BridgeText: Messaging for People with Visual, Auditory and Speech Impairments

ROHITH SATHIAMOORTHY PANDIAN, University of California San Diego, USA
ANTARIKSHA RAY, University of California San Diego, USA
SANTOSH MUTHUKRISHNAN, University of California San Diego, USA
JUYOUNG PARK, University of California San Diego, USA
SHUJIE TIAN, University of California San Diego, USA

In an interconnected world shaped by communication technologies, individuals with sensory impairments encounter obstacles in exchanging information. This project strives to overcome these challenges by introducing BridgeText, an inclusive communication platform tailored to diverse sensory needs. Utilizing Wear OS and Amazon Echo Dot, our initiative facilitates multi-modal interactions, incorporating Braille text input, haptic feedback, and voice commands. The report details the motivation, design rationale, and seamless integration of Wear OS and Amazon Echo Dot technologies. We also highlight BridgeText's potential to enhance messaging accessibility for users with diverse sensory impairments, paving the way for comprehensive digital communication solutions.

CCS Concepts: • **Human-centered computing → Accessibility technologies**.

Additional Key Words and Phrases: Ubiquitous computing, Assistive technology, Wearable devices, Smart Watch, Voice Assistant

1 INTRODUCTION

In a world that is increasingly getting interconnected with help of communication technologies, individuals with various sensory impairments often encounter a lot of challenges when trying to send and receive information. The absence or limitations of important senses such as sight, hearing, and speech can create barriers that hinder effective communication. This project aims to address these challenges by providing a versatile and inclusive communication platform tailored to the unique needs of individuals with sensory impairments.

Initially, we dive into existing works and the motivations behind our project. While there have been existing work and technologies to enable communication amongst people with similar disabilities, there is a lack of platforms that allows communication between people with different disabilities. We also wanted to improve the user experience for users with vision impairments since they usually require standalone devices which are not that portable. While there have been attempts and previous work on facilitating braille text entry on smartwatches, these usually come with some form of limitations. Our prototype seeks to overcome these constraints by combining diverse technologies and facilitating multi-modal interactions. This innovative approach enables users to input text using braille, receive feedback through haptic vibrations, and navigate the system through voice commands, resulting in a more adaptable and user-friendly experience.

Subsequently, we outline our design rationale and the developmental process, incorporating Wear OS and Amazon Echo Dot technologies. Leveraging Wear OS for the user interface and smartwatch integration and Amazon Echo Dot for voice control, our integration strategy creates a seamless workflow. We will be elaborating more details of the system architecture, used technologies, and features of our application.

Finally to conclude, we present the outcomes of our testing and evaluation, details on our team collaboration and outlining plans for future work. Our prototype successfully integrates Wear OS and Amazon Echo Dot technologies, catering to the communication needs of smartwatch users with vision and sensory impairments.

Authors' addresses: Rohith Sathiamoorthy Pandian, University of California San Diego, California, USA; Antariksha Ray, University of California San Diego, California, USA; Santosh Muthukrishnan, University of California San Diego, California, USA; Juyoung Park, University of California San Diego, California, USA; Shujie Tian, University of California San Diego, California, USA.

The incorporation of braille text input, vibration feedback, normal text and voice commands fosters private and accessible communication on these devices. While there is a significant learning curve, our prototype showcases the potential of integrating these technologies to enhance accessibility and communication for those with vision and other impairments.

We also express our intention to perform additional testing and evaluation to refine the prototype's usability. We believe that our solution could potentially help individuals with vision and sensory impairments a more accessible and convenient means of communicating with each other. We hope our work will serve as a catalyst, inspiring further research and development in this domain.

2 MOTIVATION AND BACKGROUND

Communication serves as the cornerstone of human connection. Yet, individuals grappling with sensory impairments, whether related to hearing, speech, or vision, often encounter limitations in existing communication solutions, confining their interactions primarily to those with similar disabilities. Acknowledging this gap, our project endeavors to create a communication medium tailored to users with diverse combinations of sensory impairments.

While our initiative addresses the challenges faced by individuals with sensory impairments, it is crucial to contextualize our efforts within the broader landscape of accessibility research. A recent comprehensive review of accessibility papers presented at the ACM CHI and ASSETS conferences, spanning 1994-2019, sheds light on the evolving nature of accessibility research [7]. This study provides valuable insights into current trends, historical context, and research gaps within the accessibility community, and how blind and low-vision users remain by far the most common community of focus.

Numerous attempts have been made to bridge communication effectively among people with diverse impairments. For instance, Choudhary et al. developed a smart glove translating the Braille alphabet into text and vice versa [3]. This device communicates messages via SMS to a remote contact, catering specifically to the deaf-blind community. Additionally, Rastogi et al. devised SHAROJAN BRIDGE, a wearable technology incorporating a sensor glove, a microphone, and a speaker, aiming to bridge the communication gap for individuals who are blind, deaf, and mute [12]. While these examples showcase innovative approaches, they rely on external devices like gloves, which might be inconvenient and lack empathy from a user's perspective. In contrast, our focus is on developing inconspicuous devices commonly used by both impaired and non-impaired individuals: a smartwatch and a voice assistant.

We firmly believe that technology should empower individuals, regardless of their disabilities, to engage in inclusive conversations. By seamlessly integrating communication tools into ubiquitous smartwatches, our goal is to make connectivity an inherent part of daily life. This project represents a dedicated effort to break down communication barriers, enabling individuals not only to communicate within their specific communities but also fostering meaningful dialogues with those facing different challenges.

Despite existing research and development in this domain, a comprehensive solution using a wearable device with a small form factor is lacking, particularly for users with different sensory disabilities. Our proposed solution empowers users who are blind, deaf, mute, or a combination of these, to independently and effectively communicate through the use of a smartwatch and a virtual assistant, both ubiquitous in everyday life.

Numerous initiatives and projects aim to enhance accessibility in devices like mobile phones or smartwatches for individuals with various impairments. Efforts to integrate virtual Braille features into smartphones and smartwatches, such as Google's TalkBack Braille keyboard [5], have focused on providing a tactile interface for text input and output, primarily benefiting those with visual impairments.

Beyond mobile phones, accessibility features extend to wearable devices, particularly smartwatches. Dot Watch (https://www.dotincorp.com/), a Braille smartwatch developed by Dot Incorporation, features a dynamic Braille display. Although excelling in providing information through Braille, it leans toward displaying information rather than enabling user input.

While both watchOS and wearOS support international braille displays, incorporating a Braille keyboard remains absent. "Elle" [9], a smartwatch app, focuses on enabling users to input messages through a Braille keyboard, addressing the needs of individuals with low visibility. Its features, including Braille vibrations and Morse vibrations, enhance accessibility and facilitate effective communication for those with visual impairments. Additionally, Elle offers an Echo Dot version, catering to users who find verbal communication more convenient.

Voice assistants have been identified to exhibit similar benefits and disadvantages not only for visually impaired people but also for their partners, equalizing experience across abilities and presenting complex tradeoffs for families regarding interpersonal relationships, domestic labor, and physical safety [14]. Integration of a smartwatch with a virtual assistant, such as Alexa, Google Assistant, or Siri, allows users to use voice commands to compose messages. The voice-enabled functionality streamlines communication for individuals capable of using their voice and hearing. Efforts to integrate the Text-To-Speech (TTS) feature into virtual assistants [1] become particularly significant for users who are mute, providing a valuable alternative for inputting text and having it audibly articulated. The convergence of technologies, including virtual assistants, Braille keyboards, and Text-To-Speech (TTS) features, results in innovative solutions facilitating communication among individuals with diverse impairments. This holistic approach ensures that the smartwatch becomes a versatile tool, accommodating a spectrum of communication needs. The combination of voice-enabled commands, Braille input options, and TTS functionalities not only streamlines interactions for users with visual or hearing impairments but also provides valuable alternatives for those who are mute. Through the amalgamation of these technologies, we pave the way for inclusive communication solutions that bridge gaps and empower individuals, regardless of their specific impairments, to engage more seamlessly in the digital world. The next generation of virtual personal assistants is expected to increase the interaction between humans and machines by using technologies in addition to speech recognition, such as gesture recognition, image/video recognition, the vast dialogue and conversational knowledge base, thereby opening up exciting opportunities for supporting accessibility [6].

Braille, a foundational mode of communication for individuals with multi-sensory impairments, has witnessed notable technological integration. Pioneering solutions like HoliBraille [10], V-Braille [4], BrailleTouch [13], and BrailleType [11] showcase tactile feedback on touchscreen devices, primarily tailored for larger screens like smartphones. However, the accessibility of braille interfaces for smaller wearables such as smartwatches remains a critical gap in the current landscape. Exploring the field of braille-technology integration unveils both progress and limitations. Projects like BrailleEnter [2], and EdgeBraille [8] focus on text entry for Braille, designed predominantly for smartphones. Yet, these solutions encounter challenges in adapting to smaller form factors and lack versatility for wearable devices. In the context of our project, BridgeText, we emphasize the significance of universally accessible braille interfaces within communication solutions. Our ongoing efforts are dedicated to extending braille technologies to compact wearables, ensuring inclusivity for individuals with multi-sensory impairments, especially in the evolving landscape of smartwatches. While existing studies underscore the potential of braille-technology integration, it is essential to recognize the current emphasis on larger screens and external devices. The forward trajectory involves refining these technologies to accommodate smaller devices, thus closing the accessibility gap and fostering inclusivity for users with diverse sensory impairments.

3 DESIGN

3.1 WearOS application

Our WearOS application can be divided into two distinct sections, each serving a specific purpose. The first section, Standard Messages, is dedicated to typing and reading conventional text messages. Users can compose and peruse standard messages within this interface. On the other hand, the second section, Braille Messages, is tailored for users interested in typing and reading text messages represented in Braille patterns. To ensure the optimal design and functionality for each case, the WearOS application underwent several iterations of design changes over the four weeks.

3.1.1 *Standard Messages.*

In the early stage of development, our primary focus was on foundational aspects such as defining the package structure, addressing separation of concerns, and crafting essential functionalities like the HTTP client, utility, and logger functions. We allotted a lower priority to the general UI, using the standard Android widgets instead of the material library. The device was registered by entering the complete user id. Subsequently, we introduced user-specific message handling for the Standard Messages and added a scrollbar to display long messages.

