Universal Access in the Information Society (2026) 25:94
https://doi.org/10.1007/s10209-026-01351-6

REVIEW

Understanding design and accessibility issues in block-based programming for people with disabilities: a literature review

Rubel Hassan Mollik[1] · Wajdi Aljedaani[2] · Stephanie Ludi[1]

Received: 3 November 2025 / Accepted: 9 June 2026 / Published online: 30 June 2026

Abstract
Accessibility barriers in block-based programming environments (BBPEs) continue to hinder equitable participation in computer science education, particularly for learners with disabilities. While prior research has examined accessibility challenges for programmers with visual impairments, BBPEs introduce additional barriers due to their highly visual and drag-and-drop nature. This paper presents a systematic literature review of 22 peer-reviewed studies focusing exclusively on accessibility in virtual BBPEs such as Blockly, Scratch, and App Inventor. The review identifies six core categories of barriers–interaction and input limitations, navigation and orientation difficulties, inaccessible content and outputs, cognitive load, assistive technology conflicts, and educational or technical constraints. It also synthesizes proposed solutions, including accessible keyboard navigation, voice-based interaction, screen reader integration, program navigation models, and inclusive interface design. Despite promising developments, such as Accessible Blockly and Blocks4All, existing research remains fragmented and primarily focuses on visual impairments. This review consolidates the state of knowledge, exposes underexplored disability groups, and outlines a research roadmap for designing universally accessible block-based programming environments that promote inclusive computer science education.

Keywords Block-based programming · Blockly · Scratch · Accessibility · Screen reader · Students with disabilities · K-12 computing education

1 Introduction

In today's digital age, computational thinking [1] and programming skills have become essential competencies in education systems worldwide. One popular approach to teaching programming to novices, particularly in K-12 settings, is through visual programming paradigms [2]. Block-based programming languages (BBPLs) [3] allow users to construct programs by dragging and dropping graphical blocks that represent textual code elements, thereby reducing syntactic complexity and making programming more accessible for beginners [4]. This method gained significant traction following initiatives such as the Computer Science for All (CSforAll) [5] program launched by former U.S. President Barack Obama in 2016, which aimed to expand access to computer science education. Popular virtual block-based programming environments (BBPEs), such as Scratch [4], Microsoft MakeCode [6], MIT App Inventor [7], and Google Blockly [8], enable students to practice programming concepts in an engaging, interactive manner [9].

However, the accessibility of these block-based platforms remains a critical concern [10]. The inaccessibility of these environments creates barriers to computer science education for learners with disabilities [11]. While these tools promote inclusivity for neurotypical learners by bridging the gap between abstract concepts and tangible outcomes, they often exclude those with visual, motor, cognitive, or other impairments. For equitable access to computer science education in classrooms [12], these environments need to be accessible to all students. Learners who are blind or have

✉ Rubel Hassan Mollik
rubelhassanmollik@my.unt.edu

Wajdi Aljedaani
waljedaani@sdaia.gov.sa

Stephanie Ludi
Stephanie.Ludi@unt.edu

[1] University of North Texas, Denton, TX, USA

[2] Saudi Data & Artificial Intelligence Authority, Riyadh, Kingdom of Saudi Arabia

low vision often face inconsistent keyboard focus, limited screen reader output, and spatial layouts that are difficult to perceive non-visually [13]. Motor disabilities may hinder precise manipulation of blocks, while cognitive impairments could exacerbate issues with interface complexity or information overload [14, 15]. This exclusion is particularly concerning in the context of inclusive education mandates, such as the United Nations Convention on the Rights of Persons with Disabilities (UNCRPD) [16] and the U.S. Individuals with Disabilities Education Act (IDEA) [17], which emphasize equitable access to learning opportunities.

The growing emphasis on digital inclusion underscores the urgency of making BBPLs accessible [3]. Despite the popularity of BBPLs in K-12 settings, where they are used to foster 21st-century skills [18], research on their accessibility for learners with disabilities remains fragmented and limited. This gap is compounded by the rapid evolution of BBPLs, with new tools emerging without built-in accessibility features, leading to ad-hoc adaptations rather than systemic solutions [19]. Consequently, educators and developers lack consolidated guidance on barriers, effective interventions, and best practices, hindering the design of truly universal programming environments [20].

Several studies have proposed innovative solutions to address these accessibility issues [21, 22]. For example, Mountapmbeme and Ludi [23] introduced Accessible Blockly, a prototype block-based programming library offering keyboard-only navigation and screen reader support for blind or visually impaired students. Similarly, Tabassum et al. [24] developed an accessible environment that renders code in a 2D grid format, enabling blind and low-vision programmers to navigate structured puzzles. Okafor et al. [25] created a voice-enabled version of Blockly, designed for users with motor impairments, reducing the need for mouse or keyboard interactions. Additionally, Milne et al. [26] proposed Blocks4All, a touch-based programming environment for iPad that uses VoiceOver for visually impaired students. Similarly, Bulovic et al. [27] developed an accessible mobile app compatible with both Android and iOS.

Despite these valuable contributions, no prior research has systematically synthesized the literature on the accessibility of block-based programming environments for individuals with various disabilities. A related study by Khalajzadeh et al. [28] conducted a systematic literature review on low-code development environments, which broadly encompasses visual programming languages and Integrated Development Environments (IDEs), as Unified Modeling Language (UML), domain-specific visual languages, model-driven architecture, and block-based programming. However, their focus on accessibility was limited to users with visual impairments, without a detailed discussion of the specific challenges and barriers associated with block-based programming. Rezende et al. [29] provided an early review of accessibility in block-based programming for people with visual impairments, where our review provides a broader synthesis of accessibility barriers, solutions, participant populations, and design implications across virtual BBPEs for learners with disabilities. Aboubakar et al. [30] conducted a literature review to identify accessibility barriers in programming for people with visual impairments, analyzing text-based, tangible, and block-based programming approaches. However, their review focused on general programming accessibility barriers targeting the students with blindness and visual impairments (BVI). As a result, block-based programming received only limited attention, and accessibility issues in this domain were discussed briefly rather than as a central focus.

In this study, we systematically synthesize the literature to catalog accessibility challenges in block-based programming and expose unresolved gaps. To our knowledge, this is the first study to target accessibility exclusively in virtual block-based environments for learners with diverse disabilities. The resulting findings offer a clear roadmap for inclusive design and future research toward block-based programming environments that are accessible by design.

2 Scope and research question

2.1 Scope

The primary objective of this study is to systematically review the existing literature on the accessibility of virtual block-based programming environments (BBPEs) for learners with disabilities. By synthesizing prior scholarship, this study aims to elucidate the challenges faced by learners with disabilities in these environments and to examine the solutions proposed to address them. To achieve this, we outline the following contributions:

1. Identify and analyze the challenges and barriers that learners with various disabilities face when accessing BBPEs and examining how different types of impairments interact with the visual and interactive nature of these environments.
2. Examine and discuss the solutions, adaptations, and assistive technologies proposed in the literature to mitigate these accessibility challenges.
3. Delineate the gaps in the existing literature regarding accessibility features for particular types of impairments.
4. Propose recommendations that integrate these findings for designing accessible BBPEs that align with universal design principles and accessibility standards.

As illustrated in Fig. 1, the scope of this study focuses on the intersection of domains: learners with disabilities, virtual BBPEs, and assistive technologies or accessible tools/plugins. This intersection represents the specific context where accessibility challenges emerge and solutions must be implemented.

Learners with disabilities encompass K-12 students or adult novices who require assistive technologies or accessible tools to interact with virtual block-based programming platforms. This population represents individuals with disabilities, including visual, auditory, motor, and cognitive impairments. We specifically target virtual visual programming platforms that support block-based programming on digital devices such as desktops, tablets, or smartphones without relying on physical tools, such as tangible blocks, during programming activities. We exclude tangible or hybrid programming tools to narrow our scope to virtual BBPEs. This approach allows us to explore the strengths and limitations of software-based accessibility challenges and solutions.

2.2 Research questions

Guided by the scope outlined in Sect. 2.1, we formulate the following research questions to structure our literature review and analysis. These questions are designed to systematically explore the current landscape of accessible block-based programming, identify existing challenges, and evaluate proposed solutions to inform future development efforts.

RQ1: What patterns characterize accessibility research on virtual block-based programming environments for learners with disabilities to date?

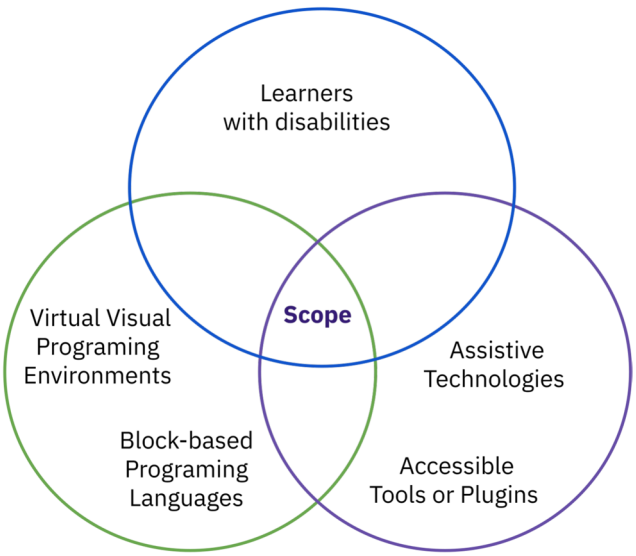

Fig. 1 Scope of the literature

RQ1.1: How has the number of studies evolved over time on the accessibility of block-based programming platforms?
RQ1.2: How are participants distributed across block-based programming accessibility studies by sample size range, participant type, and research method?
RQ1.3: Which technologies and platforms are employed in the development of accessible block-based programming environments?
RQ1.4: What input modalities and interaction techniques have been studied or proposed in the literature?

RQ2: What accessibility challenges do learners with disabilities face that are directly attributable to the structure and functionality of block-based programming environments/tools?
RQ3: What are the current proposed solutions to reported barriers?

3 Methodology

We conducted this systematic literature review (SLR) following the established guidelines and procedures outlined by Kitchenham et al. [31, 32] to comprehensively identify and synthesize research on accessibility of virtual block-based programming environments (BBPEs) for students with disabilities. Our methodology ensures a rigorous, reproducible, and transparent approach to identifying, evaluating, and analyzing relevant literature. The review process is structured around three primary phases: Planning, Conducting Review, and Data Synthesis and Analysis.

3.1 Planning

The planning phase focused on defining the search strategy, including keyword selection, database identification, and query formulation to ensure comprehensive coverage of the published articles.

3.1.1 Keywords selection

The two authors conducted a manual pilot search in two prominent scholarly databases, IEEE Xplore and ACM Digital Library, to identify relevant terms. From the pilot search results, the authors identified frequently used keywords in the literature, including synonyms for block-based programming, accessibility, and disability groups. Through collaborative discussion, the two lead authors refined and categorized these terms into three groups: *T1* - block-based

Table 1 Search keywords used to perform digital library searches

T1 - Block-based programming		T2 - Accessibility and assistive technologies		T3 - User group with disabilities
block-based programming, visual programming, block programming, block-based environments, block programming languages, visual programming languages, visual programming environments, graphical programming, blockly, scratch, app inventor, makecode	AND	accessibility, a11y, accessible programming, assistive technology, screen reader, keyboard navigation, audio feedback	AND	blind, low vision, visual impairment, bvi, disabilities, visually impaired, motor impairments, cognitive impairments, autism spectrum disorder, learning disabilities, K-12, adhd

programming, *T2* - accessibility and assistive technologies, and *T3* - disability user groups. This categorization facilitated the construction of Boolean search queries for scholarly databases in a later step. Table 1 summarizes the selected keywords, which were iteratively refined to capture variations in terminology while avoiding overly broad terms (Fig. 2).

