# Code-to-Speech: An Exploratory Study of AI Text-to-Speech for Source Code Readability

Marco Cardia
Department of Computer Science
University of Pisa
Pisa, Italy
marco.cardia@di.unipi.it

Letizia Angileri
Department of Computer Science
Univeirsity of Pisa
Pisa, Italy
letizia.angileri@di.unipi.it

Marina Buzzi
IIT
CNR
Pisa, Italy
marina.buzzi@iit.cnr.it

Barbara Leporini
Computer Science
University of Pisa
Pisa, Italy
ISTI
CNR
Pisa, Italy
barbara.leporini@unipi.it

## Abstract

As multimodal interaction becomes central to digital learning, technical interfaces, such as eBooks and conversational agents, present unique accessibility challenges for users relying on the auditory modality. While modern Text-to-Speech (TTS) systems excel at rendering natural language, they often fail to convey the complex, structural syntax of source code, creating a significant barrier for blind and low-vision programmers. We frame the vocal rendering of source code as a cross-modal translation task, in which spatial and symbolic information visually encoded in code must be mapped into a linear auditory signal. This study explores "Code-to-Speech" (CTS) as a critical component of multimodal technical communication. We present a comparative evaluation of five leading AI-driven TTS engines (including ChatGPT, Gemini, and ElevenLabs) to determine their efficacy in rendering mixed-content (text and code) materials. Adopting a mixed-methods approach, we combine automated linguistic metrics (WER, BLEU, METEOR) with a human-centered qualitative assessment. Our findings reveal significant differences: ChatGPT and Gemini consistently achieved lower error rates, while Copilot omitted critical tokens. However, all systems struggle with essential structural cues such as indentation, pauses, and code comments. We also show that designed prompts improve intelligibility, particularly for novices. Our work establishes a benchmark for CTS and underscores the need for multimodal AI architectures that support inclusive technical interaction.

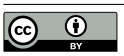
## CCS Concepts

• **Human-centered computing** → **Accessibility design and evaluation methods**; *Empirical studies in accessibility*; *Auditory feedback*; • **Computing methodologies** → *Natural language generation*; • **Social and professional topics** → *People with disabilities*; *Computer science education*; • **Hardware** → Sound-based input / output.

## Keywords

Text-to-Speech, Accessibility, Multimodal Interaction, Programming Education, Assistive Technology, Blind Users, Cross-modal Translation, Human-centred AI

**ACM Reference Format:**
Marco Cardia, Letizia Angileri, Marina Buzzi, and Barbara Leporini. 2026. Code-to-Speech: An Exploratory Study of AI Text-to-Speech for Source Code Readability. In *INTERNATIONAL CONFERENCE ON MULTIMODAL INTERACTION (ICMI '26), October 05–09, 2026, Napoli, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3776574.3831223



This work is licensed under a Creative Commons Attribution 4.0 International License.
*ICMI '26, Napoli, Italy*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2318-6/26/10
https://doi.org/10.1145/3776574.3831223

## 1 Introduction

The increasing use of digital resources for learning programming, such as technical manuals and eBooks, presents significant accessibility challenges for screen reader users who rely on Text-to-Speech (TTS) systems [9, 16]. TTS engines are steadily improving in quality, yet they are still primarily optimized for reading narrative text rather than non-textual content or source code. In recent years, moreover, TTS has become a key component of conversational agents, both in voice-based dialogues and through "read aloud" functions that vocalize system responses. As a result, TTS systems are increasingly essential tools for reading across different contexts. Consequently, they must be optimized to render all types of content with appropriate pauses, inflections, and prosody. Despite their growing effectiveness and quality in processing narrative text, TTS systems encounter significant challenges when reading programming code [4, 8, 13, 15, 18]. Code is often treated as ordinary text, or worse, as if it were narrative text or poetry. Yet source code has characteristics very different from natural language: it is dense with syntax, special characters, and strict structures that must be conveyed precisely in order to preserve meaning. Therefore,

code cannot be read in a fluid, continuous manner like narrative text. Furthermore, when code is embedded in eBooks or generated dynamically in conversational environments, missing pauses, mispronunciations, omissions, or unnatural prosody can make the reading tedious and hinder comprehension, ultimately complicating the learning process. For novice programmers, such difficulties represent an additional barrier in an already demanding context [4, 8]. Beyond these challenges, the problem of rendering source code through speech can be seen as a multimodal interaction task. In multimodal human–computer interaction, information is conveyed through multiple modalities, such as visual layout, textual structure, and auditory output, which must be coherently integrated to support user understanding [3]. Source code is inherently multimodal: its meaning depends not only on textual tokens, but also on spatial organization, indentation, and visual grouping. When code is transformed into speech, these visual and structural cues must be translated into the auditory modality. This requires more than accurate pronunciation; it involves mapping spatial and symbolic information into temporal and acoustic signals. Failures in this cross-modal translation, such as missing pauses, incorrect rendering of symbols, or the absence of structural cues, can significantly degrade comprehension, particularly for users who rely exclusively on auditory interaction. From the perspective of multimodal interaction research, this task can be understood as a modality translation problem, where structured visual information is converted into a linear auditory representation. Current TTS systems, however, largely treat code as flat text, without leveraging intermediate structural representations. As a result, they often fail to preserve key aspects of code semantics and organization. This perspective aligns with the broader goals of multimodal AI research, which aims to develop human-centered systems capable of integrating and transforming information across modalities [2].