To enhance the overall user experience, we implemented the Material library's dark theme, providing a cohesive and visually pleasing design to our application. Furthermore, we conducted improvements on the Messages Read screen by updating the previous and next buttons. This adjustment was made to optimize screen space, ensuring a more comfortable and efficient viewing experience for users. In addition, we finalized the icons for the buttons, adding a visual element that enhances both functionality and aesthetics.

3.1.2 *Braille Messages.*

Initially, we delved into researching and prototyping the layout of the Braille keyboard. As we navigated the process of devising a user-accessible Braille Keyboard layout, we soon realized that arriving at the definitive final design would be a challenging journey. Given the constraints of time, we made a decision to prioritize functionality over design intricacies. We chose to begin the implementation process with a basic and intuitive design, intending to refine and enhance it through testing different features and gathering user feedback. This decision resulted in the development of the initial design of the Braille keyboard, featuring a simple arrangement of 8 buttons in a grid with 2 columns and 4 rows (Fig.1). The upper 6 buttons represented the Braille cell (Fig.2), while the lower two buttons served for customized functions. The left button facilitated going back to the previous character or clearing the current character input (for reading or writing, respectively), and the right button served the dual purpose or registering a character and sending a message (for writing) or moving to the next message (for reading). This approach allowed us to focus on swiftly integrating core functions, acknowledging that refinement and fine-tuning of the design would be addressed in subsequent stages of the development.

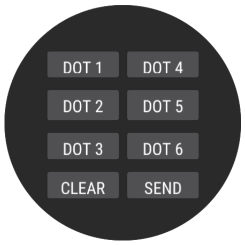


Fig. 1. Initial Layout of Braille keyboard

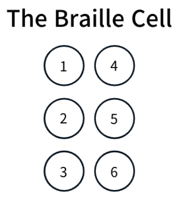


Fig. 2. The Braille Cell

After implementing and ensuring essential features for reading and writing Braille Messages, we then moved to improve the design. The initial layout of 8 buttons was evidently prone to misclicks during blindfolded testing, prompting us to make a strategic decision to reduce them to necessary 6 buttons. Additionally, we observed that the initial design included unnecessary empty spaces surrounding the buttons. Recognizing the lack of visual access for blind users, we opted to remove these superfluous spaces and maximize the utilization of the watch face with the whole 6 buttons (Fig.3).

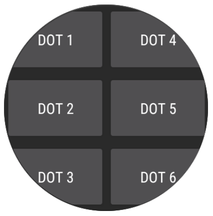


Fig. 3. Second Layout of Braille keyboard

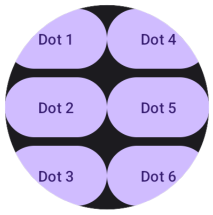


Fig. 4. Final Layout of Braille keyboard

Reducing the original 8 buttons to 6 by eliminating the bottom two buttons necessitated the development of a new solution to replicate the functions previously handled by the removed buttons. To address this, our initial approach involved overriding swipe gestures, as well as incorporating long-press functionality for one of the 6 buttons. Specifically, swiping right to left was designated for registering a character or moving to the next character, while swiping bottom to top served to clear a character or move to the previous character. Additionally, a long press on dot 6 was designated for sending a message. However, we found that partially overriding swipe gestures in only two directions resulted in unpredictable behaviour with the swipe-to-dismiss action interfering with the bottom-to-top swipe gesture. On the other hand, a complete overriding of swipe gestures would not follow WearOS navigation guidelines and lead to inconsistent UX behaviour, forcing the users to use the physical button for back navigation.

As a result, we opted to abandon the swipe gestures and integrate additional functionalities by utilizing more buttons from the essential set of 6. Taking into account the location of these 6 buttons, we chose to utilize the edge buttons (1, 3, 4, and 6), as they were more intuitive and easier to remember. Also, after blindfolded testing, we transitioned from a standard button-clicking motion to a long-press approach. Given that blind users lack visual access to the separation of these 6 buttons and must navigate positions by touch, relying on standard clicks was prone to misclicks. Conversely, long-press proved effective in preventing this issue. Beyond this consideration, our focus on delivering the best tactile response to users led us to favor long-press approach. The earlier approach of standard button-clicking provided vibration feedback to the user's writs, whereas transitioning to the long-press method allowed for haptic feedback directly to the user's fingertip, aligning more closely with the tactile nature of real Braille. The final layout of the Braille keyboard, now refined with the Material library and icons for the buttons, along with the conclusive instructions for typing/reading Braille messages, is outlined below:

- **Inputting Braille Messages:**
  - Long-press dot 1-6: Input the raised dots order-agnostically; each pressed button provides haptic feedback on the user's fingertips.
  - Double-tap dot4: Send the whole sentence; watch will vibrate once.
  - Double-tap dot3: Clear a letter; watch will vibrate twice shortly.
  - Double-tap dot6: Register a letter; watch will vibrate once.
- **Reading Braille Messages:**
  - Long-press dot 1-6: Read the raised dots order-agnostically; each pressed button provides haptic feedback on the user's fingertips if raised dots are present.
  - Double-tap dot1: Go back to the previous text message.
  - Double-tap dot4: Move to the next text message.
  - Double-tap dot3: Go back to the previous letter.
  - Double-tap dot6: Move to the next letter.
    (Each action triggers a single vibration for feedback.)

3.2 Voice Assistant

The voice assistant for BridgeText is designed to assist users with speech impairments in a simple and intuitive manner. When the skill is invoked, it greets the user and prompts them for their next action. Users have the option to either send or receive a text message. The flow is intentionally minimalistic to ensure ease of use.

3.2.1 User Confirmation.

For user confirmation, we employ the logic of checking the username API return to determine whether it contains an empty ID. If there is no corresponding user found, Alexa will provide a voice prompt to initiate the conversation again. Fig 5 shows the overview of the user confirmation flow.

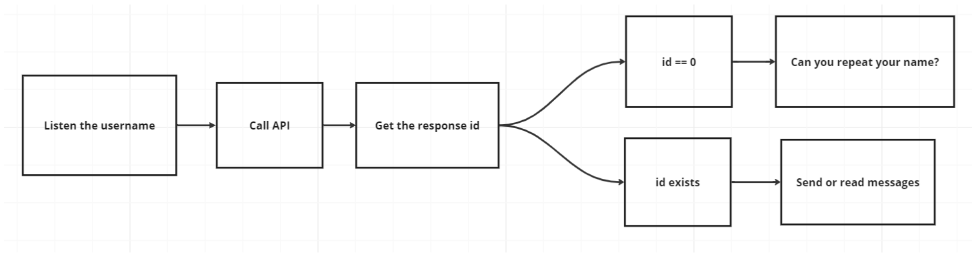


Fig. 5. User Confirmation

3.2.2 Flow Design.

Send Messages: When sending a message, the user first states the recipient they want to send it to. If the recipient's username exists (ID query is not empty), the recording of the message begins. After finishing speaking, the Alexa voice flow will repeat the message, and the user confirms the content for a second time. If the content is correct, the message will be sent to the recipient user, and a success or failure notification will be provided. After the process is complete, the user can choose to send another message, check and play unread messages, or exit the program. Fig 6 shows the overview of the flow for sending messages.

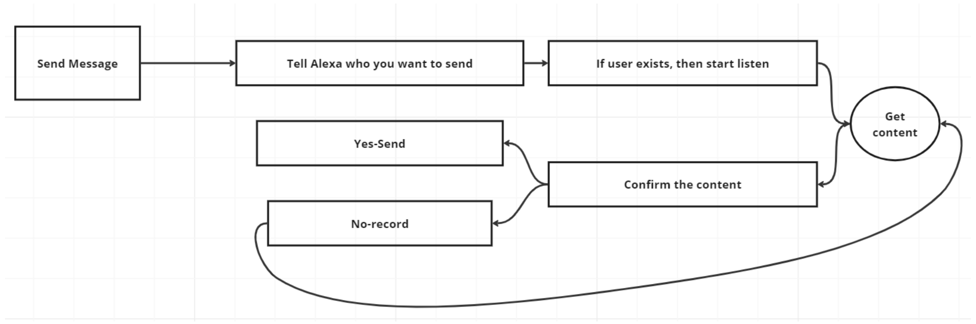

Send Message
Tell Alexa who you want to send
If user exists, then start listen
Get content
Yes-Send
Confirm the content
No-record

Fig. 6. Flow of Sending Message

Play Messages: If a corresponding user is successfully identified, the message check API call will be made to check for three different scenarios: If there are no unread messages, Alexa will transition to the main voice flow where the user can choose to send a message or exit directly. If there is only one unread message from a specific sender, Alexa will directly inquire whether to play the unread message from that sender. If there are unread messages from multiple senders, Alexa will prompt the user to make a selection among them. Fig 7 shows different scenarios for playing messages.

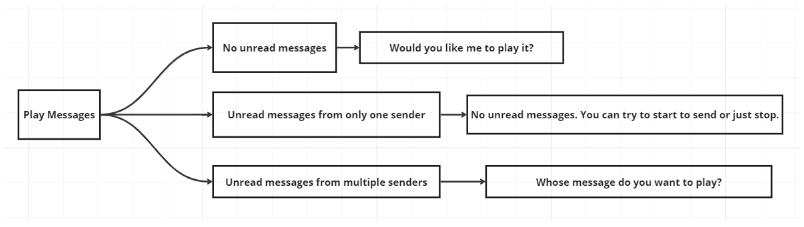

Play Messages
No unread messages
Would you like me to play it?
Unread messages from only one sender
No unread messages. You can try to start to send or just stop.
Unread messages from multiple senders
Whose message do you want to play?

Fig. 7. Different Scenarios in Play Messages

4 SYSTEM DEVELOPMENT

4.1 Architecture

The wearable device and the voice assistant act as the clients in our system, whereas our server is hosted in the cloud using an Amazon Elastic Compute Cloud (Amazon EC2) instance. The server exposes a RESTful service, and the clients make HTTP requests over the network to read, write, and update information.