3.1.2 Search process

For the search process, we conducted searches across different digital libraries. We selected six prominent scholarly databases: ACM Digital Library, IEEE Xplore, ScienceDirect, Scopus, SpringerLink, and Web of Science, all of which are considered reliable. These databases are well-known for their extensive indexing of high-quality journal articles and conference proceedings related to visual programming, accessibility, and human-computer interaction. Additionally, these databases provide query interfaces that enable advanced searches using Boolean queries based on various metadata of published articles, including title, abstract, keywords, language, and publication title.

Table 2 Digital libraries search keywords

Digital Library	Example Query
	`Title:(block* OR "visual" OR "graphical") AND Title:("programming" OR "language" OR "environment") AND Abstract:(blockly OR scratch OR "app inventor" OR makecode OR block* OR BBPE OR "visual programming" OR code.org) AND Abstract:(blind OR "low vision" OR "Visual impairment" OR "Visually impaired" OR disab* OR "k-12" OR bvi OR accessib* OR "cognitive impairment*" OR "motor impairment*" OR "autism spectrum disorder" OR "a11y" OR "assistive technology" OR "screen reader" OR "keyboard navigation" OR "audio feedback")`

To ensure reproducibility and deterministic search results, we tailored search queries specific to each platform, using the AND operator to combine high-level terms such as *T1*, *T2*, and *T3*, and the OR operator within each category. Queries were restricted to English-language publications and covered all relevant studies up to September 2025, which marks the endpoint of our data collection. For example, the query used for the Digital Library is shown in Table 2.

We conducted searches using these queries across all the selected scholarly databases. We found a total of 645 articles in our initial search. After removing duplicates, 492 records remained for screening. Metadata extracted at this stage included titles, abstracts, publication years, and sources, facilitating subsequent filtering.

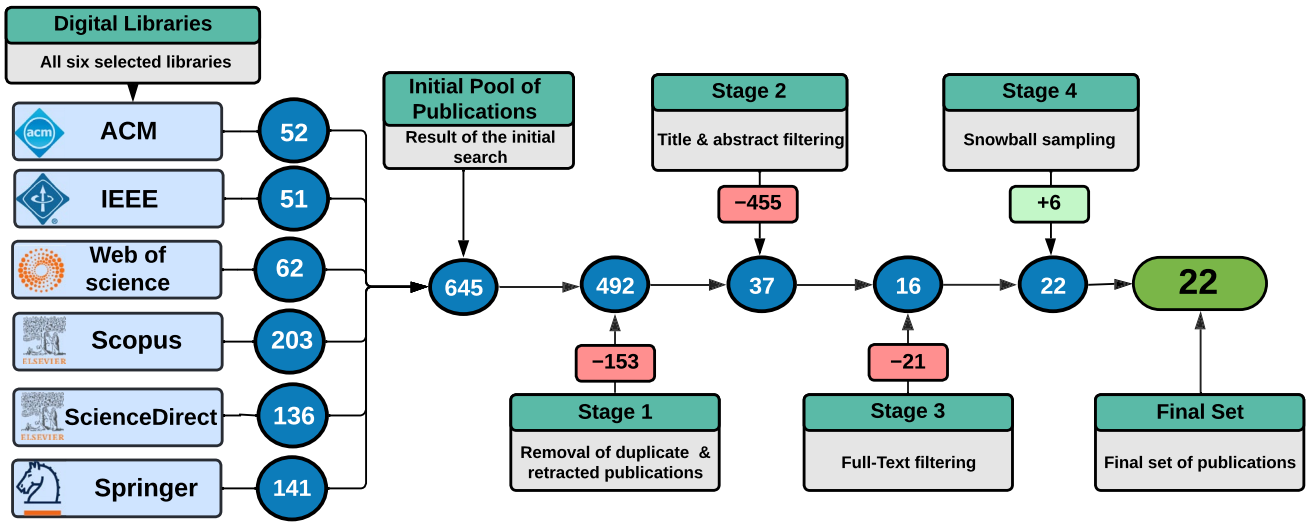

Fig. 2 Overview of publications filtering process

3.2 Conducting the review

We employed a multi-stage filtering process to identify and select the final set of primary studies for this review. This phase involved three key stages: (1) defining exclusion and inclusion criteria, (2) performing a rigorous screening and filtering of the literature, and (3) applying snowballing to capture any additional relevant papers.

3.2.1 Stage (1): exclusion and inclusion criteria

The goal of this literature review is to identify and analyze relevant publications that focus on the accessibility of virtual block-based programming environments. To keep our study relevant and maintain the quality of the studies to be reviewed, we have adopted exclusion and inclusion criteria [33]. These were applied sequentially during title/abstract screening and full-text review.

Inclusion Criteria:

1. The paper addresses accessibility issues or solutions in BBPEs for students in K-12, higher education, or equivalent settings.
2. The paper discusses the design of accessible BBPEs aimed at eliminating barriers for specific disability groups.
3. The paper evaluates custom BBPE tools, plugins, or adaptations as solutions to accessibility barriers.
4. The paper assesses the accessibility of existing BBPEs and identifies associated challenges or barriers.
5. The paper examines limitations of interaction modalities or assistive technologies in BBPEs.**Exclusion Criteria:**

1. The paper without full-text access online.
2. The manuscript is not written in the English language.
3. The paper does not address accessibility for users with disabilities.
4. The paper discusses accessibility in computer science curricula, but with no specific focus on BBPEs.
5. The paper focuses on non-virtual BBPEs, such as tangible or robotic programming involving physical tools.
6. The paper focuses solely on teaching practices for BBPEs without addressing accessibility.
7. The paper is an abstract, poster, or short paper lacking empirical evidence such as evaluations, user studies, or prototypes.

3.2.2 Stage (2): screening and filtering

Based on our exclusion and inclusion criteria, the first two authors independently conducted an initial screening of titles and abstracts, rated each paper's relevance as *relevant*, *maybe*, or *not relevant*. Screening agreement was quantified using Cohen's kappa (κ) [34]. Inter-reviewer agreement on these ordered categories was quantified with *weighted* Cohen's kappa ($\kappa_w = 0.86$), where two reviewers showed almost perfect agreement [35]. Any disagreements were resolved through discussion to reach consensus and improve screening consistency. This screening process excluded 455 irrelevant records. Most of these discarded papers focused on general CS education, K-12, or robotics and visual programming languages (VPLs), without addressing accessibility, while others addressed accessibility but not block-based environments. Many were false positives in which "block" or "visual" referred to unrelated fields such as blockchain, signal processing, vision science, or urban planning. This process reduced the pool to 37 studies for consideration in the next full-text phase. During the full-text review, following discussion among the authors, several abstract-only and poster papers were excluded for insufficient empirical depth, such as the absence of user testing or evaluation. This process yielded 16 primary studies.

3.2.3 Stage (3): snowballing

Finally, we applied backward [36] and forward snowballing techniques [37] to identify additional relevant studies from the initial pool and ensure that no significant research was overlooked. References and citations of the 16 studies were manually screened using Google Scholar and Scopus, yielding six additional relevant articles that met our criteria. No further iterations were required, resulting in the addition of six papers to the list. Therefore, our study consists of the final set of 22 primary studies, which are presented in Table 9 (see Appendix Sect. 8).

3.3 Data synthesis and analysis

The 22 primary studies were independently read in full by two authors. Data extraction focused on themes aligned with our research questions (RQs), including accessibility challenges, proposed solutions, interaction mechanisms, and evaluation methods. Using a shared spreadsheet, authors extracted structured data based on the themes, alongside open-ended notes, for qualitative synthesis, following the reflexive approach of Braun et al. [38]. This involved an initial open coding phase to identify recurring concepts across studies, followed by axial coding to group related codes into higher-order thematic categories. The final key themes that emerged from this process included:

- Accessibility challenges and barriers in BBPEs
- Interaction mechanisms and assistive technologies
- Design recommendations and proposed solutions

- Target disability groups and educational contexts

Based on the results of the coding process, the data were organized and refined to identify similarities, differences, and relationships among the publications, as discussed by the first two authors. Answers to the research questions were then organized and synthesized following the analysis. The results of our analysis are discussed in the remainder of this paper.

The authors met weekly via virtual meetings to discuss emerging codes and resolve discrepancies. Once coding was finalized, data were synthesized to identify commonalities, differences, and gaps in the existing research. This thematic structure formed the basis for answering our research questions, and the results of this analysis are presented in the results and discussion sections of this paper.

4 Results

4.1 RQ1: what is the current distribution of studies on the accessibility of virtual block-based programming environments?

This research question examines the temporal distribution of studies that address accessibility in block-based programming environments. Specifically, it examines how publication patterns and research trends have evolved over time, providing insight into the field's development, the momentum of scholarly activity, and the broader context in which current work is situated.

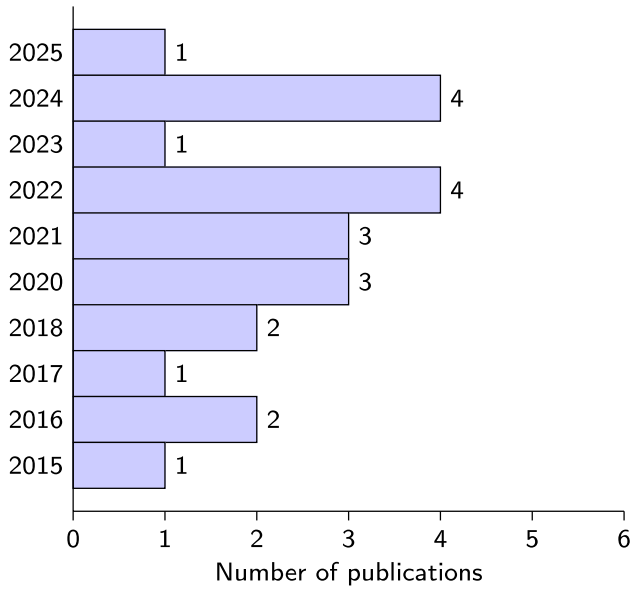

Fig. 3 Publications by year

4.1.1 RQ1.1: how has the number of studies evolved over time on the accessibility of block-based programming platforms?

To analyze the evolution of accessibility research in block-based programming, we systematically categorized selected studies by year of publication. Figure 3 illustrates the number of research papers published annually. The number of studies on the accessibility of block-based programming has grown linearly, with a significant increase in focus since the earliest works in 2015. This trajectory can be understood as a progression through distinct phases, evolving from identifying core accessibility barriers to developing inclusive design approaches and accessible environments that cater to a broader range of disabilities.

Early studies from 2015 to 2017 investigated core problems and began exploring foundational solutions, with a primary focus on users with visual and motor impairments. These studies have highlighted that mainstream block-based programming environments, such as Scratch, are inaccessible to blind users due to their reliance on visual shape cues and mouse-driven drag-and-drop interactions [39]. Additionally, Scratch 2.0 was based on Flash technology, which made accessibility impossible [40]. During this period, researchers explored alternative input methods, such as keyboard navigation, auditory cues, and voice input, to enhance code navigation and comprehension. For example, early efforts explored Vocal User Interfaces (VUIs) such as Myna, which added a voice-controlled background layer over Scratch to enable programming by voice for learners with motor impairments [41]. Non-visual interaction models, such as Pseudospatial Blocks (PB) [39], have been proposed. Researchers also began modifying the Blockly library to support screen reader compatibility and keyboard navigation [42]. The use of WAI-ARIA attributes was identified as a key technical approach to make the visual structure of blocks understandable to assistive technologies [42].