This study addresses the issue by introducing the concept of Code-to-Speech (CTS), that is, the vocal rendering of programming code through TTS systems. Specifically, we analyze how some of the leading AI-driven TTS engines, including Google TTS (gTTS), ChatGPT, Gemini, ElevenLabs, and Copilot, read portions of source code. The goal is to evaluate the readability of mixed-content materials, consisting of descriptive text and code snippets, and to establish an initial benchmark for CTS. The contributions of this work are threefold. First, we present a comparative analysis of TTS performance in code reading, highlighting differences across engines. Second, we introduce and adapt readability metrics for mixed text-and-code scenarios, combining automatic measures such as Word Error Rate (WER), Bilingual Evaluation Understudy (BLEU), and Metric for Evaluation of Translation with Explicit ORdering (METEOR) with subjective assessments such as the Mean Opinion Score (MOS) [11, 14, 17]. Third, we report an experimental evaluation that integrates automatic analysis with human feedback, offering both quantitative and qualitative perspectives. By framing CTS as a new evaluation domain, our study aims to highlight the limitations of current TTS systems and to encourage the development of more inclusive solutions, relevant both for programming education and for the advancement of conversational AI systems.

The paper is organized as follows. Section 2 reports related work. Section 3 describes the methodology, including dataset, experimental design, and evaluation metrics. Section 4 presents the results of both qualitative and quantitative analyses. Section 5 discusses the findings and introduces a conceptual framework for code vocalization. Finally, Section 6 concludes the paper and outlines directions for future work.

## 2 Related Work

While modern TTS engines excel at reading natural language, their application to programming content presents unique accessibility challenges. Source code contains symbols, indentation, and syntax that are critical for comprehension but often lost in spoken form. As a result, conventional TTS produces monotonous streams of speech that obscure block hierarchy, nesting, and syntax, presenting even more challenges for novices [6, 18, 22]. Further challenges arise from the inconsistent vocalization of programming symbols and abbreviations, which can confuse novices and increase cognitive load, making programming less accessible to blind and visually impaired students [4, 8].

To address these limitations, researchers have proposed several approaches. The Audio Programming Language (APL) provided an audio-first paradigm, enabling blind novices to develop algorithmic thinking without dealing with complex syntax, instead relying on spoken feedback [1, 19, 20]. CodeTalk, a Visual Studio extension, improved accessibility by categorizing difficulties into discoverability, glanceability, navigability, and alertability, offering auditory tree views, error tones, and "TalkPoints" for debugging [15]. Similarly, the Grid-Coding paradigm structured source code in a 2D grid, mapping indentation levels into auditory cues such as earcons (short, abstract sounds that represent functions or structures, like auditory icons) and spearcons (compressed speech cues, created by speeding up spoken words until they sound like unique tones), to provide more reliable navigation and editing experiences [6, 24].

Other research emphasizes the need for standardized vocalization. Hermans et al. propose a code phonology to reduce ambiguity and ensure consistency in how symbols, operators, and identifiers are spoken, thereby lowering cognitive effort for novices [8]. Complementary work highlights the utility of non-speech cues: Roberts and Weaver demonstrated that spearcons and tones improve recognition of braces and nesting levels [18], while Finlayson and Mellish found that combining speech with auditory icons can improve learners' ability to "glance" at code structure, especially for abstract information such as nesting depth [7].

Recent years have also witnessed substantial progress in TTS systems powered by deep learning and large language models, which produce natural-sounding speech with improved prosody and reduced listening fatigue [5, 23]. Yet, a rigorous evaluation of these modern systems in the context of source code is missing. Most prior work has focused on specialized auditory displays or controlled environments, rather than the performance of widely available systems, which are the ones students are most likely to encounter [12]. This gap motivates our study, which systematically benchmarks contemporary AI-driven TTS systems on mixed narrative and code materials, combining objective metrics such as WER, BLEU, and METEOR with the human rating MOS to assess intelligibility and accessibility for both novice and expert learners.

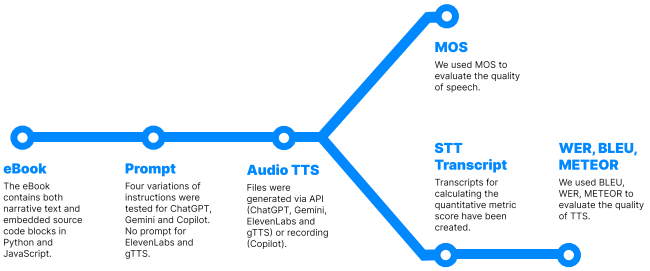



**Figure 1: Evaluation workflow: eBook content was converted to audio via TTS systems and assessed using MOS for subjective quality and BLEU, WER, and METEOR for quantitative evaluation.**

## 3 Method

Our objective was to examine how contemporary AI-based TTS systems handle programming code when that code is embedded in expository text. Rather than treating this as a pure speech-technology benchmark, we designed the study as a human computer interaction evaluation focused on intelligibility for two listener profiles: beginner and expert programmers, who have different needs and expectations when listening to code. Our experimental pipeline, illustrated in Figure 1, begins by synthesizing audio from an educational eBook using various TTS engines and instructional prompts. We then evaluate the resulting audio through two complementary method: a subjective listening test to gather perceptual judgments (MOS) and an objective analysis based on automated transcription metrics (WER, BLEU, METEOR).

### 3.1 Materials and TTS

To run the experiments, we defined an educational eBook titled *"JavaScript and Python: A Beginner's Guide to Loops and Functions"*, which contains both narrative text and embedded source code blocks in two programming languages: Python and JavaScript. We used this eBook as dataset for the evaluation of the TTS. It is accessible to beginners, as it introduces fundamental programming concepts such as naming conventions, functions, and control structures (e.g., `while` and `for` loops). At the same time, it covers the core syntax of these languages in a manner that can be extended to more complex functionality, thereby remaining relevant for more expert readers. To illustrate the challenges faced by learners, we included a representative sample from our dataset:

> ··· In this guide, you will learn about the for loop, the while loop, and functions.
> Chapter 1: The for Loop
> JavaScript Syntax

```
for (initialization; condition; incr) {
  // Code to execute
}
```

> Example:

```
for (let i = 0; i < 5; i++) {
  console.log("Iteration number: " + i);
}
```

> How it works:
> - let $i = 0$ initializes a counter.
> - i < 5 keeps the loop running while the condition is true.
>
> ...
> Python Syntax

```
for variable in range(start, end, incr):
  # Code to execute
```

> Example:

```
for i in range(0, 5, 1):
  print("Iteration number:", i)
```

The full content of the eBook can be found on the Appendix A.When read by screen readers, the transition between narrative text and code often lacks a clear pause or change in prosody, resulting in a monotonous flow. This makes it difficult for listeners, especially beginners, to perceive where the code begins, which symbols are present, and where the narrative resumes. To conduct our evaluation, we generated audio from five distinct TTS systems:

- OpenAI `gpt-4o-mini-tts`.
- Gemini `gemini-2.5-flash-preview-tts`.
- ElevenLabs `eleven_multilingual_v2`.
- Copilot `gpt-4o`.
- Google Text-to-Speech `gTTS v2.5.4`.