The BridgeText application (for WearOS and Alexa) lies in the application layer of the OSI model, as shown in Fig 8. We use the HyperText Transfer Protocol (HTTP) which uses the Transmission Control Protocol (TCP) in the transport layer. The application uses a WiFi connection to communicate with the server via the Internet. For the watch app, user interactions with the UI (with or without data inputs) update the XML views. This may include an intermediate step of making a REST API call to the server to create or update a resource or fetch data to update the data model, which in turn may reflect an update in the information displayed by the views. A read/write operation with the app-specific storage in the file system is performed when retrieving or storing the user information.

Our application, powered by a MongoDB database and a Node.js server, offers functionalities for user management and messaging. Below, we provide brief insights into the application's API routes and data models.

- The User model captures essential user information, providing a comprehensive user management system.
- The Message model encapsulates the intricacies of our messaging system, facilitating seamless communication between users.
- Our user-centric API routes empower administrators to manage user data effectively:
  - **Get All Users:** Retrieves a comprehensive list of all users, enabling a quick overview of the user base.
  - **Create User:** Facilitates the addition of new users to the system.
  - **Get User ID:** Retrieve user ID information given the username.
  - **Get User by ID:** Enables targeted retrieval of user information based on the unique user ID.
  - **Update User:** Allows for modification of user information.
  - **Delete User:** Provides a mechanism for removing users from the system.
- Our messaging API routes cater to the dynamic and interactive nature of communication within the application:
  - **Create Message:** Send a new message to a specified user.
  - **Get Messages:** Retrieve messages for a user based on specified parameters.
  - **Get Message Metrics:** Delivers insights into messaging activity for an individual user.

These API routes, combined with our data models, form the backbone of our application, ensuring a seamless and feature-rich user experience. In the current version of our application, even though we define a simple text file in the app-specific storage to store user credentials, due to time and capacity constraints, we do not fully define or handle privacy and security. However, it is possible and necessary to implement a comprehensive authentication and security layer, which is addressed in Section 7. Fig 8 shows the overview of request-response and data flow within the components of our system.

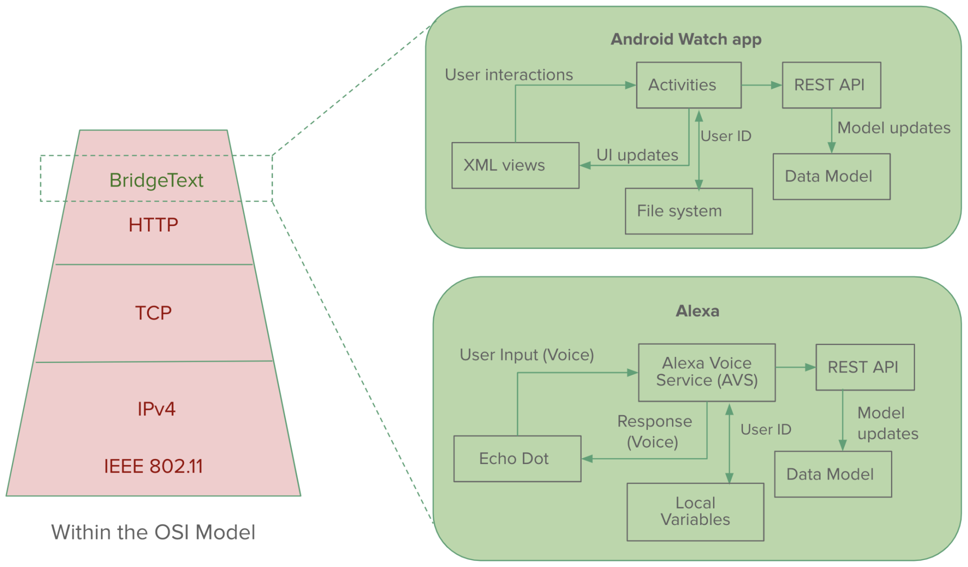


Fig. 8. The application architecture (WearOS and Alexa) within the OSI model

4.2 Technology used

4.2.1 Back-end Application.

The BridgeText back-end application is developed using Node.js, a javascript runtime platform. We choose Node.js as it is offers efficiency, speed and performance while being easy to use due to the team's familiarity with JavaScript. Consequently, the development and code review is less time-consuming, increasing overall developer productivity. See more at https://nodejs.org/en/about.

- **express**: A robust web application framework for Node.js, simplifying the development of backend services and APIs. See more at https://expressjs.com/.
- **axios**: A versatile HTTP client for Node.js, ideal for making asynchronous requests to external APIs from the backend. See more at https://nodejs.org/en/about.
- **cors**: Cross-Origin Resource Sharing middleware for Express.js, ensuring secure communication between the backend and client-side applications. See more at https://expressjs.com/en/resources/middleware/cors.html.
- **helment**: A security middleware for Express.js, enhancing the security of the backend by setting various HTTP headers. See more at https://helmetjs.github.io/.
- **http-Status**: A utility for handling HTTP status codes in your backend application, simplifying status code management. See more at https://github.com/adaltas/node-http-status.
- **mongoose**: A powerful MongoDB object modeling tool for Node.js, providing a schema-based solution for backend data management. See more at https://mongoosejs.com/.

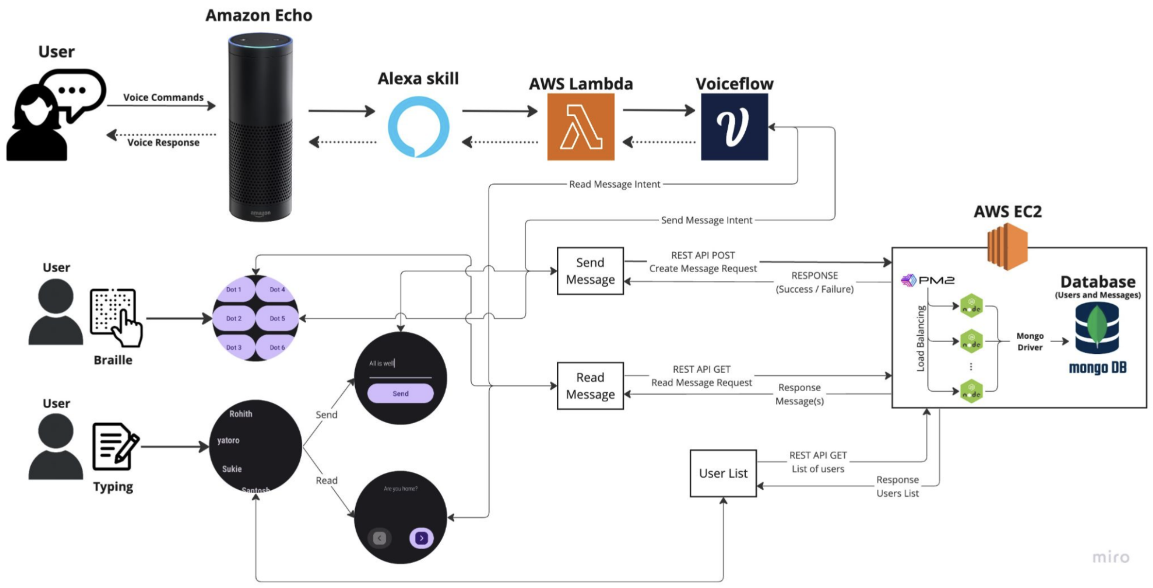


Fig. 9. Networking and I/O, request-response and data flow within the components of the system

- **nodemon**: A development utility for nodejs, monitoring changes and automatically restarting the server during development. See more at https://nodemon.io/.
- **PM2**: PM2 is a robust process manager tailored for Node.js applications in production. With features like automatic restart, load balancing, and seamless deployment, PM2 ensures high availability and efficient monitoring of our backend services. See more at https://pm2.keymetrics.io/.

4.2.2 Database.

We utilize MongoDB to power the server in our BridgeText project. Opting for a NoSQL approach, MongoDB offers the flexibility and scalability crucial for our design. Its compatibility with JavaScript, facilitated by the JSON data format, ensures smooth integration within our Node.js environment. See more at https://www.mongodb.com/.

Additionally, for ease in development purposes and enhanced database management, we utilize MongoDB Compass, which provides a graphical user interface. See more at https://www.mongodb.com/products/tools/compass.

As mentioned above, we have two tables in the database connected to the server.

- The user model schema is given in Table 1. Here, the 'name' field captures the user's name, 'email' ensures uniqueness in lowercase, and 'role' denotes the user's role, with a default setting of 'user'. The 'admin' role is used for administrators to access routes with certain additional privileges.

Name	Type	Attributes
name	String	Required, Unique
email	String	Required, Unique, Lowercase
role	String	Enum: ['user', 'admin'], Default: 'user'

Table 1. User Model Schema

- The message model schema is given in Table 2. Within the ‘messages’ array, detailed subdocuments ensure a comprehensive representation of each message. The message subdocuments model schema is given in Table 3. This structure captures sender, receiver, message content, and the read status, ensuring a rich and detailed messaging experience.

