Google, and recognition results using language model adaptation on 100% of the transcripts. Model adaptation improved the recognition of homophones such as "for" versus "four", "i" versus "eye", "u" versus "you" and "two" versus "to". Recognition of "for", "i", "u" and "two" were improved by 70%, 74%, 8%, and 9% relative respectively, indicating that the model learned the context in which these words were used.

Language model adaptation with all transcripts (444 lines) yielded a WER of 19% (averaged across participants). This means that nearly one in every five words spoken was not accurately recognized by the system. This level of performance is not ideal for a speech recognizer, especially when compared to the lower WERs achieved in recent years for natural language [2]. Additionally, the words that do not occur in natural language such as "int" and "num" were frequently misrecognized (99% and 74% of the time, respectively) even after adapting the model. This limitation could be addressed by incorporating more domain-specific language into the training data or by using a different language model that is specifically tailored to technical language. Furthermore, the fact that the gains in accuracy began to diminish as more transcripts were added indicates that there are other factors that are limiting the system's accuracy, such as the sophistication of Google's language model adaptation algorithm, the quality of the acoustic models, or the robustness of the system to different accents and speaking styles. These limitations highlight the need for future research to explore alternative approaches that can improve the accuracy and robustness of speech recognition systems in the domain of spoken code.

6 DISCUSSION

Today's educational and work landscape requires people to be well-versed in computational thinking, ideally with at least some exposure to programming. Programming is already a cognitively demanding task, adding additional demands only creates additional barriers to entry. Further, novice programmers, or even experts moving between languages, can struggle with the syntactic minutia of a particular language. Our user studies focused on a simple and, we argue, core part of programming by voice, namely the input of common statement types. Rather than imposing a prescriptive grammar for writing such statements, we instead observed how both novice and expert programmers would like to be able to speak such statements. The advantage of our approach is that if we can support flexible and more natural speaking of code, we may ease or eliminate the need for programmers to memorize a complicated set of commands.

Our current approach was to have people record themselves speaking code to a hypothetical system. It could be that a person's speaking style changes when interacting with a real system. Lacking a real system, one could instead collect audio via a Wizard of Oz approach. However, we think a more significant problem is collecting a larger and more diverse set of data. This data should include programmers with motor impairments, a wider range of programming constructs, and languages beyond Java. While many languages share similar programming constructs, the details of a specific language may require additional support for certain purposes (e.g. how to specify indentation in Python).

In our user studies, we asked programmers to speak a single line or a small section of code. People may speak differently when creating an entire program from scratch, when creating more difficult programs, or when using more advanced language constructs. While it might be desirable to support generating blocks of code from a single utterance (e.g. "create a for-loop that prints all the

Adaptation Data	Programming experience	
	Novice	Expert
None (baseline)	28.29±3.0	23.13±2.0
All missing transcripts	22.75±2.0	18.92±2.0
All highlighted transcripts	23.00±2.0	17.96±2.0
25% missing + 25% highlighted transcripts	23.33±2.0	18.67±2.0
50% missing + 50% highlighted transcripts	22.46±2.0	18.12±2.0
75% missing + 75% highlighted transcripts	21.75±2.0	17.63±2.0
All missing and highlighted transcripts	20.79±2.0	17.04±2.0

Table 4: Word error rate (WER) using Google speech recognizer. ± values represent sentence-wise bootstrap 95% confidence intervals.

Target code	Transcript	Text
`String veryLargeString2="world";`	human	string very large string two equals world
	base model	string very large string to equals world
	adapted model	string very large string two equals world
	human	string space very large string two equals quote world quote
	base model	bring space very large string two equals quote world quote
	adapted model	string space very large string two equals quote world quote
`while(num>=1)`	human	while num greater than equal to one
	base model	well none greater than equal to one
	adapted model	while none greater than equal to one
	human	while loop condition num greater than zero end condition left bracket
	base model	while loop condition none greater than zero and condition lock bracket
	adapted model	while loop condition num greater than zero and condition left bracket
`for(int i=1; i<=n; i++)`	human	for int i equals one i is less than or equals to n i plus plus
	base model	four plus two is equals one i is less than or equal to n high bus bus
	adapted model	for and i equals one i is less than or equal to n i plus plus
	human	for i equals one i less than or equal to n i plus plus
	base model	four equals one i less than or equal to an eye plus twelve
	adapted model	four equals one i less than or equal to n i plus plus
`largeNumCounter--;`	human	large num counter minus minus
	base model	large and dumb counter minus minus
	adapted model	large num counter minus minus
	human	decrement large num counter
	base model	decrement large and i am connor
	adapted model	decrement large num counter