From 2018 to 2020, research expanded beyond web-based prototypes for blind users to include touchscreen environments and the specific needs of users with cognitive impairments. Blocks4All [26] marked a significant shift by developing and evaluating an accessible block-based environment for the iPad and VoiceOver. This work investigated usable touch interactions for children with visual impairments, comparing methods, for instance, "audio-guided drag and drop" with "select, select, drop". An AST-based structural navigation model, based on CodeMirror-Blocks (CMB), was proposed to improve navigation [43]. Studies began to incorporate educators' perspectives by involving them in research, design, and evaluation processes [44]. Findings revealed that mainstream tools, such as Scratch, were rarely used in practice, with educators instead relying

on hybrid platforms, for example, Swift Playgrounds, despite navigation and editing difficulties [13]. These studies reinforced the inaccessibility of mainstream BBPEs and highlighted the need for features such as focus navigation and connection validation [45].

From 2021 to 2022, research expanded to develop accessible block-based environments for new disability groups. Self-voicing solution introduced with refined auditory feedback to address screen-reader learning barrier to entry for K-12 students [46]. User-centered and iterative design processes guided the creation of Accessible Blockly [23], which was subsequently tested with students who have visual impairments. Research also captured teachers' perspectives to identify systemic accessibility gaps in existing platforms [30]. Voice-based Blockly added a new dimension by focusing on users with upper-limb motor impairments (ULMI), providing speech as an alternative input modality and voice-based commands [25].

From 2023 onward, researchers focused on designing practical, tool-centric, accessible platforms for block-based programming. Design recommendations were proposed to co-create accessible programming platforms for children with autism spectrum condition (ASC) by identifying problems through user group studies [47]. Hardware-accelerated graphics pipelines, such as Quorum Blocks, were designed to make 2D/3D graphical output accessible [21]. Mobile applications, for instance, OctoStudio, integrate screen readers, sensors, and sonification to provide inclusive experiences for both blind/visually impaired learners and sighted learners. GridWorld, a web-based BBPL, offered keyboard accessibility and debugging support. Meanwhile, Accessible Blockly continued to be refined for greater stability and usability [48].

Table 3 Distribution of participants in BBPE accessibility studies by range, method, and type

Range	PS	Method	Total	Participants Type
0	PS19	System Design	0	N/A
	PS21	System Design	0	N/A
1–5	PS4	Expert Review (iterative)	2	Professionals (Teachers/TVIs/Experts): 2
	PS11	Heuristic evaluation	2	Professionals (Teachers/TVIs/Experts): 2
	PS13	Usability study	4	Blind/Low Vision: 4
	PS2	Semi-structured interview	5	Blind/Low Vision: 5
	PS10	Mixed methods (survey + usability study)	5	Motor disabilities (Adult): 5
	PS17	Semi-structured interview	5	Blind/Low Vision: 5
6–10	PS3	Heuristic evaluation	6	Professionals (Teachers/TVIs/Experts): 6
	PS8	Focus group	6	Professionals (Teachers/TVIs/Experts): 6
	PS6	Semi-structured interview	7	Professionals (Teachers/TVIs/Experts): 7
	PS20	Mixed methods (survey + controlled user study)	7	Blind/Low Vision (Adult): 7
	PS22	Mixed methods (survey + usability study)	7	Motor disabilities (Adult): 2, Graduate students without disabilities: 5
	PS9	Mixed methods (survey + usability study)	9	Motor disabilities (Adult): 9
	PS15	Usability study	9	Blind/Low Vision (Adult): 9
	PS18	Semi-structured interview	9	Cognitive disabilities: 9
11–15	PS1	Semi-structured interview	11	Blind/Low Vision (Adult): 11
	PS7	Semi-structured interview	12	Blind/Low Vision (Adult): 12
	PS12	Semi-structured interview	12	Professionals (Teachers/TVIs/Experts): 12
	PS14	Survey	12	Professionals (Teachers/TVIs/Experts): 12
	PS16	Mixed methods (usability test + interview)	13	Blind/Low Vision (Adult): 13
20+	PS5	Focus group	26	Professionals (Teachers/TVIs/Experts): 4, Blind/Low Vision (Adult): 1, Blind/Low Vision: 6, Students with multiple disabilities: 15

4.1.2 RQ1.2: how are participants distributed across block-based programming accessibility studies by sample size range, participant type, and research method?

Participants Distribution. In the reviewed studies, the distribution of participants shows a clear imbalance across sample size (# of participants), participant type, and research method as presented in Table 3. Out of 22 studies, 73% (n=16) involve small groups, with the typical number of participants being fewer than 10, and prioritize adults with visual impairments or educational professionals over K-12 learners. The small number of participants in these studies is a frequently acknowledged limitation, which studies repeatedly mention as "difficulty in recruiting from this population" as a reason for having a small number of participants [26, 43, 49]. Accessing K-12 students presents unique logistical barriers that often lead researchers to use smaller samples or alternative recruitment strategies [49]. To address these constraints, researchers frequently recruit Teachers of the Visually Impaired (TVIs) and other educational professionals as proxies [10, 13, 44, 47]. Additionally,

the tendency to recruit adults is directly linked to recruitment challenges, as adults are easier to recruit [23, 25, 43, 48]. However, sometimes involving experienced adult programmers with disabilities is a deliberate research strategy to produce cleaner, more focused results [39].

Research method. The studies employed diverse methodologies, reflecting the various stages of research from conceptual design to empirical validation. Evaluations are often formative or exploratory, relying on methods such as semi-structured interviews, usability tests, or heuristic evaluations with limited samples [39, 42]. Semi-structured interviews were the most frequently used method, reported in 32% (n = 7) of the primary studies. This method was often employed to gather rich, qualitative data from the target population, such as users with disabilities or experienced professionals, including teachers and domain experts. Heuristic evaluations (n=2) and expert reviews typically involve only a handful of professionals or Teachers of the Visually Impaired, producing prescriptive recommendations that inform early-stage prototypes [13, 46]. Usability studies (n = 2) and mixed-methods studies (n = 4) were commonly used in testing new prototypes for users with Visual Impairments (VI) or Motor Disabilities [39–41, 43]. These approaches engage end-users more directly, combining performance data with qualitative feedback to capture both what participants can accomplish and how they experience the tools [23, 25, 26], with some studies combining initial pilot testing on adults without disabilities with smaller-scale evaluations on target users [41]. A smaller number of studies employ surveys (n = 1), focus groups (n = 2), or expert review (n = 1), which situate accessibility challenges within broader educational contexts and generate explanatory models of user interaction [10, 49]. A couple of studies employed the system design (n = 2) method to propose design prototypes of novel accessible programming tools and libraries, and to outline initial design considerations. These studies did not involve any formal participant evaluation [39, 42].

Table 4 Accessible custom BBPEs reported in the literature

Base technology	Tools (examples)	Total	Literatures	Platform
Google Blockly (augmentation)	Accessible Blockly; Grid World; Voice-enabled Blockly; Accessible Blockly Prototype; Modified Blockly; Pseudospatial Blocks (PB)	6	PS1, PS2, PS7, PS9, PS11, PS13, PS19, PS20, PS21	Cloud/ Browser
Pocket Code	Pocket Code	1	PS11	Mobile
Quorum / OpenGL–WebGL	Quorum Blocks	1	PS5	Cloud/ Browser (WebGL)
CodeMirror	CodeMirror-Blocks (CMB)	1	PS16	Cloud/ Browser
OctoStudio	OctoStudio	1	PS4	Mobile (Android/ iOS)
Blocks4All	Blocks4All	1	PS17	Mobile (iOS)
Scratch	Myna (Overlay via Java)	1	PS22	Desktop

4.1.3 RQ1.3: which technologies and platforms are employed in the development of accessible block-based programming environments?

The development of accessible block-based programming environments (BBPEs) relies on a variety of underlying technologies and platforms, which strongly influence both the technical feasibility and the accessibility affordances of the resulting systems. Two dominant categories can be observed in the studies: (1) web-based environments and (2) native mobile applications, as shown in Table 4. However, early work also explored desktop overlay approaches that retrofit accessibility onto existing environments, as in Myna's vocal interface mapped to Scratch [41].

Out of the 12 accessible block-based programming environments identified in the literature, 66% of accessible BBPEs (n=8) are *web-based*. The most common strategy is to extend the Google Blockly codebase, an open-source JavaScript library built with CSS and SVG. This approach is particularly attractive because many mainstream BBPEs, such as Scratch [4], MakeCode [6], and App Inventor [7], are now built using the Blockly library. Modified derivative of Blockly includes Accessible Blockly [23, 48], Grid World [24], Voice-enabled Blockly [14, 25], Accessible Blockly Prototype [46], Modified Blockly [40], and Pseudospatial Blocks (PB) [39]. These solutions exploit the flexibility of the Blockly framework to integrate features such as keyboard navigation, ARIA-based semantic labeling, speech synthesis, and voice input, all delivered through the browser.

While adapting Blockly is a common approach, researchers have also introduced custom web frameworks designed from the ground up with accessibility in mind [21, 43]. Quorum Blocks [21] implements a novel OpenGL/WebGL-based rendering pipeline that avoids performance bottlenecks for screen readers. CodeMirror-Blocks (CMB) [43], on the other hand, extends the widely used CodeMirror text editor by mapping abstract syntax trees to accessible DOM elements annotated with ARIA attributes. Both examples illustrate the potential of custom rendering pipelines and structural navigation techniques for advancing accessible programming experiences on the web.

Alongside web-based tools, native mobile applications represent another critical platform for accessible BBPEs. These apps are designed to run directly on mobile operating systems, such as iOS and Android, and to integrate with built-in assistive technologies. Blocks4All [26] runs on iOS and takes advantage of Apple's VoiceOver screen reader and gesture-based interaction model to provide non-visual programming on tablets. OctoStudio [27] was designed for both iOS and Android phones and tablets. This app leverages mobile device sensors (e.g., tilt, shake, tap) as accessible input mechanisms while supporting compatibility with VoiceOver and TalkBack. Finally, Pocket Code [22] adapts Scratch-like functionality to mobile devices, relying on intuitive gesture controls and sensor integration to broaden accessibility.

4.1.4 RQ1.4: what input modalities and interaction techniques have been studied or proposed in the literature?

The inherent visual and mouse-centric design of traditional block-based programming environments (BBPEs) has historically created significant barriers for learners with disabilities [26, 39, 42, 44, 45]. The literature reveals a diverse range of alternative input modalities and interaction techniques tailored to the specific needs of users with visual, motor, and cognitive impairments. These modalities range from foundational keyboard and screen reader support to innovative uses of voice commands, touch-based gestures, tangible objects, and device sensors (see Table 5).

The most widely studied input modality is the keyboard, serving as the primary substitute for mouse-based interaction for learners with visual and certain motor impairments [23, 26, 42]. Twelve of the reviewed studies explicitly incorporate keyboard navigation. Effective use of the keyboard is often paired with screen reader support [23, 43, 46], enabling auditory output for non-visual navigation and editing. While many block-based solutions rely on mainstream screen readers such as NVDA, JAWS, or VoiceOver [21, 26, 27, 40, 48], some studies, such as Grid-world [24] and Das et al. [46], adopted self-voicing interfaces using synthesized speech to reduce dependence on prior screen reader expertise.