Name	Type	Attributes
userId	String	Required
messages	Array	Required

Table 2. Message Model Schema

Name	Type	Attributes
from	String	Required
to	String	Required
message	String	Required
readStatus	Boolean	Required, Default: false

Table 3. Message Subdocument Schema

4.2.3 *API Schema.*

- User API Schema:
  - **Get All Users:**

Listing 1. Get All Users API Schema

```
GET /api/1/user

Headers:
- Content-Type: application/json
```

  - **Create User:**

Listing 2. Create User API Schema

```
POST /api/1/user

Headers:
- Content-Type: application/json

Body (raw JSON):
{
    "name": "test",
    "email": "test@ucsd.edu",
    "role": "user"
}
```

  - **Get User:**

Listing 3. Get User API Schema

```
GET /api/1/user/:userID

Headers:
- Content-Type: application/json
```

- **Update User:**

Listing 4. Update User API Schema

```
PUT /api/1/user/:userID

Headers:
- Content-Type: application/json

Body (raw JSON):
{
    "name": "random",
    "email": "random@ucsd.edu",
    "role": "user"
}
```

- **Delete User:**

Listing 5. Delete User API Schema

```
DELETE /api/1/user/:userID

Headers:
- Content-Type: application/json
```

- **Get User ID:**

Listing 6. Get User ID API Schema

```
GET /api/1/user/id

Query Params:
- userName: test

Headers:
- Content-Type: application/json
```

- Messages API Schema:
  - **Create Message:**

Listing 7. Create Message API Schema

```
POST /api/1/message

Headers:
- Content-Type: application/json
```

```
Body (raw JSON):
{
    "from": "senderUserId",
    "to": "receiverUserId",
    "message": "test message"
}
```

– **Get Messages:**

Listing 8. Get Message API Schema

```
GET /api/1/message

Query Params:
- userId (Receiver): receiverUserId
- from (Sender): senderUserId
- consolidated: false (false will return one message | default =
    true)
```

– **Get Message Metrics:**

Listing 9. Get Message Metrics API Schema

```
GET /api/1/message/metrics

Query Params:
- userId: test
- responseType: text (object | text)
```

4.2.4 Deployment.

Our server infrastructure is hosted on an Amazon Elastic Compute Cloud (Amazon EC2) instance, utilizing Amazon Linux as the operating system. Both the Node.js application and the MongoDB database were installed on the EC2 instance.

We opted for Amazon EC2 due to its notable advantages in providing scalable and on-demand compute capacity. EC2 allows us to easily scale our server infrastructure based on fluctuating workloads, ensuring optimal performance and cost-efficiency for our BridgeText project.

To enhance reliability, we utilized the Process Manager 2 (PM2) tool for efficient process management and automatic restarts. This configuration, comprising Amazon EC2, Amazon Linux, Node.js, MongoDB, and PM2, establishes a scalable and resilient foundation for our cloud-hosted server. For client-server interaction, clients access the server using the Public IPv4 DNS.

4.2.5 WearOS Application.

For the wearable device, we use the Samsung Galaxy 4 smartwatch. We developed a WearOS application for the smartwatch that allows users to read messages from each other, and also write messages to each other. Our application is developed for Android 13 (Tiramisu), although we also provide backward compatibility till Android 11 (R), thus allowing most smartwatches released within the last four years to run our application.

- We develop the application using Android Studio in Java, and for the HTTP client, we use the Retrofit library (https://square.github.io/retrofit/), which provides a type-safe HTTP client for Android and Java. Retrofit is built on top of OkHttp (https://square.github.io/okhttp/), another popular networking library

by Square, which provides a higher-level abstraction to define API endpoints and handle responses in a type-safe manner.
- OkHttp HttpLoggingInterceptor is used to log the network request and response information. (See more at https://square.github.io/okhttp/3.x/logging-interceptor/okhttp3/logging/HttpLoggingInterceptor.html)
- Retrofit's GSON converter is used for serialization of the data to and from JSON for the HTTP requests (See more at https://github.com/square/retrofit/blob/master/retrofit-converters/gson/README.md).
- Retrofit's Scalar converter is used (https://central.sonatype.com/artifact/com.squareup.retrofit2/converter-scalars) for scalar value types.

4.2.6 *Voice Assistant Application.*

For the voice assistant, we utilized an Alexa Echo Dot. We developed an Alexa skill using Voiceflow, enabling users to interact with the skill and access our features, allowing them to read and send messages to each other. The Alexa skill is built and deployed on the Alexa Voice Service (AVS). Using Voiceflow instead of building the skill on Lambda from scratch ensures that Voiceflow is cross-platform, and we can easily deploy it on other services like Google Voice Assistant.

- We continued to use the method of integrating Alexa into Voiceflow using Lambda functions. We didn't use Lambda functions even during the development process.
- We employed the developer console for debugging the project.
- We had an understanding of Natural Language Processing (NLP) and used the Alexa Skills Kit (ASK) to process and interpret user voice commands.
- For building the interaction and logic of voice applications, we utilized Amazon's tools and platform to create and manage skills.

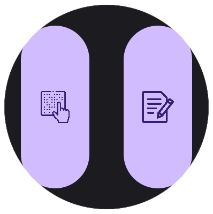
4.3 SmartWatch Features

Our smartwatch application provides the user with an option to write a text message to another user, and also view the list of messages from another user that is not yet seen. In addition to the standard Android keyboard, we developed our own Braille keyboard that can be used to read and write text messages. The device needs to be first registered to a user before being able to send or receive messages.

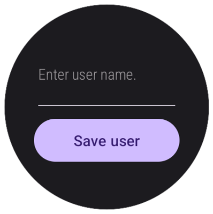
4.3.1 App entry. The entry point to the application is its icon on the home screen, which on click launches the application. We list the different screens of the watch application in Fig 10. The main menu shown in Fig 10a consists of two buttons, where the left button displays a touch icon for the Braille menu and the right button displays a typed input icon for the Standard menu.

4.3.2 Device registration. On the Standard Messages menu as in Fig 10e, clicking on the Messages text view launches the User id registration screen as in Fig 10b where the user name can be entered for registering the device. We placed it here as we believe that for a low-vision user, the first-time device registration could be performed by a friend or a family member who would be more at ease using the standard keyboard. Device registration is essential when the application is installed for the first time, and thereafter it is possible to re-register it as needed. The user enters the user name and clicks on the Save user button, which registers the user id to the device by making an HTTP GET request to the server with the user name as a query parameter to retrieve the user id associated user id. The result of the operation is notified to the user using a Toast message. If the user id is found for the given user name, a successful message is shown to the user as in Fig 10c. On the other hand, if the user id is not found, the server returns an empty message and a failure message is shown to the user as in Fig 10d. The user id is persisted into a file in the app-specific storage of the WearOS file system.

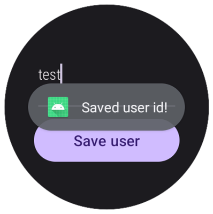
(a) Main menu



(b) User id registration

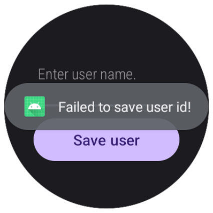


(c) Registration success



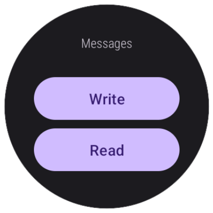
Saved user id!

(d) Registration failure

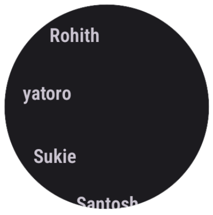


Failed to save user id!

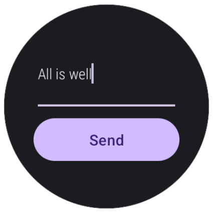
(e) Standard Messages menu



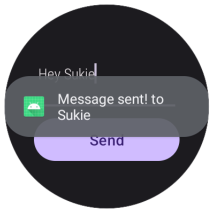
(f) List of sample users registered for the service



(g) Message write screen

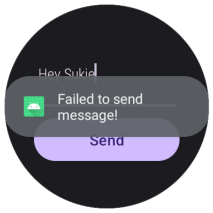


(h) Message successfully sent



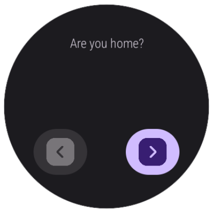
Message sent! to Sukie

(i) Message failed to send

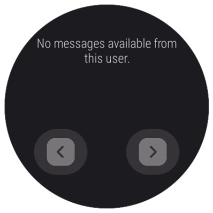


Failed to send message!

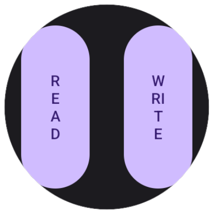
(j) Message read (no previous message available


Are you home?

(k) Message read (no messages available)


No messages available from this user.

(l) Braille Messages menu



(m) Braille keyboard

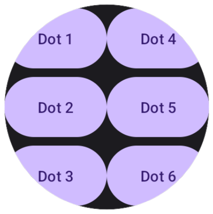


Fig. 10. The different screens of the watch application on a small size round WearOS device

4.3.3 *Standard Messages.*

Our application provides a Standard Messages menu to send messages by typing text with the standard WearOS keyboard. The menu is shown in Fig 10e, and contains a TextView and two buttons. The buttons Write and Read navigate to the list of users as in Fig 10f, from which a specific user is selected to write the message to or read the messages from. We obtain the list of users by making an HTTP GET request to the server. A user from this list of users is selected to write a message to or read the messages from (based on the previous screen selection), and the selection launches the respective Write or Read screen, while also passing the user id of the user registered on the device by reading it from the file system.

The Message write screen is shown in Fig 10g, which contains an EditText field to enter the text, and a button to send the message. An HTTP POST request is used to send the message, with the body of the message containing the information of the sender (known from the user selection context), the receiver, and the actual message. The A success Toast message as in Fig 10h notifies the user of a successful response from the server (201 Created), whereas a failure Toast message notifies the user of an error response (400 Bad Request) as in Fig 10i.

The Message read screen consists of a TextView to display the message and two buttons (styled as arrow icons in the left and right directions) that can be used to move between the previous and next message within the list of messages from the sender. A scrollbar allows the viewing of particularly lengthy messages, and the previous and next buttons are enabled and disabled based on the number of messages remaining in the respective direction. Fig 10j and Fig 10k depict the situations where there are no previous messages available and no messages available respectively. On entering the Message read screen, we make an HTTP GET request to the server to get all messages from the user selected on the previous screen, and switching between different messages is performed by updating the TextView.

4.3.4 *Braille Messages.*

Our application also provides a Braille Messages menu to write and read messages using touch-based interaction with our Braille Keyboard. We designed the keyboard in the layout of a standard Braille cell, with 6 dots laid across three rows and two columns. We use haptic feedback through the use of watch vibrations to represent raised dots. Referencing Fig 10l, the Braille Messages menu comprises two buttons: READ and WRITE. The buttons transition users to the identical Braille Keyboard, depicted in Fig 10m, yet with a distinction in modes-READ for reading messages sent to the user, and WRITE for composing and sending new messages to a specific user. In contrast to our standard messages' user list feature, which is challenging to implement for low-vision users, we have opted for an alternative approach. Instead, we prepend the username in front of each message. In the reading mode, users identify the message sender by reading the username appended at the beginning of each received message. In the writing mode, users select the intended recipient by typing the receiver's username before inputting the message content.