Table 5: Recognition results used the base model and the adapted model on 100% of the transcripts. Recognition errors are highlighted in red and underlined.

prime numbers in array nums"), it may not always align with programmers' needs and preferences. Novice programmers or even experienced programmers may prefer to speak a single line of code when they want to make an edit or modification to an existing codebase, rather than generating a large block of code from a single utterance. We think supporting robust entry of individual lines is a necessary first step prior to considering the input of larger blocks. Related, even single lines of code in some cases can be quite long. This may require an interface that incrementally displays a partially completed line code as the user speaks. This introduces the additional challenge of recognizing an utterance specifying only part of a statement and also converting this utterance to an incomplete line of code.

Existing code corpora [7, 10] are sourced from written code. While these corpora can be useful for tasks like code summarization, code suggestion, and code generation, they are not well-suited for training a system to transcribe spoken code line-by-line. Our collected data maps each line of a human transcript with the corresponding target code. To our knowledge, this is the first work to build a spoken program corpus of this nature.

While we were able to reduce WER by 27% relative to the baseline (no adaptation) with our modest amount of transcribed spoken code, we reached a plateau. One potential solution is to use transfer learning and fine-tune a large pre-trained model on a small amount of data. This approach would allow the model to better understand the nuances and variations in spoken code where we have full control of the acoustic and language model rather than using a commercial service as we did here. In addition, individuals might have specific preferences for programming such as formatting and naming conventions. The language model may perform better if conditioned on surrounding code or on a programmer's preferred coding style. Another future research direction would be to adapt to individual programming styles by incorporating user feedback into the system. The system can track and learn from the programmer's feedback over time, allowing it to continually improve its ability to recognize an individual's coding style.

We found both novices and experts spoke in a natural way as well as in a literal way. Using natural language prompts for code generation is becoming increasingly popular and can help make programming more accessible and intuitive for a wider range of people. However, there can be issues with ambiguity and confusion in natural language prompts, which can lead to incorrect code being generated. For example, a prompt like "find the most similar items in a collection" could be ambiguous without specifying how to define the most similar. By analyzing our collected data of both natural and literal spoken code, we can investigate which style enables the model to better understand and interpret programmer prompts. Further research is required to investigate ways to fine-tune LLMs to better adapt to the individual coding styles of different programmers. By doing so, the generated code could better match the programmer's preferred formatting and syntax choices, leading to more seamless integration with their existing codebase.

Our study revealed that both novice and expert programmers did not exhibit a different speaking style in the missing versus highlighted condition, but they did speak faster in the highlighted condition. This could be attributed to the reduced cognitive demand in speaking the presented code, as opposed to generating code from scratch in the missing line condition. Moreover, novices skipped speaking the missing lines of code may be because they lacked knowledge. However, when designing a data collection methodology, it is important to carefully consider the task's goals and objectives and the trade-offs between different approaches. A hybrid approach that combines both missing and highlighted conditions may help balance the trade-off between realism and cognitive load, resulting in a more comprehensive dataset that accurately reflects the demands of coding tasks while enabling faster data collection. Moreover, providing additional context and guidance for novices, such as prompts or hints, may improve data validity by helping them generate code more effectively.

Scaling up from our relatively small corpus poses a number of challenges. First, data needs to be collected from people with at least some programming experience, a relatively small portion of the population that is expensive to recruit. Second, it is time-consuming to transcribe the utterances. This could perhaps be made more efficient by having people (e.g. crowdsourced workers) correct automatically generated transcripts of the utterances. However, this would require reasonably accurate recognition in the first place to be faster than typing the transcript from scratch. Third, even with large amounts of data, it may be challenging to model the commonalities in spoken code when they involve unconstrained tokens (e.g. variable names in a for-loop statement).

7 CONCLUSION

As speech recognition technology continues to improve, it is likely that voice programming systems will become increasingly prevalent and accessible. In this study, we provided valuable insights into the challenges of developing such a system. Regardless of the experience, programmers tended to speak faster using natural language, thus motivating the need for a naturally spoken programming system. We found standard speech recognizers had a high error rate due to the mismatch between the written English they were trained on and how people speak code. However, even using the modest number of examples of spoken code we collected, we showed recognition errors could be reduced. Our study demonstrated the trade-off between the realism of the task and cognitive demand, which is important for researchers to consider when designing experiments and selecting data collection methodologies. By understanding this trade-off, researchers can make informed decisions about the most appropriate method for collecting data that aligns with their research goals. Overall, our findings can inform the design of future voice programming systems that takes into account diverse ways in which programmers might speak code, making the system more accessible to a wider range of users.

