Enhancing Communication Features In Yioop

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Aditya Jagdishkumar Prajapati

December 2025

The Designated Project Committee Approves the Project Titled

Enhancing Communication Features In Yioop

by

Aditya Jagdishkumar Prajapati

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2025

Dr. Chris Pollett	Department of Computer Science
Dr. Navrati Saxena	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

Enhancing Communication Features In Yioop

by Aditya Jagdishkumar Prajapati

Modern communication applications offer a variety of features to improve user experience. Interacting with other users keeps the users engaged with the application, so most modern applications offer some sort of messaging or communication features. Because of this, most users now have a mental model of the basic features that they expect from any communication system. Yioop, an open-source web platform, currently offers basic chat functionality but lacks some of these, and this project aims at bridging the gap between what is offered versus expected.

This project augments Yioop's chat system by integrating features such as audio messages, speech-to-text summary, direct translation and summary of messages and "rich text" styling using markdowns. These features improve accessibility and make the user experience better.

The project was divided into two phases. The first part focused on understanding the coding structure and standards and getting familiar with the User Interface (UI) of the platform. This culminated into the main phase, which was around AI-enhanced features like text translation, transcription, and summarization. We also added helper scripts to populate test data into Yioop so that it can be used as a reference by anyone who wants to independently benchmark the work done in this project.

Keywords: Yioop, Chat Enhancement, Speech-to-Text, Markdown Parsing, Summarization, Accessibility in Communication Systems

ACKNOWLEDGMENTS

I want to take a moment to thank my mentor, Dr. Christopher Pollett, for his time and patience throughout my project. This project would not have been possible without the prior work that he and all his students have completed before me.

I would also like to thank the department of Computer Science at SJSU for their support.

Also a huge shout out to my family and friends for their unwavering support.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Yioop [1] is an open-source platform for search and collaboration.Users can share documents and images, but text is still the main mode of communication. With tools like Whatsapp, Discord, and Slack being hugely popular, people expect certain default services like voice messages, translations, and AI driven formatting and summarization.

This project adds these missing pieces to Yioop. The aim is to make communication easier especially for users who prefer speaking, have different language requirements or need a quick summary of large amount of texts. This makes Yioop more useful by including different functionalities and also not compromising on privacy and data control.

1.1 Problem Statement

Before the research conducted here, Yioop showed a lot of weaknesses in its communication systems. Although users were allowed to add documents and images, the voice message delivery process was quite a complex one. It required the user to record the audio as an independent file, save it and send it in a file format. As a result, the voice message was not as spontaneous as is being provided by modern messaging systems.

Additionally, while users can change their locale on the platform, it only translates locale specific texts in the interface, i.e., Sender's messages were not translated to their respective languages. Therefore communicating with users across different languages meant sending messages in one language, receiving responses in a different language.

Lastly, access to edit the wiki services of Yioop was provided on groups levels with no distinction made between members who can edit the content and those who can only view it. Moreover, the MediaWiki format used by the Yioop wiki is quite technical and most people find it difficult to navigate through the format.

1.2 Project Goals

A key focus for this project was enhancing the functionality of Yioop to better match the expectations of its users than it does today. The two main goals were to improve the user experience (reduce the user's cognitive load) and to allow users to more easily interact with each other via the platform.

One of the first new functionalities to be added to the platform was the ability to send and receive voice messages. This allowed users to both create voice messages to send to others, and to listen to messages that they had received. In order to further enhance this new feature and to provide additional usability, the system was designed to generate a transcript of all messages that are sent via voice message. Users could then view the content of their messages rather than having to play back the recording, which would be very beneficial for those working or living in a quiet environment, as well as for accessibility.

Another primary objective of the project was to enable automated translation between users of different native languages. When a user selects a specific contact and prefers to have all messages translated into a certain language, that choice will be reflected in the conversation with that individual. Therefore, a user may select to have messages from one contact translated into English and messages from another contact translated into Spanish simultaneously, but the way that the originator of each message sees the conversation will remain unchanged.

Finally, another component of the project was the use of large language models to provide thread summaries to users when they enter a lengthy conversation. The project also included an optional feature called Markdown, which the users could avail to format their messages according to other popular messaging styles.

1.3 Significance of the Project

This project shows how legacy open source projects can still integrate with the latest technologies, and provide better User Experience by adding AI-powered features. This work introduces a self-contained infrastructure for AI inference instead of just offloading responsibility to a third party service.

Developers often focus on releasing new features but it requires proper experience to create features that are accessible to and cater to a wide range of users.