To facilitate the transition of Braille patterns into text messages and vice versa, we have devised a system utilizing a JSON file to store mappings between braille patterns and English alphabet, numbers (0 - 9), and essential punctuation marks: period (.), comma (,), question (?), exclamation (!), and colon (:). Considering time and technological constraints, our focus has been exclusively on Grade 1 Braille due to its more straightforward implementation compared to more complex alternatives. In representing braille patterns, we employed a 6-digit binary string, where '1' indicates a raised dot, and '0' signifies its absence. As an illustration, consider the Braille pattern for the English alphabet 'a', which is represented as a raised dot in the first position, resulting in our binary mapping of '100000'. Our JSON file consists of two distinct maps: one for the alphabet, which includes punctuation marks, and another for numbers. This separations is implemented due to the overlapping patterns between certain alphabets and numbers.

When the user clicks the READ button within the Braille Messages menu, our application initiates an HTTP GET request to the server. This request is designed to retrieve all messages that have been sent to the user, including the corresponding usernames of the message senders. Upon retrieving the messages sent to the user, we proceed to restructure the presentation of these messages for the user. First, we prepend the retrieved username of the sender to each message. To distinguish between the sender from the message content, we introduce a colon (':') in between, as a separator between the username and the actual message. Subsequently, these restructured messages undergo translation into Braille patterns. Each character is translated into the corresponding 6-digit binary string based on the aforementioned Braille map. As noted earlier, there are overlapping patterns for numbers and some alphabets. To address this, the Braille system incorporates an indicator pattern that precedes the main Braille pattern when the character is a number. Similarly, another indicator pattern is used to distinguish between lowercase and uppercase letters, preceding the main Braille pattern when the character is a capital letter. Thus, when a character is either a number or a capital letter, we append the corresponding indicator Braille pattern before the main Braille pattern. Through this process, all messages sent to the user are converted into concatenations of 6-digit binary strings. Users can read these messages by interacting with the 6 buttons on the Braille keyboard layout. During a long press on each button, the watch delivers haptic feedback on the screen if the corresponding section of the reconstructed binary string message is 1, indicating an upraised dot. Conversely, if the section is 0, no haptic feedback is given. When there is no feedback from any of the 6 buttons, it signifies a space in the Braille pattern, indicating no dots raised. Users can navigate to the next character by double-tapping the dot6 button, located at the bottom right of the watch face, and move back to the previous character by double-tapping the dot3 button, located at the bottom left of the watch face. Upon completing the reading of a message and desiring to proceed to the next one, users can double-tap the dot4 button, located at the top right. Similarly, to revisit the previous message, users double-tap the dot1 button, located at the top left. Following each double-tap, when characters or messages are present, the watch vibrates to indicate the successful execution of the intended behavior. Conversely, if there's no character or messages to read (at the beginning or end of a message, or when no previous or next message exists), no feedback is provided.

When users enter the writing or sending mode by selecting the WRITE button in the Braille Message menu, their initial step is to input the username of the intended recipient before proceeding to compose the message. During this process, users input each character of the recipient's username using the corresponding Braille patterns. This is achieved by long-pressing the button associated with the upraised dots for each character. When the users long-press a button, the watch provides haptic feedback on the screen to signal that the button has been clicked. With each button press, the corresponding digit of the 6-digit binary string becomes 1, remaining 0 otherwise. Users double-tap the dot6 button to register each character, and the watch vibrates once to signify the successful execution of the action. To reset clicked buttons during character registration, users can double-tap the dot3 button, accompanied by a brief double vibration indicating the character reset. Upon completing the input of the recipient's username, users double-tap the dot4 button. The concatenation of 6-digit binary strings is then translated into letters using the previously mentioned mapping feature. Subsequently, our application initiates an HTTP GET request to the server to retrieve the userId of the recipient. A successful request prompts a single vibration, while an unsuccessful one results in a double short vibration, signaling an issue with the input username and prompting the user to retype the recipient's username. After successfully setting the recipient, users can input the message they want to send using a similar process to setting the recipient's username. Each character of the intended text message is input into Braille patterns by long-pressing the corresponding button, with each pressed button's corresponding digit becoming 1 and remaining 0 otherwise. The haptic feedback feature is employed, providing the same tactile response as when registering the recipient. Clearing a character and registering a character follow the same process. Upon completing the writing of a message and desiring to send it, users double-tap the dot4 button. The concatenation of the 6-digit binary strings is then converted to text based on the aforementioned mapping feature. Subsequently, our application initiates an HTTP POST request to the server. If the POST request is successful, the watch vibrates to indicate to users that the message was sent successfully.

4.3.5 Aesthetics. We use the Material theme for our application, as the Material library offers modern, stylistic customization of UI widgets including color, typography, and shape attributes. We use buttons with small icons instead of text to maximize the screen space when reading the messages, and also follow the WearOS design guidelines wherever possible, such as using a WearableRecyclerView with a list adapter for the circular scrolling of list items for the list of users.

4.3.6 Accessibility. We ensure that all our UI elements are screen-reader compliant. Audio feedback from TalkBack allows users to navigate the application without directly looking at the watch.

4.4 Voice Assistant Features

4.4.1 Welcome Statement (User Login Confirmation).

When a user uses Alexa for the first time, they need to initiate the skill by saying: “Hi, Alexa, open Bridge Text." Alexa will respond by playing a welcome statement: “Hello, could you tell me your name?" Alexa will then wait to capture the user’s reply as their current username. At this point, the user will say their username, and Voiceflow will call the “get user id" API to check if the user exists. The Voiceflow workflow is shown in Fig 11.

- If the user exists, Alexa will respond with: “Hello [user’s name], would you like to send a new message or play your received messages?"
- Else if the user does not exist, Alexa will ask the user to register or repeat their name, saying: “Sorry, maybe you need to register first. Can you please repeat?"

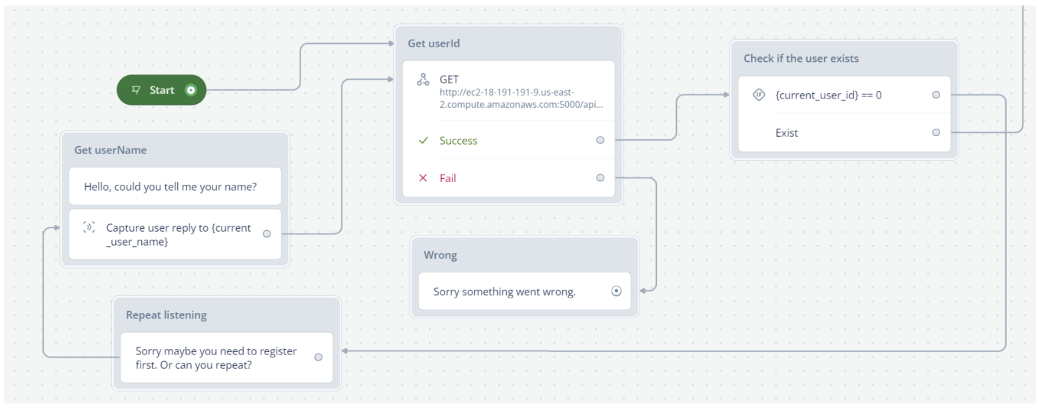

Start
Get userId
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Check if the user exists
{current_user_id} == 0
Exist
Get userName
Hello, could you tell me your name?
Capture user reply to {current _user_name}
Wrong
Sorry something went wrong.
Repeat listening
Sorry maybe you need to register first. Or can you repeat?

Fig. 11. User Confirmation Voiceflow

4.4.2 Main Action choice.

After successful user ID confirmation, Alexa begins capturing user intent and stores it in an action. If the user’s utterance contains the word “send" (indicating the intent to send a message), Alexa calls the API to get a text summarizing the received message status and returns it to the user. The Voiceflow workflow is shown in Fig 12.

- If the user’s utterance contains the word “play" (indicating the intent to play unread messages), Alexa asks the user specifically whom they want to send a message to: “Who would you like to send a message to today?" Alexa then waits to capture the user’s response, storing it in the “recipient name" variable.
- If the user’s intent cannot be correctly matched, the conversation flow is exited.

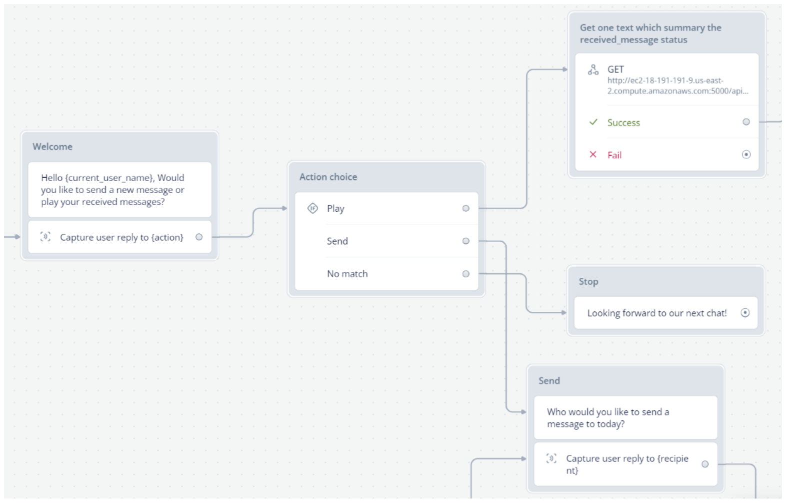

Get one text which summary the received_message status
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Welcome
Hello {current_user_name}, Would you like to send a new message or play your received messages?
Capture user reply to {action}
Action choice
Play
Send
No match
Stop
Looking forward to our next chat!
Send
Who would you like to send a message to today?
Capture user reply to {recipient}

Fig. 12. Main Action Choice Voiceflow

4.4.3 *Sending a Message to a Specific User.*

Upon receiving the name of the recipient that the user wishes to send a message to, the same API is used to confirm whether the returned ID is empty, thereby verifying the user's identity. The Voiceflow workflow for recipient confirmation is shown in Fig 13.