Table 5 Alternative input and interaction methods referenced in the literature

Interaction method	Literatures
Keyboard	PS1, PS2, PS3, PS5, PS7, PS12, PS13, PS16, PS17, PS19, PS20, PS21
Screen Reader	PS1, PS4, PS5, PS7, PS16, PS17, PS19, PS20
Synthesized speech	PS2, PS13, PS21
Audio cues	PS2, PS4, PS13, PS20, PS19
Touch-based	PS4, PS6, PS11, PS12, PS14, PS17
Braille display	PS5, PS8, PS14
Speech/Voice command	PS9, PS10, PS22
Mouse and Keyboard	PS11, PS18
Magnification	PS14, PS15
Sensors	PS4

For learners with upper-limb motor impairments (ULMI), speech-based control has been investigated as a primary modality. Myna, a standalone desktop overlay, was first introduced as a vocal user interface mapped to Scratch that enabled programming by voice through structured commands and numbered block references [41]. Later voice-enabled environments [14, 25] integrate speech recognition APIs to support commands for navigation, connection, and editing. These block-based systems often employ structured command sets or distinct modes (e.g., navigation, connect, edit) to minimize ambiguity and improve recognition accuracy.

Given the growing use of mobile devices in education, several studies investigated touch-based interactions. For instance, Blocks4All [26] demonstrated that VoiceOver (iOS) and TalkBack (Android) can support non-visual drag-and-drop block placement, making touch interactions viable for learners with visual impairments [46]. Touch input is also noted as beneficial for some learners with cognitive disabilities, given its more direct and intuitive nature [22, 47].

Several studies expand interaction beyond traditional on-screen methods to improve accessibility of virtual block-based environments. Refreshable Braille displays provide tactile output that complements keyboard navigation, enabling learners to read code and workspace messages through touch [13, 21, 44]. When purely on-screen environments pose accessibility barriers, tangible and hybrid approaches also appear as compensatory workarounds, such as TVIs incorporating unplugged or physical programming activities to bridge the gap [10]. OctoStudio integrates device sensors as an alternative input modality in mobile-based systems, allowing on-screen scripts to be triggered by physical gestures such as shaking or tilting the device [27]. For learners with low vision, magnification software remains a common accessibility strategy [13, 45].

4.2 RQ2: what accessibility challenges do learners with disabilities face that are directly attributable to the structure and functionality of block-based programming environments/tools?

We conducted a structural analysis of all primary studies following the process described in Sect. 3.3. Through this analysis, we extracted the accessibility challenges and barriers that learners with disabilities encounter when using virtual block-based programming platforms. Our investigation

revealed numerous issues, many of which recurred across multiple studies. We consolidated these recurring issues under generic category names to avoid redundancy and improve analytical clarity.

To systematically organize our findings, we further categorized the identified issues into six primary accessibility barriers in block-based programming environments. This categorization provides a high-level framework for understanding the scope and nature of accessibility challenges. Table 6 presents all curated issues organized within their respective categories.

4.2.1 Interaction and input barriers

Block-based programming environments (BBPEs) are inherently visual in nature and rely on drag-and-drop interfaces to access programming features. These platforms depend heavily on mouse-centric input for user interactions, which poses significant accessibility barriers for learners with visual and motor impairments [24, 27, 42, 44, 46, 50]. Many learners with these disabilities are unable to perform the drag-and-drop mouse actions required to assemble blocks within block editors [39, 45]. Due to these barriers, they must rely on alternative input methods, such as keyboards. However, popular BBPEs, such as Blockly and Scratch, remain incompatible with keyboard-only input for program navigation and editing [23, 48].

Students with motor impairments, including those with cerebral palsy, muscular dystrophy, or multiple sclerosis, face difficulties performing tasks such as typing on a keyboard or dragging and dropping virtual blocks with a mouse, which can be extremely challenging [14, 25, 41]. Motor impairments are also prevalent among students with autism spectrum disorder (ASD), who often struggle to coordinate mouse operations and translate hand movements into corresponding on-screen actions [47]. Zubair et al. [47] found that some children with ASD struggle particularly with performing a double-click.

Students with visual disabilities sometimes experience additional dexterity issues when navigating blocks through touch gestures while using assistive technologies such as VoiceOver on iPads [13]. For students with cerebral palsy, articulating lengthy voice commands becomes problematic [25], and their speech often proves difficult for systems to recognize due to atypical accents, low voices [14], or meter of speech [41]. Students with low vision frequently encounter magnification-related issues, as the default magnification height for blocks restricts their view to only two or three lines of content at a time, making the rest of the user interface inaccessible [21]. They also find zooming tedious [26].

4.2.2 Navigation and orientation challenges

Effective navigation forms the foundation of code comprehension and programming practice. When navigation systems prove inaccessible, students with disabilities face significant barriers to developing programming skills.

Students with visual impairments encounter fundamental obstacles when navigating block-based programming interfaces. They experience difficulty moving between different sections of the programming environment [45], including inaccessible toolboxes and workspaces [26]. Mountapmbeme et al. [23] revealed that typical block navigation proves both tedious and unpredictable for users with visual impairments. They reported that simple navigation requires traversing all of a block's connection points, often leaving users uncertain about their current location when they attempt to backtrack. Also, navigation becomes particularly problematic when users accidentally trigger actions that cause them to lose their place or focus within the programming environment [24, 47]. The inherently spatial, non-linear layout of block languages also obscures program traversal order due to arbitrary positioning [21].

Navigation challenges extend to other disabilities in distinct ways. The non-linear arrangement of blocks can confuse some neurodivergent learners who may find spatial relationships harder to parse [26]. Learners with mobility impairment face additional barriers when navigating programming environments. Those who favor one-handed operation find striking simultaneous keys for shortcuts particularly difficult, and case-sensitive shortcuts compound these challenges [25]. Zubair et al. [49] found that participants, especially those with fine motor skills difficulties, struggled with dragging items within the Scratch interface. They encountered difficulties positioning their sprites during story setup and dragging blocks to new positions.

Editing programs is consistently reported as a major challenge for students with visual impairments. The difficulty is often tied to inadequate audio feedback and complex navigation models [45]. For instance, editing on an iPad with VoiceOver was found to be frustrating due to the complex rotor gestures required for basic actions, such as cut, copy, and paste [13]. Block modification presents another persistent source of frustration, as many users cannot move blocks across the workspace, and functionality was limited to connecting blocks only through insertion from the toolbox [46]. An evaluation of Scratch highlighted that the "absence of an alternative way to move blocks, such as a keyboard" was a primary reason for task failure among users with disabilities [45].

Visually impaired learners encounter significant challenges in code comprehension because standard line-based search methods prove inadequate for navigating the

Table 6 Accessibility challenges in virtual BBPEs as reported in literature

Category	Accessibility Issue	Studies	Count
Interaction and input barriers	Reliance on mouse-centric and drag-and-drop interfaces	PS1, PS2, PS3, PS4, PS7, PS8, PS13, PS15, PS19, PS21	10
	Interaction difficulties with mouse or keyboard (dexterity)	PS6, PS9, PS10, PS11, PS22	5
	Dexterity issue in touch gesture	PS14	1
	Speech input recognition challenges	PS9, PS10, PS22	3
	Magnification Issue	PS5, PS17	2
Inaccessible content and outputs	Inconsistent visual presentation	PS11	1
	Color contrast issue (text/icons/UI)	PS3, PS5	2
	Ambiguous block shape semantics and colors	PS9, PS11, PS18, PS17, PS21	5
	Dragging blocks and sprites	PS18	1
	Inaccessible program output	PS8, PS15, PS17	3
	Inaccessible material for BVI	PS8, PS12	2
Navigation and orientation challenges	Inefficient and cognitively demanding navigation	PS1, PS7	2
	Inaccessible code comment	PS13	1
	Loss of focus	PS2, PS17	2
	Unclear layout	PS5	1
	Limited search functionality	PS16	1
	Difficulties editing programs	PS12, PS13, PS14, PS15	4
	Inaccessible code navigation	PS14, PS15, PS20	3
	Inaccessible toolbox and workspace	PS17	1
	Difficulty understanding program structure	PS16, PS17	2
Cognitive barriers	Memory burden	PS1, PS2, PS7, PS9	4
	Sensory sensitivities to sound	PS6	1
	Cognitive overload (choices for sprites, backgrounds, and programming blocks)	PS6, PS8, PS11, PS18	4
	Difficulty dealing with change	PS6	1
	Struggles reading text labels	PS6	1
	Dealing with abstraction, planning, and sequencing	PS18	1
	Lack of constraints	PS18	1
	Information overload through screen reader	PS19	1
Assistive technologies and feedback barriers	Lack of screen reader support	PS1, PS4, PS5, PS8, PS15, PS19	6
	Synthesized speech conflict with screen reader	PS2	1
	Inadequate audio feedback	PS2, PS13, PS14	3
	Speech API sometimes kept reading stale elements	PS2	1
	Missing alternative text for visual graphics	PS3	1
	Screen reader navigation is serial and hierarchical	PS16, PS21	2
	Incompatible with braille	PS5	1
	Visual-only feedback	PS19	1
	Moving blocks not SR-friendly	PS17	1
Educational and technical barriers	Accessible starvation	PS5	1
	Platform technology overriding assistive tech	PS19	1
	Lack of accessible tools in cloud-based environments	PS16	1
	Mobile device compatibility	PS6	1
	Slow system response	PS9	1
	Limited basic computing skills	PS12, PS14	2
	Training video was overwhelming	PS4	1
	Steep learning curve of screen reader	PS13	1
	Teacher and curriculum barriers	PS12, PS16	2

hierarchical and spatially organized nature of block-based code [43]. This limitation forces users to struggle with forming accurate mental models and deriving overall program structure and logic flow. The fundamental problem arises because audio feedback systems lack access to a program's tree structure, forcing BVI learners to rely heavily on their short-term memory to keep track of complex programming constructs such as nested conditionals inside a loop [26, 43]. Inconsistent audio feedback can make navigation more challenging, creating confusion [48]. Moreover, some essential features for code comprehension in block-based environments, such as comments or tooltips that provide more information about a block, are not always accessible via the keyboard [21, 46].

4.2.3 Inaccessible content and outputs

Inconsistent visual presentation represents a fundamental accessibility challenge across popular BBPEs. Sharfuddeen et al. [22] identified significant inconsistencies in the visual presentation of information within *Code.org*, Kodu, and Pocket Code that create difficulties for users accessing and creating content. Low color contrast between text, icons, and backgrounds compounds these problems, making it particularly difficult for users with low vision to perceive and interact with interface elements. Kamil et al. [50] performed a systematic accessibility evaluation to reveal the scope of these issues. In their study, StarLogo showed the highest number of accessibility violations in the WCAG perceivable category, with problems concentrated in missing form labels, very low contrast ratios, and unlabeled selection elements. Students also faced issues with low contrast in dark mode [21].

Block-based environments often rely on visual cues, such as shape and color, to convey a block's function, type, or how it can connect to other blocks. This information is inaccessible to users who are blind and can be confusing if colors are too similar or shapes are not distinct enough. The colors of the blocks in different menu categories confuse learners with cerebral palsy [25]. Similar challenges were observed in *Code.org* and Pocket Code for learners with learning disabilities [22], and children with cognitive impairments struggled to identify categories even when colors were used [49].