While Microsoft Copilot is also powered by a gpt-4o model, we evaluated it as a distinct system. This is because our study focuses on the real-world user experience, assessing the final product as a user would encounter it via the "Read Aloud" function on the Copilot webpage. The specific implementation, interface constraints, and preprocessing choices made by the application can significantly impact the final audio output. For each system, we synthesized the same eBook under each prompt. A challenge with modern AI-based synthesizers is that they can produce slightly different results each time they are run. To account for this inherent inconsistency and avoid judging a system on a single, potentially unrepresentative attempt, we generated five independent audio samples for every combination of an engine and a prompt. This allowed us to assess the typical or average performance of each system. This step was performed for all systems whose APIs permitted multiple generations (OpenAI, Gemini, ElevenLabs, Copilot); due to their deterministic output in our setup, gTTS provided a single synthesized file per condition.

### 3.2 Experimental Design

To evaluate how TTS systems adapt their vocalization for code, we designed an experiment that varied the instructional prompts given to each of the five TTS systems. Recognizing that novice programmers benefit from hearing explicit syntax (e.g., vocalizing punctuation and parentheses), while experts may prefer a more fluid reading, we created prompts to target these different needs. Following the principles of prompt engineering defined by Saravia [21], we wrote four distinct prompts:

P1 Generic `Read the contents of the following text aloud.`

P2 Profile `Read the following text aloud. I am a blind user and a programming {novice|expert}. The text includes source code.`
P3 Syntax `Read the following text aloud. I am a blind user and a programming {novice|expert}. The text includes Python and JavaScript source code. Please preserve syntax and structure when reading.`
P4 Literal `Read the following text aloud. I am a blind user and a programming {novice|expert}. The text includes source code; treat it as text while reading.`

These prompts were designed to systematically vary the amount of context provided to the TTS engine regarding the user's programming expertise, accessibility needs, and the nature of the content itself. Our design ranged from a generic, context-free command (P1) to prompts that specified a user profile and expertise level (P2), requested explicit syntax preservation (P3), and asked for a more literal, non-interpretive reading (P4). This approach allowed us to analyze how different instructions influenced the systems' prosody, articulation of symbols, and overall code intelligibility. A methodological note on prompt scope: the four prompts (P1–P4) were applied only to the three engines that accept user-controllable instructional input at synthesis time, ChatGPT ($gpt-4o-mini-tts$), Gemini ($gemini-2.5-flash-preview-tts$), and Copilot (via the "Read Aloud" function in its web interface). ElevenLabs ($eleven_multilingual_v2$) and gTTS (v2.5.4) were evaluated under a single prompt-free condition, since their APIs do not accept natural-language instructions that condition the rendering on user profile or content type. Reading-style control in these systems is exposed only through low-level parameters (e.g., voice selection, stability, similarity boost, speed), which do not provide an equivalent of the prompt-based contextualization tested for the three other engines. We therefore consider ElevenLabs and gTTS as non-prompt-conditioned baselines, and interpret their results accordingly: their performance reflects the default behavior of the engine on mixed text-and-code input, without the benefit of the user-context information that P2–P4 provide to the prompt-conditioned systems. This asymmetry is intrinsic to the current TTS landscape rather than a design choice of our evaluation.

## 3.3 Evaluation Metrics

We evaluated the clarity and accuracy of the generated speech from two different perspectives: a qualitative human evaluation and a quantitative automated evaluation. This dual methodology allowed us to capture both the perceived readability and the technical fidelity of each TTS system.

*3.3.1 Human Evaluation.* First, we conducted a preliminary listening test, based on the Mean Opinion Score (MOS). The evaluation was performed by three participants: two sighted researchers with expertise in programming and one blind user with expertise in programming and proficient in the use of screen readers. This diverse panel ensured that the evaluation captured both technical accuracy and the practical accessibility needs of the target population. The participants rated their quality based on five criteria designed for reading code, adapted from the guidelines proposed in the literature [4, 8, 13, 15, 18]:

(1) Use of pauses. Did the TTS pause correctly at the end of lines or between logical blocks of code?
(2) Recognition of code blocks. Did it acknowledge structural elements like curly braces {, } or the indentation used in Python?
(3) Pronunciation of symbols. Were symbols and punctuation marks like =, <, >, ; and , read out accurately?
(4) Pronunciation of brackets: Could the TTS distinguish between different types of brackets (round, curly, and square)?
(5) Reading of comments. Were comments within the code correctly identified and read aloud?

Each of these criteria was scored on a 1 (Unsatisfactory) to 5 (Excellent) scale, considering also the target audience, provided also in the input prompt. To assess the consistency of the qualitative ratings, we computed inter-rater agreement across the three raters using Krippendorff's $\alpha$ for ordinal data [10]. The agreement was 0.82 (95% CI: 0.75–0.95), indicating substantial consistency among raters. Disagreements were most frequent on the pronunciation of symbols criterion, where individual prosodic preferences (especially between the novice and expert profiles) introduced expected variability; on structural criteria such as use of pauses and recognition of code blocks, raters were almost always in agreement, since these features were typically either entirely absent or unambiguously present in the audio. A total of 124 audio files were evaluated across all conditions. For systems that produced multiple outputs (ChatGPT, Copilot, Gemini, ElevenLabs), the rater assigned an average score across the five generations for each criterion. For deterministic systems (gTTS), a single score was assigned. The qualitative MOS served as a multi-rater expert audit rather than a generalizable user study. Its purpose was to identify systematic breakdowns in code vocalization that inform future user-centered evaluations.