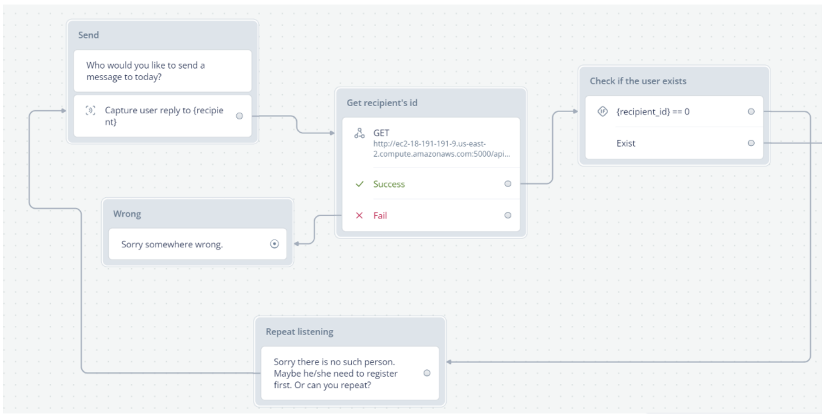

Send
Who would you like to send a message to today?
Capture user reply to {recipient}
Get recipient's id
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Check if the user exists
{recipient_id} == 0
Exist
Wrong
Sorry somewhere wrong.
Repeat listening
Sorry there is no such person. Maybe he/she need to register first. Or can you repeat?

Fig. 13. Recipient Confirmation Voiceflow

After confirming the recipient's identity, Alexa prepares to send the message. It says, "Please say the message that you want to send to recipient." Then, Alexa waits to capture the user's voice response, storing it in the "message" variable. Next, it asks the user to confirm the specific content of the message: "Please confirm if this is the message you would like to send: message. Say yes to confirm or no to record the message again." It captures the user's response in the "confirmation" variable. The Voiceflow workflow is shown in Fig 14.

- If the user's response contains the word “yes" (indicating an intent to confirm the message's accuracy), Alexa calls the API to send the message and responds with success: “Your message has been successfully sent to recipient. Do you want to send more? You can also start to check your unread messages or just stop."
- If the user's response contains the word “no" (indicating an error in the message), Alexa captures the user's voice response again and stores it in the "message" variable for re-recording.

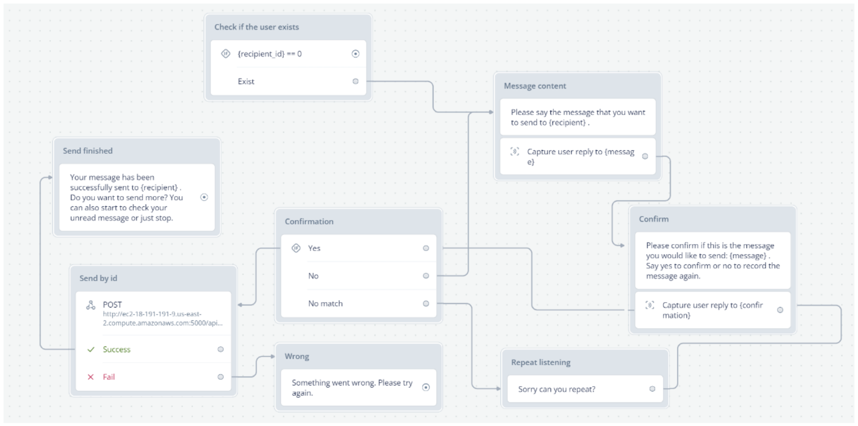

Check if the user exists
{recipient_id} == 0
Exist
Message content
Please say the message that you want to send to {recipient} .
Capture user reply to {message}
Send finished
Your message has been successfully sent to {recipient} . Do you want to send more? You can also start to check your unread message or just stop.
Confirmation
Yes
No
No match
Confirm
Please confirm if this is the message you would like to send: {message} . Say yes to confirm or no to record the message again.
Capture user reply to {confirmation}
Send by id
POST
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Wrong
Something went wrong. Please try again.
Repeat listening
Sorry can you repeat?

Fig. 14. Send Message Voiceflow

4.4.4 *Playing Unread Messages from a Specific User.*

For playing unread messages, first, after calling the API to query the summary of unread messages, based on the returned results, determine if there are any unread messages. If there are none, respond with: “No unread messages. You can try to start sending or simply stop." If there are unread messages, then check whether there are unread messages from multiple users (details to follow). If there are messages from multiple users, ask: “You can tell me whose messages you want to play." Then, use the same method to confirm the specific username said by the user. The Voiceflow workflow is shown in Fig 15.

Next, retrieve the messages from the specific sender one by one through an API call: “Got it! You want to check the messages from specific_sender. Now let me play them one by one." After playing all the messages, Alexa responds: “That's all. You can check messages from other friends if there are any, or you can also start sending or simply stop." The Voiceflow workflow for playing messages is shown in Fig 16.

When there is only one user with unread messages, Alexa directly asks the user if they want to play: “Do you want me to play?" and proceeds to the aforementioned process of retrieving messages through the message playback API, fetching them one by one. Fig 17 elaborates the Voiceflow workflow when there is only one sender.

4.4.5 *Exiting the Voiceflow.*

Alexa currently faces some issues with the "stop" command as it is a specific word that can prematurely terminate the entire flow. Currently, we have Alexa guide users during the conversation to use alternative statements like “end" or “That's all" that convey their intent to end the interaction. This way, Alexa can correctly recognize the user's desire to conclude the conversation without any errors.

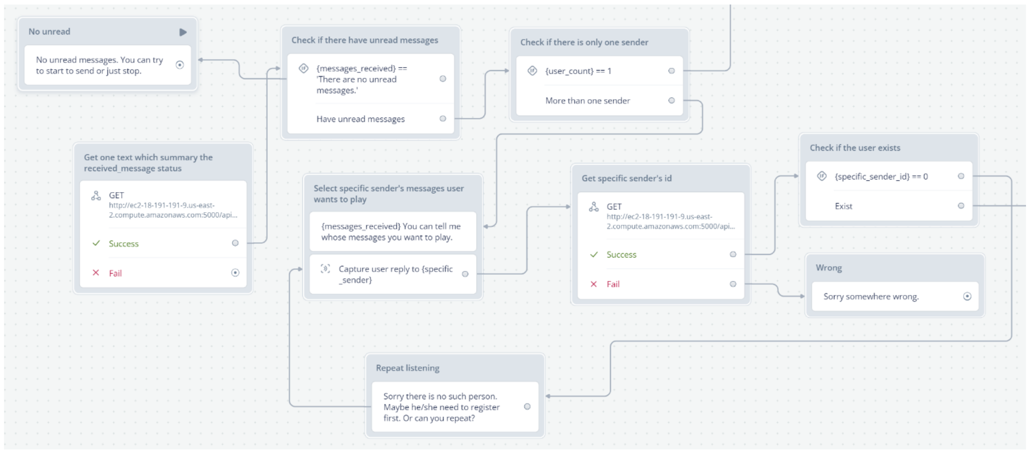

No unread
No unread messages. You can try to start to send or just stop.
Check if there have unread messages
{messages_received} == 'There are no unread messages.'
Have unread messages
Check if there is only one sender
{user_count} == 1
More than one sender
Get one text which summary the received_message status
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Select specific sender's messages user wants to play
{messages_received} You can tell me whose messages you want to play.
Capture user reply to {specific_sender}
Get specific sender's id
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Check if the user exists
{specific_sender_id} == 0
Exist
Wrong
Sorry somewhere wrong.
Repeat listening
Sorry there is no such person. Maybe he/she need to register first. Or can you repeat?

Fig. 15. Received Messages Check Voiceflow

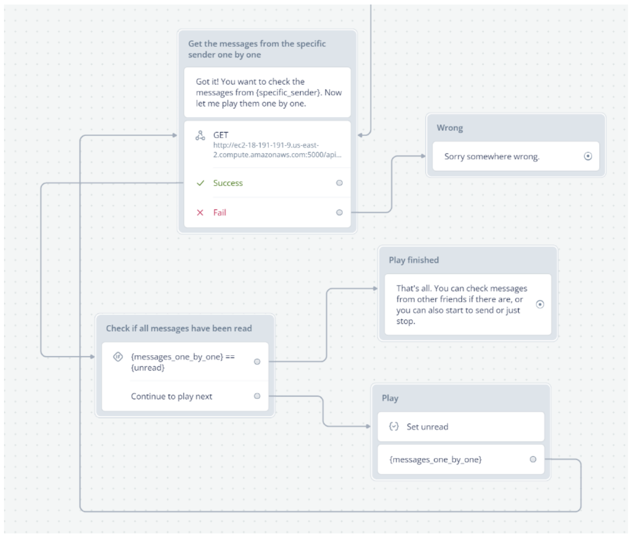

Get the messages from the specific sender one by one
Got it! You want to check the messages from {specific_sender}. Now let me play them one by one.
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Wrong
Sorry somewhere wrong.
Play finished
That's all. You can check messages from other friends if there are, or you can also start to send or just stop.
Check if all messages have been read
{messages_one_by_one} == {unread}
Continue to play next
Play
Set unread
{messages_one_by_one}

Fig. 16. Play Messages One by One Voiceflow

5 TESTING AND EVALUATION

We followed a soft test-driven development cycle, where instead of using a unit-testing framework due to time and capacity constraints, we manually executed the test cases. In week 10, we performed end-to-end (E2E) testing and monkey testing of the entire system.