Many introductory programming environments emphasize visual outputs, such as animations, games, or moving avatars, which are not perceivable by learners with BVI [45]. Although Scratch, ScratchJr, Hopscotch, and Tynker provide some audio output options, the majority of games and projects focus on visual output and thus remain inaccessible [26]. Due to the visual nature of learning materials and tools, BVI students are often assigned alternative activities that may not cover equivalent concepts. Many materials in block-based environments are perceived as inaccessible for these learners [10, 44].

4.2.4 Cognitive barriers

Issues related to memory burden arise when users must rely heavily on short-term memory to navigate or understand a programming environment. In the Accessible Blockly study, a participant noted that remembering connection points during navigation increased cognitive load [23]. Learners with visual impairments also struggled to recall numerous shortcut keys for different actions [24, 48]. Similarly, users face a learning curve associated with memorizing the required verbal commands to operate the interface in voice-enabled systems [25].

Children with Autism Spectrum Condition (ASC) often experience heightened sensory sensitivities, particularly to sounds [47]. The abundance of options in BBPEs, such as sprites, backgrounds, sounds, and blocks, combined with open-ended scenarios like blank projects, can overwhelm these learners, leading to distraction, repetitive behaviors, or decision-making difficulties [22, 44]. Confusion often arises when navigating different interface areas or recognizing buttons and links [49]. Additionally, unexpected or poorly managed interface changes can be distressing, as some children with ASC strongly prefer consistency [49]. Interfaces that rely heavily on text for labels, buttons, and blocks pose additional challenges for those with reading or comprehension difficulties [49].

Learners with cognitive impairments often struggle with abstraction, planning, and sequencing. A study with cognitively impaired learners reveals that all participants struggled with structuring and sequencing the events in their stories, breaking down the story idea into components in Scratch [49]. These learners also struggle to stay focused, often exhibiting distraction or non-goal-oriented behavior due to insufficient constraints. For blind or visually impaired (BVI) learners, screen readers can cause information overload by presenting excessive information, such as reading an entire nested code block as a single string, overwhelming short-term memory [42]

4.2.5 Assistive technologies and feedback barriers

A primary and frequently mentioned barrier is the incompatibility of most BBPEs with screen readers [42, 45]. Since individuals with visual impairments rely on screen readers and keyboards to access digital content, this lack of support renders many platforms completely inaccessible [27, 48]. Visual block languages are generally not screen reader accessible [21]. Evaluations report that users cannot focus

on blocks in the toolbox or workspace, and that block labels and symbols are not read correctly [26, 45]. Mainstream environments such as Lego Mindstorms and Scratch are found to be incompatible with screen readers [44].

Some studies have explored synthesized APIs as alternatives to screen readers for audio feedback [24, 39, 46]. However, this approach can conflict with a user's existing screen reader or deliver stale messages when speech output fails to update [24]. Many BBPEs provide limited or no feedback on user actions, such as navigation, movement, or block manipulation. Teachers reported in a study [13] that BBPEs offer limited feedback through screen readers when students try to navigate, locate, move, or manipulate blocks on the workspace. For example, in a study [46], a participant attempted to create a variable but received no audio cue that the system was ready for input. Screen readers can also struggle to provide precise information within a BBPE, failing to meet the needs of a visually impaired student during programming tasks [24]. Evaluations of mainstream BBPEs, such as Scratch and App Inventor, confirm that screen readers cannot adequately describe blocks for blind users [50].

A key limitation of screen readers is that their navigation is serial and hierarchical, which fails to communicate the nested, two-dimensional structure of block-based code [39, 43]. Many BBPEs rely exclusively on visual cues to provide feedback (e.g., a visual change on the workspace, pop-up messages, dialog boxes) that are inaccessible to those without vision [42]. In one evaluation of nine environments, moving blocks were not screen-reader friendly, leaving users unaware of a block's current location or its relation to a target [26]. Additionally, BBPEs are often incompatible with refreshable Braille displays, another essential tool for blind users [21].

Users with low vision often face barriers accessing visual elements in block-based programming platforms [23, 24, 26]. While magnification is available at the operating system level, low vision users find zoom features tedious because zooming in obscures the rest of the workspace, removing vital contextual information required to understand code structure [26, 46]. Magnification can also make it difficult to keep track of the mouse position across the interface, which hinders drag-and-drop block operations [26]. Furthermore, the visual cues used to convey data types, such as the shape and color of a block, are often inaccessible to low vision users due to their small size [21, 26]. While Blockly-based BBPEs offer built-in zoom in and out features using the mouse, they are not accessible through keyboard-only interaction [42, 46].

4.2.6 Educational and technical barriers

The literature highlights several technical barriers to assistive technologies in block-based programming environments (BBPEs). For instance, Quorum Blocks utilizes OpenGL graphics and communicates with assistive tools via operating system calls, resulting in delays of up to 10 s before screen readers respond to each keystroke [21]. Similar response-time delays have also been reported in voice-enabled systems [25]. Older environments built with technologies such as Flash can override essential keyboard shortcuts for assistive tools [42]. Moreover, the shift towards browser-based educational tools has created an accessibility gap, as many existing solutions are designed for desktop applications and are not language-independent [43]. Many BBPEs lack mobile compatibility, which limits intuitive touch-based interaction and creates additional barriers for learners with motor difficulties [47].

Teachers report that many students begin coding without foundational skills such as keyboarding or proficiency with screen readers [10, 13]. Screen-reader proficiency is not universal among K-12 students with visual impairments due to cost, recent vision loss, or the technology's steep learning curve [46]. Even accessible training materials can overwhelm users if they deliver information too rapidly, such as videos with audio descriptions that switch quickly between examples [27]. The educational ecosystem itself–including teacher expertise, curriculum design, and available materials can create significant barriers for students with visual impairments. Studies reveal that lesson plans are often inaccessible, either being too limited or overly complex for students with visual impairments, highlighting systemic challenges with accommodations and the availability of suitable study materials [10, 13].

4.3 RQ3: what are the current proposed solutions to reported barriers?

To mitigate the accessibility challenges and barriers in the virtual block-based programming environment, as discussed in Sect. 4.2, researchers have studied various solutions targeting learners with different disabilities. These solutions aim to mitigate the barriers faced by users with visual, motor, and cognitive impairments by providing alternative interaction modalities, non-visual feedback mechanisms, and cognitive supports. The following sections categorize and discuss these current solutions in detail. Table 7 presents all the solutions reported in the literature, and Table 8 organizes the mapping of barriers to the reported solutions.

Table 7 Solutions to accessibility barriers studied in the literature

Category	Solution	Studies
Alternative Input and interaction	Keyboard-based program navigation and editing	PS1, PS2, PS7, PS19
	Accessible drag-and-drop-like operations via keystrokes	PS1, PS2, PS4
	Custom keyboard shortcuts	PS1, PS2, PS7, PS13
	Enable/disable keyboard accessibility	PS1, PS2, PS7
	Voice-command navigation	PS9, PS10, PS22
	Audio-guided drag and drop	PS17
Auditory feedback and cues	Screen reader support	PS1, PS7, PS17, PS20, PS4, PS5, PS19, PS16
	Speech/TTS feedback	PS2, PS13, PS21
	Non-speech audio cues or sonification	PS2, PS13, PS17, PS20, PS4
	Spoken description of blocks	PS1, PS7, PS16
Structural and conceptual navigation	Structural navigation	PS16
	Spatial program navigation	PS1, PS2, PS7
	Pseudospatial navigation	PS21
	Ancestor readout shortcut for orientation	PS16
	Collapsing nodes	PS16
	Location-first select, drop	PS1, PS17
	Alphanumeric block labeling	PS19, PS22
	Multiple mode for navigation and editing	PS1, PS7
Interface design and customization	Distinct color coding	PS11
	Linear block layout	PS5
	Freeform blocks	PS5
	Automatic block alignment	PS4
	Embedded typed blocks	PS17
Accessible output & debugging	Step by step debugger	PS2
	Accessible graphics pipeline (2D/3D canvas)	PS5
	Accessible program output (alt text / ARIA / live regions)	PS2
Cognitive and learning support	Template support / scaffolding	PS11
	Filter block selection	PS21
	Dual-syntax functionality	PS16

4.3.1 Alternative input and interaction

Traditional BBPEs rely heavily on mouse-centric drag-and-drop interactions, creating significant barriers for many users. To address this limitation, researchers have proposed several alternative input methods. Ludi et al. [42] first proposed keyboard shortcuts as an alternative input method for users with visual impairments. Subsequently, multiple studies have developed keyboard-oriented interaction methods for program navigation [23, 24, 48], primarily through modifications to Google Blockly. These implementations typically feature a virtual cursor on the workspace, controlled by keys such as W (up), A (left/back), S (right/next), and D (down). The WASD key combinations were selected to avoid conflicts with standard screen reader hotkeys [23]. However, Koushik et al. [39] suggested using arrow keys for cursor control. These solutions also provide custom shortcuts for controlling various features in browser-based programming environments [23, 24, 46, 48]. Additionally, users are given options to enable or disable keyboard accessibility features [23, 24, 48].

To support users with upper-limb motor impairments (ULMI) who find extensive keyboard or mouse use difficult, voice-enabled systems have been developed [25, 41]. These voice-based block systems use a predefined set of voice commands to navigate, select, edit, and delete blocks [14, 25]. For touchscreens, Milne et al. [26] explored different interaction models. An "audio-guided drag and drop" method, where the system provides audio feedback on the block's location as it is being dragged, was tested but found to be unusable by children who rely on the VoiceOver screen reader. In contrast, a "select, select, drop" method, where a user first selects a block and then a location, was successfully used by all participants in a formative study [26]. Similarly, the OctoStudio app [27] implements an accessible drag-and-drop feature where the screen reader announces valid drop locations as users drag blocks, providing a fluid, non-visual method for program assembly [27].

Table 8 Mapping of accessibility barriers to solution categories

Barriers/challenges category	Reported solutions categories
Interaction and input barriers	Alternative input and interaction, auditory feedback and cues, structural and conceptual navigation, interface design and customization
Inaccessible content and outputs	Auditory feedback and cues, interface design and customization, accessible output & debugging
Navigation and orientation challenges	Alternative input and interaction, auditory feedback and cues, structural and conceptual navigation, cognitive and learning support
Cognitive barriers	Auditory feedback and cues, interface design and customization, cognitive and learning support
Assistive technologies and feedback barriers	Auditory feedback and cues, structural and conceptual navigation, interface design and customization, accessible output & debugging
Educational and technical barriers	Alternative input and interaction, interface design and customization, accessible output & debugging, cognitive and learning support

4.3.2 Structural and conceptual navigation

A significant challenge for users with visual impairments is constructing a mental model of a program's hierarchical structure [43]. To address this, several navigation paradigms have been developed. Aboubakar et al. [23, 48] introduced Spatial Navigation, which emulates the 2D visual layout of blocks. Using directional keys (WASD), users navigate as if moving across a grid. A similar strategy was adopted by Tabassum et al. [24].

In contrast, Schanzer et al. [43] proposed a Structural Navigation approach based on an Abstract Syntax Tree (AST), representing the logical structure of code rather than its visual layout. While promising, this method has not been applied to conventional block-based environments, and the authors acknowledged that it requires further evaluation. Koushik et al. [39] took a different approach named "Pseudospatial Navigation", which employs a spatial metaphor but intentionally distorts movement geometry for improved efficiency. For instance, moving "left" from any workspace block consistently navigates users to the same logical position in the toolbox. However, the effectiveness of this method has not been evaluated through user studies.