1.4 Organization of the Report

The rest of this report is organized as follows:

- **Chapter 2** outlines Yioop's architecture and reviews related work
- **Chapter 3** discusses the core ideas kept in mind while designing the solution for all problems
- **Chapter 4** details the system design and architecture of Yioop as a whole and the important components that we touched
- **Chapter 5** details the actual implementation details
- **Chapter 6** discusses approaches on how to benchmark the work and presents the findings
- **Chapter 7** summarizes the project's main contributions and further developments that can be made using it

CHAPTER 2

Background and Related Work

In this chapter we provide some background information about the history of messaging systems, and a brief description of Yioop. We also describe how the history of computer-mediated communication (CMC) systems influenced the design decisions of our project.

2.1 Evolution of chat systems

Digital communication has evolved over the last four decades from text-based communication to rich multimedia ecosystems. We will now take a brief look at this since it sets the expectations from a modern system and has guided the objectives of this project.

2.1.1 The early days: IRC

Internet Relay Chat (IRC) was developed by Jarkko Oikarinen in 1988 [2] and represents the first major Internet communication technology and set many of the terms and concepts for modern chat systems, including channels, nick names, and server relay networks.

IRC utilizes a tree structured network in which servers connect to each other in a cycle-free topology. Once a message enters a server, it will send the message to all other servers hosting members of the same channel as the original sender. Although this structure allowed IRC to remain light-weight and decentralized, it had significant reliability problems. Two of the most common reliability problems with IRC were netsplits, and message loss when the recipient was offline. Netsplits occur when a link between two servers breaks, splitting the network into two separate trees. Due to the fact that the IRC network does not contain redundant links, once the link breaks,

there is no path for the disconnected part of the network to rejoin. Additionally, IRC did not preserve messages - a message would disappear after the recipient went offline.

2.1.2 XML-Based Protocols

In the late 1990's the Extensible Messaging and Presence Protocol (XMPP) presented a better replacement for IRC using XML streams. Unlike IRC's text-based protocol, XMPP used a long-lived XML stream from the client to the server[3].

It powered the early infrastructure of major platforms like Google Talk and WhatsApp, but the overhead of parsing text-heavy XML drained battery life on early smartphones that had limited resources, which led to mobile-specific applications to move away from it [4].

2.1.3 Websockets

In the 2000s, communication moved to the web-browser and developers faced challenges due to the limitations of HTTP, which was ill-suited for real time communication. The initial solutions for real-time web communication were polling, where the client would send an AJAX request every few seconds to see if there were any new messages, and long-polling, where the server would maintain the connection open until either new data arrived, or the connection timed out. Long-polling decreased the amount of requests the client needed to make, but still required the overhead of multiple TCP/IP handshakes.

To solve these issues, WebSockets were standardized in 2011 [5] which allowed for a single, long-lived TCP connection with full-duplex communication.

2.2 From text to multiple inputs

As internet speed improved and devices became better, communication systems naturally evolved. Features like voice messages and video calls became possible, and later became common. Most modern platforms now support group chats, threads, and media sharing as standard features. Below we explore the major milestones that led to this shift.

2.2.1 Introduction of Voice messages

Due to bandwidth constraints, real-time communication systems were forced to be text-based like we have discussed so far. Text messages helped people stay connected but lacked the nonverbal cues such as tone, pitch and pacing important for communication.

2.2.2 Media Richness Theory (MRT)

The Media Richness Theory was proposed in 1986 by Daft and Lengel [6], and suggests that some communication channels convey ambiguous information better than others because they provide more cues, feedback, or clarity. According to MRT, text is suitable for simple, unambiguous tasks, but complex human coordination often fails without the "richness" of voice or visual cues.

A 2025 study on media effects found that audio messages make people feel a much stronger sense of “social presence.” In other words, audio helps create the feeling that the other person is real and present.[7]. This study also indicates that greater social presence generated by audio messaging results in a greater degree of trust among communicators than text alone.

2.2.3 Enhancing the UX for Voice Messaging

Voice messaging is becoming increasingly common today through the means of "voice notes". A voice note offers users both the advantage of a voice call (the depth of emotion conveyed), as well as the advantages of a text message (quick, easy to use). Research conducted by Aslan et al., (2025) titled “Speejis: Enhancing User Experience of Mobile Voice Messaging” [8], identifies a significant difference between Computer-Mediated Communication (CMC): While text messaging has been enhanced with richer visual enhancements (such as emoji’s and stickers) to express emotions; voice messaging remains an unenhanced "black box" of audio information. Aslan et al., (2025), identify that their participants preferred to read voice messages when they were accompanied by visual enhancements, rather than reading them without any visual enhancements.