We tested the WearOS application for the following criteria:

- **Responsiveness**: We used Android Studio's Device Manager to create virtual devices in a combination of small and large sizes with round and rectangular shapes. The results show that our application is responsive

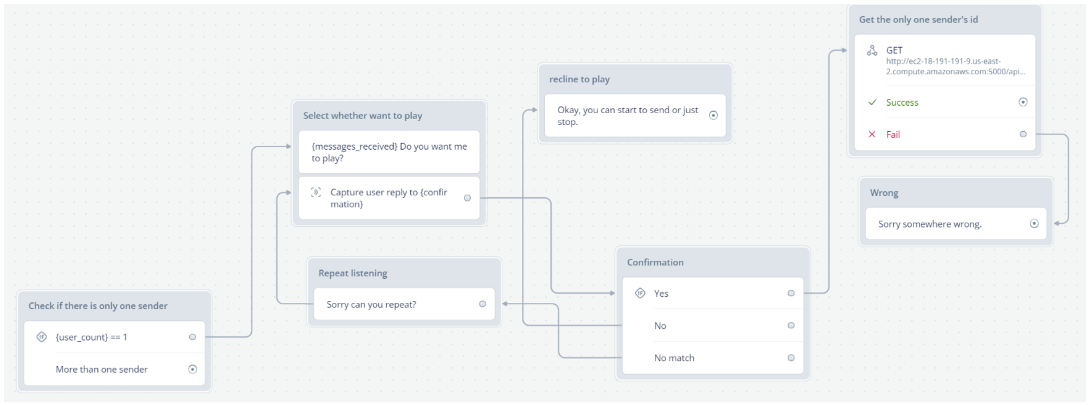

Check if there is only one sender
{user_count} == 1
More than one sender
Select whether want to play
{messages_received} Do you want me to play?
Capture user reply to {confirmation}
Repeat listening
Sorry can you repeat?
recline to play
Okay, you can start to send or just stop.
Confirmation
Yes
No
No match
Get the only one sender's id
GET
http://ec2-18-191-191-9.us-east-2.compute.amazonaws.com:5000/api...
Success
Fail
Wrong
Sorry somewhere wrong.

Fig. 17. Play Messages When There is Only One Sender Voiceflow

and functional for varying screen sizes and shapes. All our API calls are asynchronous and do not block the UI thread, allowing for a fast and responsive user experience.

- **Accessibility**: We verified that our UI elements are compliant with TalkBack for enabling app navigation without visual feedback.
- **Braille Keyboard functionality**: Since we did not have access to a real low-vision user during our final testing, we performed blindfolded user testing within our team to evaluate the functional performance of the keyboard. We used two differently-sized smartwatches (40cm and 44cm), and two of our team members participated in the user trial. Our first member was involved in the design of the keyboard and therefore had developed some muscle memory of the layout of the dots of the Braille cell on the watch screen. Hence, we refer to this user as User A, an experienced user. The other member is an inexperienced user, and we refer to this user as User B.
  Write: For the Write test, each user was asked to enter all 26 characters of the English alphabet using the keyboard. Three trials were performed, in which twice the characters were entered in sequence as they appear in the alphabet, and once in mixed order. We evaluate the accuracy by observing the number of keys correctly pressed and the edit distance between the actual message and the written message.
  Read: For the Read test, each user was asked to read all 26 characters of the English alphabet using the keyboard. Three trials were performed, in which twice the characters were read in sequence as they appear in the alphabet, and once in mixed order. We evaluate the accuracy by observing the number of keys correctly pressed and the edit distance between the actual message and the read message.
  Results: The results of the tests are averaged and reported in Table 4. We found that for Write operation, User A performs better than User B on average, whereas accuracy is similar for both users on the Read test.
  Observation: One of the primary sources of errors occurred when the blindfolded users incorrectly pressed certain Braille dots or mistakenly selected a nearby dot. For instance, the character ‘q’ was consistently misinterpreted when users missed pressing dot 3 accurately, resulting in the unintended entry of the character ‘g.’ Both ‘q’ and ‘g’ share an identical Braille representation, with the sole distinction being the absence of dot 3 in the character ’g.’ This recurring error emphasizes the importance of precise dot selection during Braille input to avoid unintended character entries.
- **Voice Recognition Accuracy:** For the voice assistant, we tried testing the voiceflow using different kinds of messages which included numbers and also phrases that could be easily misinterpreted to check how accurate the voice recognition was. We found the accuracy to be very high picking up most of the users’

Table 4. Results of Braille Keyboard test

User	Write test	Read test
A	23/26	25/26
B	20/26	24/26

input. However, there were some cases where it interpreted the users' input differently. To handle this cases, we introduced a second layer of check where the user makes sure that their message is correct and give them the option to retry if needed.

6 COLLABORATION

We divided the work among our team members based on interest and experience. Rohith worked primarily on the development of the back-end web server, Antariksha and Juyoung worked primarily on the development of the WearOS application, and Santosh and Shujie worked primarily on the development of the Alexa skills. In addition, the members also cross-collaborated on some of the design and testing of components other than theirs. Everyone participated in the sprint planning, scrum meetings, and sprint retrospection. Following Agile scrum methodology, Rohith and Antariksha alternated as scrum masters every week. We list below the weekly progress, challenges faced, and the decisions taken to resolve them.

- Week 7
  - **Project Management**
    - Rohith
      1. Setup GitHub Repository and Project
  - **Server and back-end**
    - Rohith
      1. Setup basic template for the nodejs server that can be used to build the APIs
      2. Research on what system to use for deployment that would be easy for collaborative development
  - **WearOS application**
    - Antariksha
      1. Defined the package structure for the watch application.
      2. Developed the HTTP Client and APIs, the data model, and the views for the Standard Messages (non-user-specific).
    - Juyoung
      1. Explored and considered the design for an accessible Braille keyboard
    - Challenges
      1. Compared the networking library tradeoffs to select Retrofit with OkHttp instead of Volley after considering the overall ease of use with fewer lines of code and full support of converting JSON objects for POST requests. (See more about Volley here https://google.github.io/volley/)
  - **Alexa**
    - Santosh and Shujie
      1. Compared using voiceflow instead of AWS Lambda to build the Alexa skill from scratch considering tradeoffs like implementation effort, extensibility and maintainability
- Week 8
  - **Server and back-end**
    - Rohith

    (1) Create Routes, Controllers, Services, and Models for user create, read, update and delete
    (2) Create Routes, Controllers, Services, and Models for message create and read
    (3) Setup EC2 Linux Machine with nodejs and mongodb
    (4) Deploy the basic application on EC2
    (5) Create collection with API's in postman for testing
    (6) Deploy and test the user and messages API on EC2
    (7) PM2 Ecosystem Config file for deployment
    * Challenges
    (1) Installing Node.js and MongoDB on Amazon Linux posed a challenge, but we successfully addressed it by referring to the Amazon Linux documentation.
    (2) Need to create APIs to handle alexa requirements since javascript support is only available in the licensed version of voiceflow, so there is minimal coding that can be done there.
  - **WearOS application**
    * Antariksha
    (1) Fixed the bug to handle longer messages using a scrollbar.
    (2) Extended the views and Client APIs to show list of users to send or receive user-specific messages.
    (3) Implemented a utility method to persist the user id in the app-specific storage.
    * Juyoung
    (1) Created a simple interface for composing and sending messages using braille characters.
    (2) Created the basic interface to read messages through the braille keyboard.
    (3) Implemented one-to-one mapping feature to translate braille characters (6-digit binary strings) into corresponding letters (lowercase/uppercase alphabet and numbers)
  - **Alexa**
    * Santosh and Shujie
    (1) Designed the logic and basic communication flow for sending and receiving messages to/from 1 user
    (2) Integrated flow with the backend server using the new API endpoints
    (3) Came up with a new flow for sending/receiving message from multiple users and identified a few more endpoints that needs to be set up for Voiceflow to work around Voiceflow free tier
    (4) Tested Alexa skill on AWS Alexa Developer Console to make sure the integration is working fine
    * Challenges
    (1) Fragments could not be handled easily with the standard WearOS back navigation (swipe-to-dismiss), so chose to use activities instead.
- Week 9
  - **Server and back-end**
    * Rohith
    (1) API to get user ID from name
    (2) Messages API - Update read status and return only unread messages
    (3) Messages - Get number of messages from users with name
    (4) Messages - Update list of messages
        · Input Query: userId, fromUser (Optional, if not given messages from all users will be sent), consolidated: true/false (Optional, default=true)
        · Returns array of messages info or single text message
    (5) Fix bugs
        · make username case insensitive in query

- · The "Get Message" function wrongly marks extra messages as read. For example, if User1 has unread messages from User2 and User3, querying for messages from User2 shows the unread messages, but subsequent requests for messages from User3 don't return any messages.
    - ∗ Challenges
    - (1) Updating the APIs to handle requirements for Voiceflow since it has minimal support for javascript like looping through arrays, parsing json string.
  - **WearOS application**
    - ∗ Antariksha
    - (1) Fixed the previous and next buttons enabling logic when there are no messages in that respective direction.
    - (2) Added the dark material library theme and changed the buttons to use icons to make more screen space available for viewing the message.
    - (3) Collaborated on the design and testing of the first Braille keyboard prototype.
    - ∗ Juyoung
    - (1) Improved the Braille keyboard interface by reducing the number of buttons from 8 to 6 and ensuring they occupy the entire screen, eliminating any empty space.
    - (2) Implemented mapping feature to translate letters (alphabet - lowercase/uppercase, number, basic punctuation marks) to braille characters (6-digit binary strings)
    - (3) Introduced user-specific feature enabling message senders to choose a message receiver by inputting the recipient's name in a Braille pattern, so that message receivers can identify the sender for each message, presented in the format "sender name: text message."
    - (4) Integrated with an API, allowing messages to be sent to the server through a POST request.
    - ∗ Challenges
    - (1) Experimented with overriding swipe gestures for the second iteration of the keyboard, and decided against using it due to the inconsistent UX and the unpredictable swipe behaviour encountered for complete and partial overriding of swipe gestures.
  - **Alexa**
    - ∗ Santosh and Shujie
    - (1) Integrated existing voice flow with the new endpoints
      - · Fetching user IDs from name
      - · Using the loop flow to fetch messages one by one
    - (2) Implemented option to allow users to know how many messages they have received from different uses and choose to play a specific user's messages
    - (3) Researched existing ways to communicate with watch from the server or from the voice assistant using either sockets or deep links
- Week 10 and Finals Week
  - **Miscellaneous**
    - ∗ The team collaborated on testing various components of the system through message exchanges across different modalities, including Text, Braille, and Voice.
    - ∗ Juyoung and Rohith served as blindfolded users, actively testing the Braille functionality. Antariksha acted as the moderator, facilitating the testing process.
    - ∗ Santhosh and Shujie conducted detailed testing of the voice assistant (Alexa).
    - ∗ Rohith and Antariksha collaborated on creating detailed System Architecture and Design Diagrams.
    - ∗ All team members contributed to the Final Presentation and Report.
  - **Server and back-end**
    - ∗ Rohith

(1) Include the number of users in messages metrics API
(2) Test user (create, read, update, delete) and messages (create and read) APIs

∗ Challenges

(1) Integrating roles and implementing API access authorization based on user roles is a critical enhancement for our APIs. Nevertheless, introducing these changes at this stage may impede the current momentum in building the core system. For information on how security can be integrated into the system, please refer to Section 7.