*3.3.2 Automated Evaluation.* For a quantitative assessment, we employed standard speech and translation metrics [11, 14, 17]. This involved a multi-step workflow. First, each audio file generated by the TTS systems was transcribed using a Speech-to-Text (STT) engine (in particular, ElevenLabs STT). This converted the synthesized speech back into a text format, which could then be compared against the original reference text. To validate the reliability of the transcription pipeline, we performed a manual audit of a randomized 20% sample of the audio files. A comparison between the ElevenLabs STT transcripts and manual human transcriptions yielded an average accuracy of 98.9% (WER $\approx$ 1.1%), indicating high fidelity. Furthermore, we observed that the STT engine correctly identified technical symbols (e.g., brackets, operators) consistently across all TTS voices.

To ensure a fair comparison, we applied a normalization procedure to both the reference and the transcribed text. This involved four key actions:

- Converting all narrative text to lowercase.
- Removing narrative related punctuation to focus the evaluation on code fidelity,
- Normalizing all whitespace to a single space.
- Mapping spoken phrases for symbols to their equivalent character. We built a custom dictionary mapping common spoken syntax phrases to their symbolic equivalents (e.g., ensuring "open parenthesis" or "left parenthesis" were both

aligned with the ( symbol). This prevents the system from being unfairly penalized for correctly verbalizing a symbol.

After the normalization step, we used three standard metrics to score the accuracy of the transcriptions. Each metric measures quality in a slightly different way:

- WER. This is the strictest metric. It functions like a proofreader, counting every single error—insertions (extra words), deletions (missing words), and substitutions (wrong words). A lower WER score is better, indicating fewer mistakes [14].
- BLEU score. This metric was originally designed for evaluating machine translation. It checks for overlap in phrases between the original text and the transcript, so it's more tolerant of minor word reordering with respect to WER. For BLEU, a higher score is better [17].
- METEOR score. This is a more advanced metric that also originated from translation. With respect to BLEU score, it accounts for stemming and punishes chunk reordering, providing a more structural assessment of the vocalized code blocks. A higher METEOR score is better [11].

A known limitation of using standard NLP translation metrics (BLEU, METEOR) for code is their reliance on n-gram overlap. They may score an output highly for reading most words correctly, even if a logically critical symbol (e.g., reading += as +) is altered. To facilitate open science and reproducibility, our evaluation scripts, the STT transcripts, the expected ground truth for both experts and beginners, and the normalization mapping dictionary are available in a GitHub repository[1]. By combining the qualitative insights from the human evaluation with the precise, data-driven results from these automated metrics, we built a comprehensive understanding of each TTS system's strengths and weaknesses.

*3.3.3 Ground-Truth Reference Generation.* To compute the automated metrics, we generated two distinct ground-truth text files corresponding to the "beginner" and "expert" profiles. The "beginner reference" verbalizes all syntax, while the "expert reference" follows a more fluid, conversational code-reading style. For example, for the code snippet `for (let i = 0; i < 5; i++) {`:

- **Beginner Reference:** "for open parenthesis let i equals zero semicolon i less than five semicolon i plus plus close parenthesis open curly bracket"
- **Expert Reference:** "for let i equals zero, i less than five, i plus plus, open block"

## 4 Results

This section presents the quantitative and qualitative findings from our evaluation. The results from the qualitative pilot study, based on the MOS, are presented in Table 1. The automated metrics, including WER, BLEU-4, and METEOR, are summarized in Table 2.

### 4.1 Qualitative Results

The qualitative results from our pilot MOS study, detailed in Table 1, reveal specific and recurring patterns in how the TTS systems vocalize code syntax. Our analysis identified several critical areas where all evaluated models performed poorly. Pause management was universally absent, with every model scoring the lowest possible rating (1 out of 5) on this criterion. Similarly, none of the models could correctly announce Python's indentation, a crucial feature for conveying code structure, which resulted in low scores for recognizing block delimiters. The reading of code comments also proved challenging for all systems, as the comment symbols (# or //) were frequently omitted, leading to scores of 1 or 2 across most conditions. However, systems like Gemini and ChatGPT successfully adapted to read comments in the expert context, scoring between 2.1 and 2.7.

In contrast, performance varied significantly in the pronunciation of symbols and punctuation. Gemini, ChatGPT, and gTTS demonstrated a notable ability to read symbols, achieving scores between 3 and 5 in several configurations. Gemini, in particular, correctly read symbols such as commas and equals signs for novice users. On the other hand, Copilot and ElevenLabs consistently struggled to vocalize both symbols and parentheses. In particular, Copilot often failed to read the code blocks at all, skipping the entire content of the "card" in which it was displayed. Finally, we observed inconsistencies across different audio generations from the same prompt. For example, when generating audio for the expert reference, Gemini succeeded in reading symbols in three of the five trials but failed on the other two, highlighting the non-deterministic nature of the output.

### 4.2 Quantitative Results

Our analysis of the automated metrics reveals a clear performance hierarchy among the tested TTS systems when tasked with vocalizing programming code. The analysis of WER, BLEU-4, and METEOR scores reveals significant disparities between models and highlights key factors that influence the intelligibility of CTS output.

As shown in Table 2, Gemini and ChatGPT consistently emerged as the top performers, achieving the lowest WER and highest BLEU-4 and METEOR scores across all the prompts.