ACKNOWLEDGMENTS

This work was supported in part by a Google Faculty award.

REFERENCES

[1] Stephen C Arnold, Leo Mark, and John Goldthwaite. 2000. Programming by voice, VocalProgramming. In *Proceedings of the fourth international ACM conference on Assistive technologies*. ACM, 149–155.

[2] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems* 33 (2020), 12449–12460.

[3] Andrew Begel and Susan L Graham. 2005. Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 99–106.

[4] Andrew Begel and Susan L Graham. 2006. An Assessment of a Speech-Based Programming Environment. In *Visual LAnguages and Human-Centric Computing (VL/HCC'06)*. IEEE, 116–120.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Alain Désilets. 2001. VoiceGrip: A Tool for Programming-by-Voice. *International Journal of Speech Technology* 4, 2 (2001), 103–116.

[9] Cassandra Guy, Michael Jurka, Steven Stanek, and Richard Fateman. 2004. Math speak & write, a computer program to read and hear mathematical input. *University of California Berkeley* (2004).

[10] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv e-prints* (2019), arXiv–1909.

[11] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, Vol. 1. 2.

[12] Rinor S Maloku and Besart Xh Pllana. 2016. HyperCode: Voice aided programming. *IFAC-PapersOnLine* 49, 29 (2016), 263–268.

[13] Jonathan Giovanni Soto Muñoz, Arturo Iván de Casso Verdugo, Eliseo Geraldo González, Jesús Andrés Sandoval Bringas, and Miguel Parra Garcia. 2019. Programming by Voice Assistance Tool for Physical Impairment Patients Classified in to Peripheral Neuropathy Centered on Arms or Hands Movement Difficulty. In *2019 International Conference on Inclusive Technologies and Education (CONTIE)*. IEEE, 210–2107.

[14] Sadia Nowrin, Patricia Ordóñez, and Keith Vertanen. 2022. Exploring Motor-Impaired Programmers' Use of Speech Recognition. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility* (Athens, Greece) *(ASSETS '22)*. Association for Computing Machinery, New York, NY, USA, Article 78, 4 pages. https://doi.org/10.1145/3517428.3550392

[15] Obianuju Okafor and Stephanie Ludi. 2022. Voice-Enabled Blockly: Usability Impressions of a Speech-driven Block-based Programming System. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '22)*. 1–5.

[16] David E Price, DA Dahlstrom, Ben Newton, and Joseph L Zachary. 2002. Off to See the Wizard: using a" Wizard of Oz" study to learn how to design a spoken language interface for programming. In *32nd Annual Frontiers in Education*, Vol. 1. IEEE, T2G–T2G.

[17] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[18] Lucas Rosenblatt, Patrick Carrington, Kotaro Hara, and Jeffrey P Bigham. 2018. Vocal Programming for People with Upper-Body Motor Impairments. In *Proceedings of the Internet of Accessible Things*. ACM, 30.

[19] Hadeel Saadany, Constantin Orăsan, and Catherine Breslin. 2022. Better Transcription of UK Supreme Court Hearings. *arXiv preprint arXiv:2211.17094* (2022).

[20] Waseem Sheikh, Dave Schleppenbach, and Dennis Leas. 2018. MathSpeak: a non-ambiguous language for audio rendering of MathML. *International Journal of Learning Technology* 13, 1 (2018), 3–25.

[21] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[22] Jessica Van Brummelen, Kevin Weng, Phoebe Lin, and Catherine Yeo. 2020. CONVO: What does conversational programming need?. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[23] Amber Wagner and Jeff Gray. 2015. An empirical evaluation of a vocal user interface for programming by voice. *International Journal of Information Technologies and Systems Approach (IJITSA)* 8, 2 (2015), 47–63.

[24] Amber Wagner, Ramaraju Rudraraju, Srinivasa Datla, Avishek Banerjee, Mandar Sudame, and Jeff Gray. 2012. Programming by Voice: A Hands-Free Approach for Motorically Challenged Children. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI EA '12)*. Association for Computing Machinery, New York, NY, USA, 2087–2092. https://doi.org/10.1145/2212776.2223757

[25] Angela M Wigmore, Gordon JA Hunter, Eckhard Pflügel, and James Denholm-Price. 2009. TalkMaths: A speech user interface for dictating mathematical expressions into electronic documents. In *International workshop on speech and language technology in education*.