– **WearOS application**

∗ Antariksha

(1) Collaborated on the design and testing of the second Braille keyboard prototype.
(2) Updated the UI elements with textual information to be compliant with screen-readers.
(3) Added documentation, and performed end-to-end testing of the system.

∗ Juyoung

(1) Integrated with an API, allowing messages to be received from the server through a GET request.
(2) Transitioned from the conventional button clicking method to long-press actions.
(3) Conducted blindfolded user testing to assess the functional performance of the keyboard.

– **Alexa**

∗ Santosh and Shujie

(1) Completed improving the logic flow for a more appropriate response when there are unread messages from only one user
(2) Revised the closing statements to better align with various scenarios where users expect to end the conversation
(3) Refined user flow to simplify user experience allowing them to perform their desired actions efficiently

7 CONCLUSION AND FUTURE WORK

The BridgeText project has made significant strides in addressing communication challenges for individuals with sensory impairments. Our innovative approach, integrating Wear OS and Amazon Echo Dot technologies, provides a versatile and inclusive communication platform. The incorporation of Braille text input, haptic feedback, and voice commands on smartwatches contributes to a more adaptable user experience.

In conclusion, the system showcases the potential to improve accessibility for users with different sensory impairments, fostering inclusive digital communication. Future work could focus on enhancing security and privacy through robust authentication mechanisms and server-side security measures. Implementing caching mechanisms can improve performance, while a notification system can enhance user engagement. Exploring alternative Braille keyboard placements, such as on wristbands, presents an intriguing avenue for faster message reading. With more time, resources, and capacity, several improvements and extensions can be made in the future.

- **Security and Privacy:** An authentication mechanism can be implemented to secure the communication between the server and the client. A simple username and password-based authentication such as HTTP Digest access can confirm the identity of a user before sending sensitive information in the message by one-way hashing the user credentials using a cryptographic hash function such as MD5 or SHA-256. **Server-Side Security Measures:** For the Node.js server, we can also configure firewalls to restrict unauthorized access, implement intrusion detection systems, and conduct regular security audits. Additionally, we can ensure secure data transmission by using HTTPS with SSL/TLS certificates which can significantly boost security, although this requires the overhead of generating and signing certificates using a tool like OpenSSL. **Deployment on Amazon EC2:** Enhance privacy and data protection by implementing proper access controls through AWS Identity and Access Management (IAM). Enable monitoring using AWS CloudWatch

for real-time insights into server performance and potential security threats. **WearOS** Since our WearOS application uses the HTTP protocol with OkHttp, both the above types of authentication can be easily integrated by adding a new public API that returns a version of the OkHttp client modified to include the respective context. This context would use the Authenticator interface for Digest authentication, whereas certificate-based authentication can be implemented with an SSL/TLS context, and using the KeyStore and KeyManagerFactory classes to store and manage the certificates.
- **Caching:** Caching of requests based on the HTTP headers can improve the performance of the application, as requests would only be made once the previous data expires based on the caching mechanism. In the case of the WearOS application, the OkHttp client can be modified to include a cache along with online and offline interceptors to add Cache-Control headers.
- **Notification:** Implementing a notification mechanism can enhance user engagement. In the Node.js server, we can employ the Firebase Cloud Messaging (FCM) service to send push notifications to the WearOS app. We can integrate the 'firebase-admin' SDK in the Node.js server to interact with FCM. Configure FCM to generate unique tokens for each device and use them to target specific WearOS devices with notifications. In the WearOS app, we can utilize the Firebase Cloud Messaging (FCM) client library to receive and handle push notifications, with a background service that listens for incoming FCM messages and triggers notifications on the device accordingly. We should also ensure proper handling of notification preferences and permissions within the app to offer users control over their notification settings.
- **Alternative Braille keyboard:** A significantly faster reading of messages could be achieved by placing the Braille keyboard on the smartwatch's wristbands, which would allow for a much larger space for touch-based interaction. However, this would require a bigger change in the overall architecture, along with the research, development, and integration of hardware similar to refreshable Braille displays in the wristband. The WearOS application would also need to be modified to communicate with this hardware, to update the messages as it is read.
- **Introduction to Lambda Enhancement:** In the future, we can enhance the functionality of Alexa on Lambda by reducing its reliance on API interfaces. This would address straightforward conditional issues and eliminate redundancy in interface design.
- **Interaction Logic Optimization:** To improve user-friendliness, we could optimize the interaction logic based on the existing foundation.
- **Tolerant Mechanisms Improvement:** Additional fault-tolerant mechanisms could be implemented to make the system more robust, minimizing unexpected errors.

With continuous development and consideration of these future directions, BridgeText has the potential to revolutionize communication accessibility for individuals facing sensory challenges.

8 ACKNOWLEDGMENTS

We extend our gratitude to **Prof. Nadir Weibel** for guiding us and sharing invaluable insights into technologies vital for our project. We also appreciate the ongoing support and feedback from **Chen Chen**. Special thanks to **Mary Tregoning** for providing insightful user feedback, which has significantly influenced and shaped the trajectory of our project.

REFERENCES

[1] 2023. Alexa unveils new speech recognition, text-to-speech technologies. https://www.amazon.science/blog/alexa-unveils-new-speech-recognition-text-to-speech-technologies.

[2] Mrim Alnfiai and Srinivas Sampalli. 2017. BrailleEnter: A Touch Screen Braille Text Entry Method for the Blind. *Procedia Computer Science* 109 (2017), 257–264. https://doi.org/10.1016/j.procs.2017.05.349 8th International Conference on Ambient Systems, Networks

and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.

[3] Tanay Choudhary, Saurabh Kulkarni, and Pradyumna Reddy. 2015. A Braille-based mobile communication and translation glove for deaf-blind people. In *2015 International Conference on Pervasive Computing (ICPC)*. 1–4. https://doi.org/10.1109/PERVASIVE.2015.7087033

[4] Chandrika Jayant, Christine Acuario, William Johnson, Janet Hollier, and Richard Ladner. 2010. V-Braille: Haptic Braille Perception Using a Touch-Screen and Vibration on Mobile Phones. In *Proceedings of the 12th International ACM SIGACCESS Conference on Computers and Accessibility* (Orlando, Florida, USA) *(ASSETS '10)*. Association for Computing Machinery, New York, NY, USA, 295–296. https://doi.org/10.1145/1878803.1878878

[5] Brian Kemler. 2020. A new keyboard for typing braille on Android. https://blog.google/products/android/braille-keyboard/.

[6] Veton Këpuska and Gamal Bohouta. 2018. Next-generation of virtual personal assistants (Microsoft Cortana, Apple Siri, Amazon Alexa and Google Home). In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. 99–103. https://doi.org/10.1109/CCWC.2018.8301638

[7] Kelly Mack, Emma McDonnell, Dhruv Jain, Lucy Lu Wang, Jon E. Froehlich, and Leah Findlater. 2021. What Do We Mean by "Accessibility Research"? A Literature Survey of Accessibility Papers in CHI and ASSETS from 1994 to 2019. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (<conf-loc>, <city>Yokohama</city>, <country>Japan</country>, </conf-loc>) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 371, 18 pages. https://doi.org/10.1145/3411764.3445412

[8] Elke Mattheiss, Georg Regal, Johann Schrammel, Markus Garschall, and Manfred Tscheligi. 2015. Edgebraille: Braille-based text input for touch devices. https://doi.org/10.1108/JAT-10-2014-0028

[9] Balaji Muthazhagan, Mingyi Li, Chung-Chi Chao, and Shikha Dixit. 2022. Elle Braille Keyboard Application. https://balajimuthazhagan.com/elle/.

[10] Hugo Nicolau, Kyle Montague, Tiago Guerreiro, André Rodrigues, and Vicki L. Hanson. 2015. HoliBraille: Multipoint Vibrotactile Feedback on Mobile Devices. In *Proceedings of the 12th International Web for All Conference* (Florence, Italy) *(W4A '15)*. Association for Computing Machinery, New York, NY, USA, Article 30, 4 pages. https://doi.org/10.1145/2745555.2746643

[11] João Oliveira, Tiago Guerreiro, Hugo Nicolau, Joaquim Jorge, and Daniel Gonçalves. 2011. BrailleType: Unleashing Braille over Touch Screen Mobile Phones. In *Human-Computer Interaction – INTERACT 2011*, Pedro Campos, Nicholas Graham, Joaquim Jorge, Nuno Nunes, Philippe Palanque, and Marco Winckler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 100–107.

[12] Rohit Rastogi, Shashank Mittal, and Sajan Agarwal. 2015. A novel approach for communication among Blind, Deaf and Dumb people. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 605–610.

[13] Mario Romero, Brian Frey, Caleb Southern, and Gregory D. Abowd. 2011. BrailleTouch: Designing a Mobile Eyes-Free Soft Keyboard. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services* (Stockholm, Sweden) *(MobileHCI '11)*. Association for Computing Machinery, New York, NY, USA, 707–709. https://doi.org/10.1145/2037373.2037491

[14] Kevin M. Storer, Tejinder K. Judge, and Stacy M. Branham. 2020. "All in the Same Boat": Tradeoffs of Voice Assistant Ownership for Mixed-Visual-Ability Families. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376225