For instance, in the "beginner" context with prompt P4, Gemini achieved the lowest WER, with a value of 0.358. In contrast, Copilot underperformed in all the metrics, consistently producing the highest WER and the lowest BLEU-4 and METEOR scores. Its highest WER reached 0.867 and was characterized by a high number of deletions, an error pattern we attribute to our observation that the model struggled to interpret code rendered within the web interface's "card" format. The baseline models, gTTS and ElevenLabs, delivered moderate performance and were generally outperformed by Gemini and ChatGPT. A second key finding was the high performance improvement when using the "expert" reference text compared to the "beginner" reference. For example, ChatGPT's average WER dropped from approximately 0.42 to 0.14 across prompts. Copilot did not show a clear pattern of improvement, exhibiting poor and erratic performance for both reference types. Finally, instructional prompts had a significant impact on performance, particularly for Gemini and Copilot. Performance generally improved as prompts became more specific, with prompt P4 ("treat it as text while reading") often yielding the best results. For Copilot, this prompt lowered its WER from 0.867 to 0.695 in the "beginner" context, and decreased its WER from 0.798 for prompt P1 to 0.355 from prompt P4 in the "expert" context.

[1]GitHub repository: https://github.com/cardiamc/Code-To-Speech_Evaluation

**Table 1: Qualitative evaluation (Mean Opinion Score) based on a 5-point Likert scale, averaged across three raters (including one blind user). Scores reflect performance across five criteria. The mean ± standard deviation across all criteria for each configuration is provided in the final column.**

| Model | Prompt | Reference | Pauses | Symbols | Block Delimiters | Brackets | Comments | Avg ± Std |
|---|---|---|---|---|---|---|---|---|
| ChatGPT | P1 | beginner | 1.0 | 3.3 | 1.0 | 1.3 | 1.0 | 1.52 ± 1.0 |
| ChatGPT | P2 | beginner | 1.0 | 3.0 | 1.0 | 1.0 | 1.0 | 1.40 ± 0.9 |
| ChatGPT | P3 | beginner | 1.0 | 3.3 | 1.0 | 1.0 | 1.3 | 1.52 ± 1.0 |
| ChatGPT | P4 | beginner | 1.0 | 3.7 | 1.3 | 1.3 | 1.0 | 1.66 ± 1.1 |
| Copilot | P1 | beginner | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.00 ± 0.0 |
| Copilot | P2 | beginner | 1.0 | 1.3 | 1.0 | 1.0 | 1.0 | 1.06 ± 0.1 |
| Copilot | P3 | beginner | 1.0 | 2.3 | 1.0 | 1.3 | 1.0 | 1.32 ± 0.6 |
| Copilot | P4 | beginner | 1.0 | 2.7 | 1.3 | 1.3 | 1.0 | 1.46 ± 0.7 |
| Gemini | P1 | beginner | 1.0 | 2.3 | 1.0 | 1.3 | 1.0 | 1.32 ± 0.6 |
| Gemini | P2 | beginner | 1.0 | 4.3 | 3.0 | 2.3 | 1.0 | 2.32 ± 1.4 |
| Gemini | P3 | beginner | 1.0 | 4.0 | **3.3** | 2.7 | 1.3 | 2.46 ± 1.3 |
| Gemini | P4 | beginner | 1.0 | **4.7** | 2.3 | **3.3** | 1.0 | 2.46 ± 1.6 |
| gTTS | – | beginner | 1.0 | 3.3 | 2.7 | 1.0 | 1.0 | 1.80 ± 1.1 |
| ElevenLabs | – | beginner | 1.0 | 1.0 | 1.0 | 1.0 | 2.3 | 1.26 ± 0.6 |
| ChatGPT | P1 | expert | 1.0 | 3.0 | 1.0 | 1.0 | 2.2 | 1.66 ± 1.0 |
| ChatGPT | P2 | expert | 1.0 | 3.3 | 1.0 | 1.0 | 2.4 | 1.66 ± 1.0 |
| ChatGPT | P3 | expert | 1.0 | 3.3 | 1.3 | 1.0 | 2.3 | 1.78 ± 1.1 |
| ChatGPT | P4 | expert | 1.0 | 3.7 | 1.0 | 1.0 | **2.7** | 1.88 ± 1.2 |
| Copilot | P1 | expert | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.00 ± 0.0 |
| Copilot | P2 | expert | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.00 ± 0.0 |
| Copilot | P3 | expert | 1.0 | 2.3 | 1.0 | 1.3 | 1.0 | 1.32 ± 0.6 |
| Copilot | P4 | expert | 1.0 | 2.0 | 1.0 | 1.0 | 1.0 | 1.20 ± 0.4 |
| Gemini | P1 | expert | 1.0 | 3.3 | 1.0 | 1.0 | 2.1 | 1.72 ± 1.1 |
| Gemini | P2 | expert | 1.0 | 3.0 | 2.0 | 1.3 | 2.3 | 1.92 ± 0.8 |
| Gemini | P3 | expert | 1.0 | 4.3 | **3.3** | **4.3** | 2.3 | **3.04 ± 1.4** |
| Gemini | P4 | expert | 1.0 | 4.0 | 2.3 | 4.0 | 2.4 | 2.52 ± 1.4 |
| gTTS | – | expert | 1.0 | 1.3 | 3.3 | 1.0 | 1.3 | 1.58 ± 1.0 |
| ElevenLabs | – | expert | 1.0 | 2.0 | 1.0 | 1.0 | 2.3 | 1.46 ± 0.6 |

## 4.3 Qualitative Error Analysis of Synthesized Audio

By manually inspecting the synthesized audio, we identified recurring error patterns in how TTS systems vocalize programming syntax. These patterns are summarized in Table 3. The analysis reveals systematic differences in how TTS engines handle programming syntax. gTTS exhibited the most literal behavior, explicitly vocalizing structural elements such as "opening curly bracket". While this improves syntactic transparency, it introduces excessive verbosity, increasing cognitive load for expert users. In contrast, Copilot frequently omitted reading the code, in alignment with the high error rate observe in the WER analysis. ElevenLabs and Gemini systems produced phonetic distortions of code-specific constructs: for example, the Python f-string prefix was rendered as "Pffiiis" (ElevenLabs) and "funt" (Gemini).