Beyond program structure navigation, several additional features have been identified in the literature. Alphanumeric Block Labeling [42] assigns unique hierarchical labels (e.g., A, B, B1, B1.1) to blocks, helping users understand block position and nesting. Similarly, Wagner et al. [41] used numbered block references to support structured spoken commands for selecting and manipulating blocks in Scratch. To address the challenge of navigating large programs, Schanzer et al. [43] implemented Collapsible Nodes, which allow users to expand or collapse container blocks (e.g., loops or functions), hiding their inner content to facilitate quick skimming. They also introduced an Ancestor Readout Shortcut, which verbally announces the parent hierarchy of the currently focused block, aiding orientation within nested structures. However, the latter two features were tested in hybrid text-and-block environments, which may limit their applicability to traditional block-based systems.

During navigation, one of the most challenging tasks is program editing, which involves various operations for modifying blocks. A common editing task is attaching blocks in the workspace, traditionally performed via mouse-based drag-and-drop. A "location-first select, drop" approach has been adopted in multiple studies to enable this operation using the keyboard [23, 26, 48]. Systems such as Accessible Blockly employ distinct Navigate and Edit modes to separate the tasks of exploring, supporting screen readers on both iOS (VoiceOver) and Android (TalkBack) devices, using negative load [23, 48].

4.3.3 Auditory feedback and cues

Visually impaired users rely heavily on assistive technology for feedback in programming environments, as they may have limited to no vision. As a solution, several studies have explored integrating screen reader support to make block-based programming environments compatible with assistive technologies such as NVDA, JAWS, and VoiceOver. AccessibleBlockly was among the earliest prototypes to implement screen reader compatibility, using ARIA templates that update at runtime to reflect workspace changes [21, 23, 40, 48]. CodeMirrorBlocks [43] adopted a different approach by rendering the program's structure as a DOM tree with extensive ARIA attributes (e.g., aria-label, aria-level) specifically designed to make programs navigable and understandable by screen readers. For mobile platforms, the Blocks4All [26] prototype was developed as an iPad application specifically designed for use with the built-in VoiceOver screen reader. Similarly, OctoStudio [27] incorporates navigational features supporting screen re-aders on both iOS (VoiceOver) and Android (TalkBack) devices, utilizing ARIA properties to make custom elements accessible to blind and visually impaired learners.

Some systems employ self-voicing solutions with built-in synthetic speech engines directly into the environment to reduce reliance on external screen readers. For example, GridWorld [24] utilized the JavaScript Speech Synthesis API to deliver context-specific audio feedback, such as describing a block's position or providing debugging information. Likewise, Das et al. [46] developed custom Text-to-Speech (TTS) feedback as a cost-effective alternative to commercial screen readers, such as JAWS. The Pseudospatial Blocks (PB) system [39] also used synthetic speech to describe program structure during keyboard-based block creation. However, evaluations revealed that these self-voicing approaches often respond slowly and can conflict with system-default screen readers, creating challenges for users.

To enhance comprehension, some solutions provided "Spoken Descriptions of Blocks" that provide semantic descriptions of what a block does, rather than just reading its literal syntax. Accessible Blockly [48] prepends contextual information, such as announcing "container block" before reading the block's content, to help users build a better mental model. For example, if the user focuses on the "repeat block", the system prompt screen reader "container block, repeats three times". Similarly, the CodeMirror-Blocks tool [43] developed a parser to generate a spoken label for ASTNode during HTML DOM rendering using the aria-label attribute. The description can differ from syntax like "foo: a function definition that is public, static, and produces a double".

Non-speech audio cues, such as earcons (abstract tones) and spearcons (speeded-up speech), as well as other sound effects, have also been explored to convey information about program structure, events, or block types. Ludi et al. [40] proposed earcons and spearcons to indicate block types, nesting levels, or actions, with spearcons proving more effective. GridWorld [24] employs a variety of tones and sounds to confirm actions, indicate grid element types (e.g., a "splash" for water), or signal execution progress. A unique technique found is Binaural Spatialization, which maps sounds to the left or right ear to signify the opening or closing of nested code blocks. Das et al. [46] implemented this in their modified Blockly, using distinct click sounds in the left and right earphones to denote the opening and closing of nested blocks. Another technique, sonification, maps sounds directly to a program's visual output. For instance, in OctoStudio, executing a "jump" block triggers a "boing" sound, providing immediate non-visual feedback on program actions [27].

4.3.4 Interface design and customization

Several studies emphasize the importance of functional layout and visual adaptability in block-based programming environments to support users with low vision and certain cognitive impairments [22, 26, 27]. To ensure predictable navigation for both keyboard and screen reader users, some systems constrain block placement to a linear, top-to-bottom layout rather than a freeform canvas. For example, the OctoStudio app automatically aligns and snaps blocks into a vertical sequence when a screen reader is active, reducing spatial ambiguity and improving discoverability [27]. Visual differentiation was further enhanced through high-contrast color schemes for block categories, aiding users with low vision in distinguishing between block types [22]. To support users who rely on screen magnification, one system introduced a "thin" mode that reduces the vertical padding of blocks, allowing more code to be visible on the screen at once [21].

In addition to layout adaptations, systems also incorporate features to provide greater input flexibility. Some environments introduced freeform blocks, which allow users to temporarily switch from block manipulation to direct text input for entering expressions [21]. Other designs employed embedded typed blocks, where options within a block (e.g., the number of repetitions in a loop) can be cycled through using swipe gestures or keyboard commands [26].

4.3.5 Accessible output & debugging

A critical yet often neglected aspect of accessibility in block-based programming environments (BBPEs) is the program's output, which is typically visual and inaccessible to users with visual impairments. While tangible outputs have proven effective in some contexts, they are not feasible in virtual BBPEs. To address this, GridWorld [24] introduced an approach to make program output accessible for maze puzzle programs on a 2D grid. Users can navigate each cell to hear its state via speech or sound cues, marking an initial step toward accessible outputs. While promising, the study reported that some students encountered technical issues with the output system, which limited its reliability. Beyond auditory feedback, some systems aim to make the graphical output accessible in its own right. The Quorum Blocks environment [21] introduced a custom, hardware-accelerated graphics pipeline to create an accessible 2D/3D canvas. This allowed users to explore on-screen objects, along with their properties and coordinates, non-visually with a screen reader or braille display. A noted limitation of this approach was the capped frame rate imposed by the accessibility layer.

Debugging is a critical aspect of programming that helps users analyze and resolve errors when programs behave unexpectedly. Tabassum et al. [24] were the first to introduce a step-by-step debugger in the context of accessible block-based environments. Their approach allowed users to execute programs one block at a time, with each executed block and the corresponding output changes announced aloud to support understanding of animation effects within the grid. However, the study provided limited details on how this debugging mechanism works and how it helps users identify and fix errors in practice.

4.3.6 Cognitive and learning support

To accommodate diverse cognitive needs, the CodeMirror-Blocks tool [43] enables seamless switching between a block-based and a traditional text-based code view, offering "syntax when you need it, structure when you don't". This dual-mode interface supports users with cognitive disabilities, such as Autism Spectrum Condition (ASC) or learning disabilities, who may find graphical components confusing or overwhelming. By providing flexibility in how code is visualized, the tool caters to varied user preferences and task requirements.

For users who struggle with decision overload, particularly those with ASC, systems can implement constraints to simplify interactions. One effective technique proposed by Koushik et al. [39] involves filtering the block selection to display only syntactically valid blocks for the current insertion point, thereby reducing complexity and guiding users toward correct choices. Additionally, users with ASC often face challenges in planning and structuring projects from scratch. Some studies recommended providing templates

or sample projects as starting points [39]. These scaffolding mechanisms help mitigate cognitive overload and foster confidence in novice programmers. However, many of these proposals remain at the conceptual stage and require implementation and empirical evaluation to validate their effectiveness [47, 49].

5 Implications

Our analysis of the research questions (RQs) reveals several implications for advancing accessibility in block-based programming environments (BBPEs).

5.1 Implications for researchers and practitioners

Based on our findings, we identify key implications for researchers and practitioners working to improve accessibility in block-based programming environments. These address critical areas, including participant diversity, evaluation scope, audio feedback, assistive technologies, accessibility standards, and interoperability, as well as auditory cue design. These insights aim to inform future research directions, design practices, and collaborative efforts in building more accessible and inclusive block-based programming environments.

Accessibility Standards and Interoperability for BBPEs: There is no universally adopted, prescriptive accessibility standard for block-based programming environments (BBPEs). Researchers currently rely on general guidelines [27, 42, 44, 50], such as Web Content Accessibility Guidelines (WCAG) [51] and Accessible Rich Internet Applications (ARIA) [52]. However, these standards are not always sufficient as WCAG is not prescriptive and does not specify how to ensure accessibility for novel components such as block-based controls and canvases [21]. Furthermore, ad-hoc design choices, ranging from keyboard shortcuts to voice commands, combined with architectural divergence, result in limited interoperability and hinder adoption [24, 25, 48]. To address these challenges, accessibility researchers and developers should collaborate to propose BBPE-specific accessibility standards and interoperability specifications that define consistent controls, commands, assistive technology behaviors, and Canvas and DOM models.

Auditory Cues Enhancement: A significant limitation of existing research is that studies on auditory cues are often conducted with small-scale programs and focus narrowly on navigation and comprehension tasks [40, 43]. This leaves major gaps in understanding how auditory cues can support the broader programming lifecycle [13, 42]. Future research should investigate how these cues can effectively aid more complex and dynamic tasks such as program creation, editing, and debugging, particularly within larger codebases. Further exploration is also needed into the learnability and user preference of different auditory representations (e.g., earcons vs. spearcons). To guide this exploration, approaches used in text-based and audio-centric programming systems such as SODBeans [53], Sonic Pi [54], and the Audio Programming Language (APL) [55] may offer useful design directions for incorporating auditory debugging support, using non-speech audio cues to convey the data types and syntactic compatibility typically represented by block shapes and color, and providing audio-based representations to communicate nesting levels and hierarchical program structures to support code understanding for visually impaired users. These design directions could help future BBPEs address pedagogical concerns by reducing cognitive load and supporting more accurate mental models of code structure and execution [43, 48].

Narrow and Preliminary Evaluation Scope: Many studies are explicitly described as preliminary, formative, or exploratory, with a scope often limited to a subset of the full programming workflow [13, 14, 21, 42]. For instance, several evaluations focus strictly on code navigation and comprehension, deliberately excluding the more complex tasks of code creation and editing [23, 43, 48]. The focus is often on assessing the feasibility of a specific interaction mechanism rather than the tool's effectiveness in a realistic, long-term educational context [13]. This leaves a significant gap in understanding the usability of these tools for the entire iterative cycle of programming, including creation, editing, and debugging.

Limitations of Audio Feedback and Screen Readers: While audio is a key channel for non-visual information, its implementation presents challenges. Custom synthetic speech ("self-voicing") can conflict with system screen readers and braille displays [24, 46]. Studies have often noted that screen reader output can be excessively verbose, leading to high cognitive load. Participants frequently request adjustable verbosity and speech rate, task-aware summaries, and concise phrasing [21, 24, 48]. Future work should explore adaptive audio that is sensitive to task, expertise, and user preferences while avoiding conflicts with system-assistive technologies.

Most Disabilities Remain Understudied: A recurring limitation across the literature is reliance on small and often homogeneous participant groups. Many studies report single-digit samples [22, 24, 25, 46]. Moreover, most participants are learners with visual impairments (VI) [26, 43, 45, 48], with fewer studies involving motor impairments [14, 25] and even fewer involving cognitive impairments or Autism Spectrum Disorder (ASD) [22, 49]. While recruitment in these populations is challenging, the research remains

heavily skewed toward addressing the needs of learners with visual impairments [24, 39, 48]. As a result, many other disabilities remain understudied [21] such as learners with cognitive [21], motor impairments [14], or those with Autism Spectrum Disorder (ASD) [49]. Expanding the focus to include students with multiple or co-occurring disabilities would enhance the inclusivity of block-based environments. Additionally, future work should aim for more diverse and adequately powered samples through sustained partnerships and recruitment strategies adapt-ed from broader accessibility research.

5.2 Implications for developers

Literature presents both opportunities and challenges for developers building accessible block-based environments. The following implications are particularly relevant to developers, highlighting platform and design constraints, implementation practices, underexplored areas such as mobile and touch-based accessibility, and challenges in code editing and debugging. We believe these will provide practical guidance for building more inclusive and interoperable block-based systems.

Inaccessible Code Debugging and Comprehension: The ability to comprehend code and identify and fix errors is a cornerstone of programming, yet accessible debugging tools are underrepresented in the literature. While "no-syntax-error" approaches (e.g., Scratch) reduce syntactic issues, they do not address logical errors [49]. While a couple of literatures mention debugging support as a feature [24, 43], it is largely absent from many discussions and demand evaluation [13, 39]. Accessible, step-wise tracing, state inspection, and execution summaries are needed to support learners using screen readers, braille, or audio-first workflows.

Persistent Barriers to Code Editing and Manipulation: Creating and editing code present significant hurdles. The drag-and-drop interaction model is a primary accessibility barrier for users who rely on keyboards or have motor impairments [25, 39, 44]. While alternatives such as "select, select, drop" have been explored, users and teachers still report that editing existing code is one of the most challenging tasks, particularly on touchscreen devices using complex gestures [10, 13, 26, 48]. Studies with children with cognitive impairments also found that they struggled significantly with dragging and rearranging blocks within a script [49]. This highlights the need for intuitive and efficient non-visual methods for inserting, deleting, and reordering blocks, as well as parameter editing, that minimize cognitive and motor load.

Accessibility of Program Outputs and Core Components: A core challenge in visual programming is that both the program's output and environment components are heavily visual [39, 44]. The output is primarily in the form of animations, games, or visual avatars, which are often inaccessible to learners with visual impairments [42, 45, 48]. A significant portion of the core components remains difficult to access via alternative input methods, and issues with color, shape, semantics, iconography, and control placement exacerbate these barriers [22, 26, 39, 49]. Future work should strengthen alternative outputs and ensure core components (block properties, palettes, inspectors, manuals/tutorials) are perceivable, operable, and understandable across modalities.

Ad-hoc Design and Platform Choices Constraint: The underlying technology stack fundamentally shapes the accessibility of BBPEs. For example, older versions of Scratch built with ActionScript/Flash were considered inaccessible because they lacked support for necessary specifications, such as ARIA roles and states [42]. In contrast, environments built with standard web technologies (CSS, SVG, JavaScript), such as Blockly, have the potential for accessibility if developers properly adhere to standards [42, 46]. Most research prototypes are implemented by forking the Google Blockly codebase, which creates multiple silos and makes adoption difficult [23, 24, 40]. Moreover, these solutions are not release-ready or in a shippable form that can be integrated into mainstream BBPEs for effective evaluation. Future solutions could follow a modular systems design to enable easier integration and assessment.

Sparse Evidence on Mobile and Touch-Based Tools: While mobile and touchscreen devices are widespread, most research targets desktop web environments (Table 4). Notable exceptions include Blocks4All [26] and OctoStudio [27], which are iPad/iOS and mobile apps designed to work with built-in screen readers. However, broader coverage across mobile platforms and touch interactions remains limited [13]. The majority of other accessible environments studied are primarily designed and evaluated as web-based applications for desktop computers. This paves an avenue for expanding inclusive access, especially for schools with tablet-first ecosystems.

5.3 Implications for educators

Educators play a crucial role in integrating block-based programming solutions into the classroom. However, the literature reveals that limited foundational computing skills among learners, a lack of accessible instructional materials, and insufficient systemic support often make effective classroom instruction challenging. The following implications can guide educators in addressing these challenges.

Lack of Foundational Computing Skill: Multiple studies report that learners often lack prerequisite "precursor

skills", including proficient keyboarding and the effective use of assistive technologies such as screen readers [10, 13]. These limitations impede productive engagement with BBPEs and can confound study results [27, 46]. Conducting brief, targeted training and workshops on foundational skills before primary tasks can improve both learner outcomes and study quality.

Lack of Accessible Curricula Creates a Systemic Barrier: Even when an accessible programming tool is available, its effectiveness is severely hampered by the lack of accessible curricula and lesson plans [10, 13]. Teachers consistently report that existing educational materials are inaccessible and ill-suited to students' needs [10, 39]. Only a few studies evaluate classroom lessons or K-12 activities within proposed BBPEs. We recommend coupling tool evaluations with accessible curricula, teacher supports, and classroom-ready materials to enable authentic adoption.

6 Limitations

We employed a systematic literature review (SLR) methodology to derive our findings on the current challenges and solutions related to the accessibility of block-based programming environments for learners with diverse disabilities. As with any systematic literature review, our approach has inherent limitations. Because our review primarily focuses on the accessibility of purely virtual block-based programming environments, the scope decisions we made shape what evidence is represented. Below, we outline the key limitations and the steps we took to mitigate them.

Search Scope and Selection: Defining the search scope and selecting appropriate keywords were initial challenges in conducting our review. To overcome this, we conducted a series of pilot searches with a foundational set of search terms. We then developed a set of commonly used and domain-specific terms relevant to the accessibility of block-based programming. We selected six digital libraries to broaden our search; however, some publications may have been excluded. To mitigate this limitation, we employed a snowballing technique to refine and extend our search and selection process.

Terminology Consistency: As this field is comparatively new, we observed considerable variability in terminology across studies, which posed a limitation. Different studies often used distinct terms for similar constructs (e.g., "auditory cues" vs. "sonification", "self-voicing" vs. "synthetic speech", "program manipulation" vs. "code editing") when describing accessibility challenges, solutions, assistive technologies, or programming concepts. This variability may have hindered our ability to capture all relevant data during analysis. To mitigate this, we maintained an organized and shared list of coding terminology among authors, which was consistently followed throughout the review process.

Objectivity, Coding, and Reviewer Bias: Systematic literature reviews are susceptible to limitations in objectivity, coding consistency, and reviewer bias during the screening, data extraction, and synthesis stages. To minimize reviewer bias and ensure coding consistency, each study was independently reviewed and coded by multiple authors. Any disagreements or conflicts in findings and coding schemes were resolved through consensus meetings. Nonetheless, thematic interpretation may still reflect reviewer perspectives. To further ensure objectivity, we categorized and reviewed papers in multiple steps following a consensus process. This allowed us to capture diverse perspectives and interpretations and to summarize and analyze the data as objectively as possible.

Platform and Accessibility Scope: Our review focuses on block-based programming environments that are fully virtual and accessed through digital devices. This choice provides insights into accessibility challenges faced by learners with disabilities in these contexts, but may limit the generalizability of results to hybrid or physical block-based environments. Furthermore, the available corpus skews heavily toward accessibility for learners with visual impairments. In contrast, studies addressing motor, cognitive impairments, or multidis-ability populations are comparatively sparse. As a result, our conclusions may overly focus on the perspectives of individuals with visual impairments and may not fully address the broader accessibility needs of learners with other disabilities.

Despite these limitations, our review provides a consolidated map of the primary accessibility challenges, emerging design directions, and future research opportunities for Blockly-based and related block-based programming environments.

7 Conclusion

In this paper, we present a systematic review of the current state of accessibility research in block-based programming environments (BBPEs). We synthesize the literature focusing on improving the accessibility of programming environments for learners with diverse disabilities and highlight the challenges and limitations they face in existing BBPEs. The barriers identified span key categories of accessibility issues, including input modalities, code navigation, orientation, and manipulation, program outputs, cognitive barriers, code debugging and comprehension, and limitations arising from assistive technologies. We also analyze issues specific to particular disabilities and platforms. Additionally, we provide a comprehensive overview of participatory studies

and the methodological approaches used in the literature to evaluate these challenges.

Our review reveals ongoing efforts within the research community to address these issues. We discuss proposed designs, research prototypes, and initiatives aimed at improving the accessibility of existing BBPEs. We analyze the underlying technologies and system architectures of these solutions, and summarize feedback from participants, experts, and educators to evaluate their impact on learners with disabilities.

Notable progress has been made, although the field is still in its early stages of development. Nonetheless, significant gaps remain in realizing fully accessible and inclusive BBPEs for all learners. To guide future research, we have outlined a set of takeaways that capture both the limitations of current work and the opportunities ahead. These highlight the need to engage more diverse participant groups, extend evaluations beyond navigation to cover the complete programming workflow, develop modular and interoperable solutions that integrate seamlessly with mainstream BBPEs, and embed accessibility into the broader ecosystem of classroom practices. Through these takeaways, this review aims to inspire future research and collaborations that advance inclusive block-based programming and computing education.

Appendix

See Table 9.

Table 9 List of primary studies (PS)

ID	Title	Year	Ref.
PS1	Designing accessible block-based programming environments for persons with visual impairments	2025	[48]
PS2	An Accessible Blocks Language for Students with and without Visual Impairments	2024	[24]
PS3	Evaluating Usability and Accessibility of Visual Programming Tools for Novice Programmers—The Case of App Inventor, Scratch, and StarLogo	2024	[50]
PS4	Designing for Tinkerability and Accessibility: Developing the OctoStudio mobile app to engage blind and visually impaired learners in creating with code	2024	[27]
PS5	Accessible to whom? Bringing accessibility to blocks	2024	[21]
PS6	Designing accessible visual programming tools for children with autism spectrum condition	2023	[47]
PS7	Accessible blockly: An accessible block-based programming library for people with visual impairments	2022	[23]
PS8	The perception of teachers on usability and accessibility of programming materials for children with visual impairments	2022	[44]
PS9	Voice-enabled blockly: usability impressions of a speech-driven block-based programming system	2022	[25]
PS10	Helping Students with Motor Impairments Program via Voice-Enabled Block-Based Programming	2022	[14]
PS11	Are visual programming tools for children designed with accessibility in mind?	2020	[22]
PS12	How teachers of the visually impaired compensate with the absence of accessible block-based languages	2021	[10]
PS13	Accessible block-based programming for k-12 students who are blind or low vision	2021	[46]
PS14	Investigating challenges faced by learners with visual impairments using block-based programming/hybrid environments	2020	[13]
PS15	Accessibility of block-based introductory programming languages and a tangible programming tool prototype	2020	[45]
PS16	Accessible AST-based programming for visually-impaired programmers	2019	[43]
PS17	Blocks4All: overcoming accessibility barriers to blocks programming for children with visual impairments	2018	[26]
PS18	Evaluating the accessibility of scratch for children with cognitive impairments	2018	[49]
PS19	Design Considerations to Increase Block-based Language Accessibility for Blind Programmers Via Blockly	2017	[42]
PS20	Exploration of the use of auditory cues in code comprehension and navigation for individuals with visual impairments in a visual programming environment	2016	[40]
PS21	An accessible blocks language: work in progress	2016	[39]
PS22	An empirical evaluation of a vocal user interface for programming by voice	2015	[41]

Author Contributions RH, WA in formal analysis, investigation, and writing- original draft. RH, WA, and SL worked in supervision, conceptualization, methodology, writing, reviewing, and editing.