Additionally, we observed instances of semantic misinterpretation, where comparison operators were incorrectly vocalized as arithmetic operators (e.g., "plus") or with inverted meaning (e.g., "greater than"). Such errors represent a high-severity accessibility issue, as they directly alter the logical semantics of the code.

# 5 Discussion

Our study evaluated five modern TTS systems for code vocalization, revealing substantial gaps between their capabilities and the needs of screen-reader users. In Section 5.1 we analyze the observed limitations, while in Section 5.2 we propose a conceptual framework for effective Code-to-Speech vocalization.

## 5.1 Findings and Limitations of Current TTS Systems

While our results show that models like Gemini and ChatGPT are quantitatively more accurate, our qualitative analysis highlights universal shortcomings that present critical accessibility barriers. The pilot MOS study showed that all systems struggled with fundamental aspects of code reading, including pauses, indentation, and comments. This suggests that current TTS models, despite their fluency with narrative text, are not yet equipped to parse the structural semantics of source code. They appear to treat a code block as a flat sequence of characters, ignoring the indentation that is fundamental for understanding logic in Python and the pauses needed to comprehend scope and separation.

**Table 2: Automated evaluation of TTS systems. WER, BLEU-4, and METEOR scores (mean ± std) averaged over five generations.**

| Model | Prompt | Reference | WER | | BLEU-4 | | METEOR | |
|---|---|---|---|---|---|---|---|---|
| | | | mean | std | mean | std | mean | std |
| ChatGPT | P1 | beginner | 0.417 | 0.005 | 0.417 | 0.005 | **0.445** | 0.006 |
| ChatGPT | P2 | beginner | 0.418 | 0.008 | 0.419 | 0.009 | 0.444 | 0.007 |
| ChatGPT | P3 | beginner | 0.419 | 0.004 | 0.416 | 0.003 | 0.442 | 0.006 |
| ChatGPT | P4 | beginner | 0.419 | 0.013 | 0.412 | 0.016 | 0.442 | 0.013 |
| Copilot | P1 | beginner | 0.867 | 0.055 | 0.012 | 0.018 | 0.106 | 0.028 |
| Copilot | P2 | beginner | 0.804 | 0.038 | 0.037 | 0.016 | 0.151 | 0.017 |
| Copilot | P3 | beginner | 0.780 | 0.036 | 0.043 | 0.020 | 0.153 | 0.016 |
| Copilot | P4 | beginner | 0.695 | 0.066 | 0.111 | 0.063 | 0.208 | 0.043 |
| Gemini | P1 | beginner | **0.412** | 0.027 | **0.421** | 0.029 | 0.442 | 0.024 |
| Gemini | P2 | beginner | **0.381** | 0.028 | **0.484** | 0.052 | **0.473** | 0.037 |
| Gemini | P3 | beginner | **0.379** | 0.028 | **0.486** | 0.047 | **0.460** | 0.023 |
| Gemini | P4 | beginner | **0.358** | 0.029 | **0.506** | 0.045 | **0.494** | 0.028 |
| gTTS | – | beginner | 0.423 | 0.000 | 0.411 | 0.000 | 0.420 | 0.000 |
| ElevenLabs | – | beginner | 0.439 | 0.014 | 0.396 | 0.025 | 0.420 | 0.013 |
| ChatGPT | P1 | expert | **0.133** | 0.008 | **0.797** | 0.004 | 0.772 | 0.011 |
| ChatGPT | P2 | expert | **0.137** | 0.007 | **0.792** | 0.012 | **0.777** | 0.007 |
| ChatGPT | P3 | expert | **0.162** | 0.057 | **0.776** | 0.057 | **0.759** | 0.047 |
| ChatGPT | P4 | expert | **0.140** | 0.008 | **0.791** | 0.011 | 0.772 | 0.012 |
| Copilot | P1 | expert | 0.798 | 0.084 | 0.051 | 0.059 | 0.182 | 0.049 |
| Copilot | P2 | expert | 0.855 | 0.149 | 0.052 | 0.067 | 0.133 | 0.124 |
| Copilot | P3 | expert | 0.662 | 0.048 | 0.153 | 0.042 | 0.262 | 0.019 |
| Copilot | P4 | expert | 0.355 | 0.059 | 0.542 | 0.068 | 0.519 | 0.064 |
| Gemini | P1 | expert | 0.136 | 0.033 | 0.801 | 0.038 | **0.780** | 0.033 |
| Gemini | P2 | expert | 0.310 | 0.242 | 0.610 | 0.283 | 0.660 | 0.251 |
| Gemini | P3 | expert | 0.393 | 0.161 | 0.632 | 0.082 | 0.659 | 0.129 |
| Gemini | P4 | expert | 0.218 | 0.068 | 0.732 | 0.057 | **0.779** | 0.035 |
| gTTS | – | expert | 0.207 | 0.000 | 0.745 | 0.000 | 0.749 | 0.000 |
| ElevenLabs | – | expert | 0.178 | 0.008 | 0.737 | 0.031 | 0.747 | 0.027 |

**Table 3: Examples of how the five TTS systems render programming tokens in audio across.**

| Syntax Token | Expected | Gemini | ChatGPT | ElevenLabs | gTTS | Copilot |
|---|---|---|---|---|---|---|
| `f"{i} is` | "f string, brace i brace, is" | "funt i is" | "f is equal to" | "Pffiiis" | "f (pause) i (pause) is" | (Omitted) |
| `<` | "less than" | "less than" | "greater than" | "plus" | "less than" | (Omitted) |
| `#` | "comment" | (Silence) | "hashtag" | "hash" | "hash code" | (Omitted) |
| `{ }` | "curly braces" | (Silence) | (Silence) | (Silence) | "Opening bracket" | (Omitted) |
| `__init__` | "double underscore init" | "init" | "init" | "init" | "Underscore init" | (Omitted) |

Beyond these core model limitations, our findings also reveal a critical issue at the application layer. The poor performance of Copilot, that provides the worse performances in all the metrics, was characterized by a high number of deletions. From a qualitative analysis, we observed that Copilot failed to read the code blocks at all, skipping the entire content of the "card" in which it was displayed. This "wrapper effect" illustrates how the user interface can be a bigger obstacle to accessibility than the AI model itself.

We also observed the importance of the prompt instructions provided to the TTS. Our results show the importance of providing instructions like treat code as literal text (prompt P4). This prompt often yielded the best results, lowering Copilot's WER from 0.867 to 0.695 in the beginner context. This suggests that the models' attempts to interpret code can introduce errors.

Furthermore, the generative nature of these tools introduces a reliability challenge. Gemini, for example, exhibited a higher standard deviation in all its quantitative scores compared to ChatGPT, as shown in Table 2. Furthermore it also showed inconsistencies in its ability to read symbols across different generations of the same prompt, a behavior also observed by the the author when evaluating with the MOS.

**Table 4: Design considerations for effective Code-to-Speech (CTS) vocalization, derived from observed error patterns.**

| Dimension | Identified Challenge | Design Consideration |
|---|---|---|
| **Syntactic Fidelity** | Standard NLP pipelines often strip "non-essential" punctuation, leading to critical omissions of brackets, operators, and delimiters. | **Domain-Specific Tokenization:** CTS systems may benefit from bypassing standard text normalization and adopting a consistent pronunciation dictionary for programming symbols (e.g., rendering {} as "curly brackets"). |
| **Semantic Preservation** | TTS engines may misinterpret context, reading < as "greater than" or = inconsistently, potentially altering program logic. | **Context-Aware Processing:** CTS systems could benefit from integrating structured representations such as Abstract Syntax Trees (ASTs) or semantic parsers to disambiguate symbols based on context (e.g., distinguishing assignment = from equality ==). |
| **Structural Transparency** | Treating code as a flat sequence of characters obscures hierarchical organization, including scope, nesting, and indentation. | **Auditory Cues & Prosody:** Rather than relying solely on verbose narration (e.g., "space space"), CTS systems could leverage prosodic modulation (pauses, pitch changes) or non-speech auditory cues (earcons/spearcons) to convey structural information. |
| **Cognitive Efficiency** | Highly literal readings may cause listening fatigue for expert users, while overly fluid readings may hinder comprehension for novices. | **Configurable Verbosity:** CTS systems could support adaptable verbosity profiles, where novice users receive explicit syntax vocalization and expert users benefit from more concise renderings. |

While this exploratory study involved a small group of three participants, the inclusion of a blind user provided critical insights into the lived experience of navigating code via TTS. This addresses the potential bias of sighted raters. However, we acknowledge that these findings remain preliminary and require further validation with a larger, more diverse user base to account for varying levels of programming expertise. Furthermore, our evaluation used a single educational eBook, and the systems' performance might differ on more complex codebases or other programming languages.

## 5.2 Effective Code Vocalization

A central challenge in evaluating and developing CTS systems is defining what constitutes an effective vocalization of source code. Unlike natural language, which relies on narrative flow, source code encodes meaning through a combination of syntax, spatial structure, and context-dependent semantics. Translating these elements into the auditory modality requires preserving both correctness and interpretability to ensure accurate comprehension.

Drawing on prior work on code phonology and auditory programming environments [7, 8, 18]and error patterns observed in our study, we identify four key dimensions of effective CTS. Table 4 summarizes these dimensions, links observed challenges to design considerations, and provides a conceptual lens for interpreting current TTS systems. First, syntactic fidelity concerns the accurate and consistent rendering of symbols, operators, and delimiters. Our results show that general-purpose TTS pipelines frequently omit or alter these elements, leading to incomplete or misleading representations of the code. Second, semantic preservation refers to maintaining the logical meaning of the program during vocalization. Misinterpretations of operators or language-specific constructs can directly alter program semantics, representing critical accessibility failures. Third, structural transparency captures the ability to convey hierarchical organization, including indentation, block boundaries, and nesting. Since these features are visually encoded, their absence in speech leads to a "flattening" of structure that hinders comprehension, necessitating the use of prosody or non-speech auditory cues (e.g., earcons or spearcons [24]). Finally, cognitive efficiency reflects the trade-off between completeness and listening effort. While highly literal renderings may benefit novices, they can introduce verbosity and listening fatigue for expert users, motivating the need for adaptive verbosity strategies.

Together, these dimensions suggest that CTS should be understood not merely as a speech synthesis problem, but as a structured multimodal translation task. Addressing these challenges likely requires moving beyond conventional text normalization pipelines toward context-aware approaches that incorporate structural representations, such as ASTs.

# 6 Conclusion and future work

We evaluated five AI-driven TTS systems to vocalize source code embedded in narrative text. Using a double evaluation methodology, based on qualitative (MOS) and quantitative metrics (WER, BLEU-4, METEOR), we found that while there are performance differences among models, all systems struggle with the fundamental task of reading code in a structurally meaningful way. Our quantitative analysis, using metrics like WER, revealed that while systems like Gemini and ChatGPT were more accurate, reaching a WER value of almost 0.1, others like Copilot performed poorly, often omitting critical code segments. Qualitatively, key syntactic elements in programming, such as indentations, comments, parentheses, and symbols, were problematic for the evaluated models.

Interpreting our findings through the dimensions of syntactic fidelity, semantic preservation, structural transparency, and cognitive efficiency highlights that current TTS systems are not yet able to provide reliable and accessible code vocalization. These limitations are particularly impactful for blind and visually impaired users, for whom precise auditory rendering is essential for comprehension.

We emphasize that our study is exploratory and subject to several limitations, including the use of a single dataset and a small-scale qualitative evaluation involving three participants. Future work should expand the evaluation to larger and more diverse user groups, including blind and visually impaired programmers, and explore a wider range of programming materials and formatting styles. Additionally, future research should investigate the design of syntax-aware and context-aware TTS architectures, potentially integrating structured representations such as abstract syntax trees. Lastly, future works may extend beyond JavaScript and Python to other languages.