Funding This research received no external funding.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Grover, S., Pea, R.: Computational thinking in k-12: a review of the state of the field. Educ. Res. **42**(1), 38–43 (2013)
2. Mladenović, M., Žanko, Ž, Čuvić, M.A.: The impact of using program visualization techniques on learning basic programming concepts at the k-12 level. Comput. Appl. Eng. Educ. **29**(1), 145–159 (2021)
3. Weintrop, D.: Block-based programming in computer science education. Commun. ACM **62**(8), 22–25 (2019)
4. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.: Scratch: programming for all. Commun. ACM **52**(11), 60–67 (2009)
5. Santo, R., DeLyser, L. A., Ahn, J., Pellicone, A., Aguiar, J., Wortel-London, S.: Equity in the who, how and what of computer science education: K12 school district conceptualizations of equity in 'cs for all'initiatives. In: 2019 research on equity and sustained participation in engineering, computing, and technology (RESPECT), pp. 1–8. IEEE, (2019)
6. Ball, T., Chatra, A., de Halleux, P., Hodges, S., Moskal, M., Russell, J.: Microsoft makecode: embedded programming for education, in blocks and typescript. In: Proceedings of the 2019 ACM SIGPLAN symposium on SPLASH-E, pp. 7–12, (2019)
7. Patton, E. W., Tissenbaum, M., Harunani, F.: Mit app inventor: objectives, design, and development. In Computational thinking education, pp. 31–49. Springer Singapore Singapore, (2019)
8" GrobidStyleApplied="Changed. Google. Google blockly. https://developers.google.com/blockly, 2025. Accessed: 2025-Sep-28
9. Begosso, L. C., Begosso, L. R., Christ, N. A.: An analysis of block-based programming environments for cs1. In 2020 IEEE frontiers in education conference (FIE), pp. 1–5. IEEE, (2020)
10. Mountapmbeme, A., Ludi, S.: How teachers of the visually impaired compensate with the absence of accessible block-based languages. In Proceedings of the 23rd international ACM SIGACCESS conference on computers and accessibility, pp. 1–10 (2021)
11. Ladner, R. E., Blaser, B., Stefik, A., Ko, A. J., Kushalnagar, R.: Disability and accessibility in computer science education. In: Proceedings of the 55th ACM technical symposium on computer science education V. 2, pp. 1915–1915 (2024)
12. Gretter, S., Yadav, A., Sands, P., Hambrusch, S.: Equitable learning environments in k-12 computing: teachers' views on barriers to diversity. ACM Trans. Comput. Educ. (TOCE) **19**(3), 1–16 (2019)
13. Mountapmbeme, A., Ludi, S.: Investigating challenges faced by learners with visual impairments using block-based programming/hybrid environments. In: Proceedings of the 22nd international ACM SIGACCESS conference on computers and accessibility, pp. 1–4 (2020)
14. Okafor, O., Ludi, S.: Helping students with motor impairments program via voice-enabled block-based programming. In: International conference on human-computer interaction, pp. 62–77. Springer, (2022)
15. Koushik, V.: Making block-based programming accessible to people with cognitive disabilities. ACM SIGACCESS Access. Comput. **126**, 1–1 (2020)
16. Rasmussen, M., Lewis, O.: United nations convention on the rights of persons with disabilities. Int. Leg. Mater. **46**(3), 441–466 (2007)
17. Lipkin, P.H., Okamoto, Council on Children with Disabilities, Council on School Health, J., Norwood, K.W., Jr., Adams, R.C., Brei, T.J., Burke, R.T., Davis, B.E., Friedman, S.L., Houtrow, A.J., et al.: The individuals with disabilities education act (idea) for children with special educational needs. Pediatrics **136**(6), e1650–e1662 (2015)
18. Sonjaya, R.P., Munir, M.: Examining the impact of block-based visual programming in programming education: a systematic review. Indonesian J. Teach. Sci. **5**(1), 11–20 (2025)
19. Noone, M., Mooney, A.: Visual and textual programming languages: a systematic review of the literature. J. Comput. Educ. **5**(2), 149–174 (2018)
20. Tran, K., Bacher, J., Shi, Y., Skripchuk, J., Price, T.: Overcoming barriers in scaling computing education research programming tools: a developer's perspective. In Proceedings of the 2024 ACM conference on international computing education research-Vol. 1, pp. 312–325 (2024)
21. Stefik, A., Allee, W., Contreras, G., Kluthe, T., Hoffman, A., Blaser, B., Ladner, R.: Accessible to whom? bringing accessibility to blocks. In Proceedings of the 55th ACM technical symposium on computer science education Vol. 1, pp. 1286–1292 (2024)
22. Zubair, M. S., Brown, D., Bates, M., Hughes-Roberts, T.: Are visual programming tools for children designed with accessibility in mind?. In: Proceedings of the 12th international conference on education technology and computers, pp. 37–40 (2020)
23. Mountapmbeme, A., Okafor, O., Ludi, S.: Accessible blockly: An accessible block-based programming library for people with visual impairments. In: Proceedings of the 24th international ACM SIGACCESS conference on computers and accessibility, pp. 1–15 (2022)
24. Tabassum, M., Gray, J., Smith, D.: An accessible blocks language for students with and without visual impairments. In: Proceedings of the 55th ACM technical symposium on computer science education Vol. 1, pp. 1300–1306 (2024)
25. Okafor, O., Ludi, S.: 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, pp. 1–5 (2022)
26. Milne, L. R., Ladner, R. E.: Blocks4all: overcoming accessibility barriers to blocks programming for children with visual impairments. In: Proceedings of the 2018 CHI conference on human factors in computing systems, pp. 1–10 (2018)
27. Bulovic, K., Bentley, Z., Rusk, N.: Designing for tinkerability and accessibility: developing the octostudio mobile app to engage blind and visually impaired learners in creating with code. In: Proceedings of the 23rd annual ACM interaction design and children conference, pp. 882–886 (2024)
28. Khalajzadeh, H., Grundy, J.: Accessibility of low-code approaches: a systematic literature review. Inf. Softw. Technol. **177**, 107570 (2025)
29. Rezende, S. M., Perdigão, L. T., Silva, J. M. M., Guarda, G. F., Pinto, S. C. C. S., Fernandes, E. M., Teixeira, G. A. P. B.: Beyond the eye: making block-based programming languages accessible

to visually impaired people. In: Workshop de Pensamento Computacional e Inclusão (WPCI), pp. 01–11. SBC, (2022)
30. Mountapmbeme, A., Okafor, O., Ludi, S.: Addressing accessibility barriers in programming for people with visual impairments: a literature review. ACM Trans. Access. Comput. (TACCESS) **15**(1), 1–26 (2022)
31. Kitchenham, B., et al.: Procedures for performing systematic reviews. Keele, UK, Keele University **33**(2004), 1–26 (2004)
32. Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic literature reviews in software engineering-a systematic literature review. Inf. Softw. Technol. **51**(1), 7–15 (2009)
33. Patino, C.M., Ferreira, J.C.: Inclusion and exclusion criteria in research studies: definitions and why they matter. J. Bras. Pneumol. **44**, 84–84 (2018)
34. McHugh, M.L.: Interrater reliability: the kappa statistic. Biochem. Med. **22**(3), 276–282 (2012)
35. Fleiss, J.L., Levin, B., Paik, M.C.: Statistical methods for rates and proportions. John wiley & sons (2013)
36. Jalali, S., Wohlin, C.: Systematic literature studies: database searches vs. backward snowballing. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, pp. 29–38 (2012)
37. Felizardo, K. R., Mendes, E., Kalinowski, M., Souza, É. F., Vijaykumar, N. L.: Using forward snowballing to update systematic reviews in software engineering. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, pp. 1–6 (2016)
38. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qual. Res. Psychol. **3**(2), 77–101 (2006)
39. Koushik, V., Lewis, C.: An accessible blocks language: work in progress. In: Proceedings of the 18th international ACM SIGACCESS conference on computers and accessibility, pp. 317–318 (2016)
40. Ludi, S., Simpson, J., Merchant, W.: Exploration of the use of auditory cues in code comprehension and navigation for individuals with visual impairments in a visual programming environment. In: Proceedings of the 18th international ACM SIGACCESS conference on computers and accessibility, pp. 279–280 (2016)
41. Wagner, A., Gray, J.: An empirical evaluation of a vocal user interface for programming by voice. Int. J. Inf. Technol. Syst. Approach (IJITSA) **8**(2), 47–63 (2015)
42. Ludi, S., Spencer, M.: Design considerations to increase block-based language accessibility for blind programmers via blockly. J. Vis. Lang. Sentient Syst. **3**(1), 119–124 (2017)
43. Schanzer, E., Bahram, S., Krishnamurthi, S.: Accessible ast-based programming for visually-impaired programmers. In: Proceedings of the 50th ACM technical symposium on computer science education, pp. 773–779 (2019)
44. Van der Meulen, A., Hartendorp, M., Voorn, W., Hermans, F.: The perception of teachers on usability and accessibility of programming materials for children with visual impairments. ACM Trans. Comput. Educ. **23**(1), 1–21 (2022)
45. Utreras, E., Pontelli, E.: Accessibility of block-based introductory programming languages and a tangible programming tool prototype. In: International conference on computers helping people with special needs, pp. 27–34. Springer, (2020)
46. Das, M., Marghitu, D., Mandala, M., Howard, A.: Accessible block-based programming for k-12 students who are blind or low vision. In: International conference on human-computer interaction, pp. 52–61. Springer, (2021)
47. Zubair, M.S., Brown, D.J., Hughes-Roberts, T., Bates, M.: Designing accessible visual programming tools for children with autism spectrum condition. Univ. Access Inf. Soc. **22**(2), 277–296 (2023)
48. Mountapmbeme, A., Ludi, S.: Designing accessible block-based programming environments for persons with visual impairments. Universal Access in the Information Society, pp. 1–32 (2025)
49. Zubair, M. S., Brown, D., Hughes-Roberts, T., Bates, M.: Evaluating the accessibility of scratch for children with cognitive impairments. In: International conference on universal access in human-computer interaction, pp. 660–676. Springer, (2018)
50. Żyła, K., Chwaleba, K., Choma, D.: Evaluating usability and accessibility of visual programming tools for novice programmers-the case of app inventor, scratch, and starlogo. Appl. Sci. **14**(21), 9887 (2024)
51. Caldwell, B., Cooper, M., Reid, L.G., Vanderheiden, G., Chisholm, W., Slatin, J., White, J.: Web content accessibility guidelines (WCAG) 2.0. WWW Consortium (W3C) **290**(1–34), 5–12 (2008)
52. Velasco, C. A., Denev, D., Stegemann, D., Mohamad, Y.: A web compliance engineering framework to support the development of accessible rich internet applications. In: Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A), pp. 45–49 (2008)
53. Stefik, A., Haywood, A., Mansoor, S., Dunda, B., Garcia, D.: Sodbeans. In: 2009 IEEE 17th international conference on program comprehension, pp. 293–294. IEEE, (2009)
54. Aaron, S.: Sonic pi-performance in education, technology and art. Int. J. Perform. Arts Digit. Media **12**(2), 171–178 (2016)
55. Sánchez, J., Aguayo, F.: APL: Audio programming language for blind learners. In: International conference on computers for handicapped persons, pp. 1334–1341. Springer, (2006)