## Acknowledgments

This work was carried out within the STEMMA PRIN project thanks to the funding by the European Union - Next Generation EU, Mission 4 Component 2 CUP B53D23019500006.

## References

[1] Nusaibah M. Al-Ratta and Hend S. Al-Khalifa. 2013. Teaching programming for blinds: A review. In *Fourth International Conference on Information and Communication Technology and Accessibility (ICTA)*. IEEE, Hammamet, Tunisia, 1–5. doi:10.1109/ICTA.2013.6815285

[2] Josh Andres, Chris Danta, Andrea Bianchi, Sahar Farzanfar, Gloria Milena Fernandez-Nieto, Alexa Becker, Tara Capel, Frances Liddell, Shelby Hagemann, Ned Cooper, Sungyeon Hong, Li Lin, Eduardo Benitez Sandoval, Anna Brynskov, Hubert Dariusz Zając, Zhuying Li, Tianyi Zhang, and Arngeir Berge. 2025. A Scenario-Based Design Pack for Exploring Multimodal Human–GenAI Relations. In *Proceedings of the 27th International Conference on Multimodal Interaction (ICMI '25)*. Association for Computing Machinery, New York, NY, USA, 145–154. doi:10.1145/3716553.3750749

[3] Alvaro Becerra, Ruth Cobos, and Charles Lang. 2025. Enhancing online learning by integrating biosensors and multimodal learning analytics for detecting and predicting student behaviour: a review. *Behaviour & Information Technology* 0, 0 (2025), 1–26. doi:10.1080/0144929X.2025.2562322

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

[5] Md Jalal Uddin Chowdhury and Ashab Hussan. 2023. A review-based study on different Text-to-Speech technologies. *arXiv preprint arXiv:2312.11563* (2023), 1–4.

[6] Md Ehtesham-Ul-Haque, Syed Mostofa Monsur, and Syed Masum Billah. 2022. Grid-Coding: An Accessible, Efficient, and Structured Coding Paradigm for Blind and Low-Vision Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 44, 21 pages. doi:10.1145/3526113.3545620

[7] J Louise Finlayson and Chris Mellish. 2005. The'audioview'-providing a glance at Java source code. In *Proceedings of ICAD 05-Eleventh Meeting of the International Conference on Auditory Display*. Georgia Institute of Technology, Limerick, Ireland, 127–133.

[8] Felienne Hermans, Alaaeddin Swidan, and Efthimia Aivaloglou. 2018. Code Phonology: An Exploration into the Vocalization of Code. In *Proceedings of the 26th Conference on Program Comprehension*. ACM. doi:10.1145/3196321.3196355

[9] Stephen Simei Kimogol. 2023. *Accessibility of digital teaching materials. Identifying barriers faced by students with visual impairment in digital teaching materials*. Master's thesis. Oslomet-storbyuniversitetet.

[10] Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability.

[11] Alon Lavie and Michael J Denkowski. 2009. The METEOR metric for automatic evaluation of machine translation. *Machine translation* 23, 2 (2009), 105–115.

[12] Ruskin Raj Manku, Yuzhi Tang, Xingjian Shi, Mu Li, and Alex Smola. 2025. EmergentTTS-Eval: Evaluating TTS Models on Complex Prosodic, Expressiveness, and Linguistic Challenges Using Model-as-a-Judge. *arXiv preprint arXiv:2505.23009* (2025).

[13] Sadia Nowrin and Keith Vertanen. 2023. Programming by Voice: Exploring User Preferences and Speaking Styles. In *Proceedings of the 5th International Conference on Conversational User Interfaces* (Eindhoven, Netherlands) *(CUI '23)*. Association for Computing Machinery, New York, NY, USA, Article 20, 13 pages. doi:10.1145/3571884.3597130

[14] Youngja Park, Siddharth Patwardhan, Karthik Visweswariah, and Stephen C Gates. 2008. An empirical analysis of word error rate and keyword error rate.. In *Interspeech*, Vol. 2008. 2070–2073.

[15] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y. Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3173574.3174192

[16] Sarantos Psycharis, Paraskevi Theodorou, Ypatios Spanidis, and Pantelis Kydonakis. 2021. Teaching programming skills to blind and visually impaired learners. In *The Learning Ideas Conference*. Springer, 270–279.

[17] Ehud Reiter. 2018. A structured review of the validity of BLEU. *Computational Linguistics* 44, 3 (2018), 393–401.

[18] Dominic Roberts and Karlton Weaver. 2011. Audio Aids in Source Code. (2011).

[19] Jaime Sánchez and Fernando Aguayo. 2005. Blind learners programming through audio. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1769–1772. doi:10.1145/1056808.1057018

[20] Jaime Sánchez and Fernando Aguayo. 2006. APL: Audio Programming Language for Blind Learners. In *International Conference on Computers for Handicapped Persons*. Springer, 1334–1341. doi:10.1007/11788713_192

[21] Elvis Saravia. 2022. Prompt Engineering Guide. *https://github.com/dair-ai/Prompt-Engineering-Guide* (12 2022).

[22] Issac G Tijerina and Soma Datta. 2023. A Survey on Current Speech to Text Analysis to Help Programmers Dictate Code. In *Proceedings of the 2023 6th International Conference on Software Engineering and Information Management* (Palmerston North, New Zealand) *(ICSIM '23)*. Association for Computing Machinery, New York, NY, USA, 50–57. doi:10.1145/3584871.3584878

[23] Alakbar Valizada, Sevil Jafarova, Emin Sultanov, and Samir Rustamov. 2021. Development and evaluation of speech synthesis system based on deep learning models. *Symmetry* 13, 5 (2021), 819.

[24] Bruce N Walker, Jeffrey Lindsay, Amanda Nance, Yoko Nakano, Dianne K Palladino, Tilman Dingler, and Myounghoon Jeon. 2013. Spearcons (speech-based earcons) improve navigation performance in advanced auditory menus. *Human Factors* 55, 1 (2013), 157–182.