The participants reported that normal voice messages were more difficult to follow and less engaging than voice messages which had visual enhancements. It was evident from this study that while voice can capture additional emotional cues; it is lacking in the "glanceability" of text. Therefore, developing features that provide additional visual enhancements to audio content provides a compelling argument to include audio transcription in conjunction with voice recording on Yioop.

2.3 Industry adoption of Markdown

Markdown is an extremely simple lightweight markup language, designed to be readable in its source form [9].The industry-wide adoption of Markdown is primarily a result of the “Docs as Code” philosophy[10], where documentation is treated with the same discipline as coding.

In their paper “Large-Scale Collaborative Writing: Technical Challenges and Rec-

ommendations", Hofbauer et al. (2023) also recommend lightweight markup languages such as Markdown or AsciiDoc [11] for collaborative environments. Their argument is that separating content from layout enables far better scalability, adaptability and version control in larger teams.

2.3.1 Why modern platforms use Markdown

Leading platforms like Slack, Discord, and GitHub have standardized on Markdown for chat and documentation. The advantages of Markdown include:

- Interoperability: Unlike HTML or proprietary Word formats, Markdown is universally parseable.
- Lower barrier to entry: It is simple to learn and write.

2.4 Using Generative AI: Reducing Information Overload

In large groups, the volume of text messages can easily exceed hundreds or thousands per day, and the duration of recorded audio can extend beyond hours. In such cases, it is really difficult for users to keep up with messages and very easy to miss an important note due to the information overload.

2.4.1 Cognitive Load Theory in CMC

According to the Cognitive Load Theory, human working memory is limited. When users are forced to go through chat logs or listen to long audio files to find relevant facts, they experience high cognitive load, which reduces productivity and decision-making ability ([12]).

2.4.2 Using AI to reduce this load

Generative AI (often Large Language Models) offer a means to reduce this cognitive load on the users in multiple ways. In this project, we use it to generate thread summaries so that users can quickly catch up on threads, and for translation, so that language barriers are removed.

2.5 Yioop Overview

Yioop is an open source search engine/web application platform developed with PHP. It provides distributed crawlers, collaborative workspaces, and community forums. Yioop is based on the MVC (Model-View-Controller) architecture. This provides a modular framework to add features, such as wikis, forums, chat, and AI (this projects scope) to a single application without breaking modularity.

Several architectural aspects of Yioop influenced how the current project was built. First, Yioop uses the Model View Controller (MVC) architecture with component-based controller logic. Since controller logic is broken down into components, all AI components in the current project used pre-existing social and API components as opposed to being separate structures. Second, the current project follows Yioop's security design and all the new endpoints for use standard security practices such as CSRF tokens and restrictive Content Security Headers.

In addition, the project uses many of Yioop's larger frameworks to provide a scalable and user friendly experience. To support the large amount of internationalization available within the Yioop platform, all new features support internationalization to all the languages supported by the platform. Additionally, the transcription of audio files uses the same type of background job daemons provided by Yioop, which are capable of processing media files and crawlers. Changes to data storage also follow

existing patterns - database changes were made using a version controlled migration.

CHAPTER 3

Design Philosophy and Motivation

This chapter outlines the reasons behind the new features added to Yioop during this project. The objective was to increase system usability, accessibility, and better align with modern platforms. The motivation behind this work is split into four key areas: accessibility, usability, privacy, and cost-efficiency.

3.1 Accessibility

Although the primary method of user interaction with Yioop is via text (as shown above) there are certain user groups that will face barriers based on this text centric design. The first step in addressing this issue was to identify three user groups that would be beneficial to improve upon the current platform for: users with hearing disabilities that require a transcript to understand a voice message; users with either visual/motor disabilities that use voice commands to enter data into the platform; users that speak multiple languages and prefer to read information written in their native language. As such, the following steps were taken to enhance accessibility to these user groups.

A native audio recorder was added as part of the message sender to capture voice input, and speech to text transcriptions were added. Additionally, all interface changes made to the platform were localized into over 20 languages. Every new label, button, and string that were added to the locale file to enable internationalization/localization. These enhancements enabled the platform to be used by a greater number of individuals, many of which may have had difficulty accessing the platform prior due to the text-centric nature of the interface.

3.2 Improving Human-Computer Interaction

All new features must be intuitive to use and not negatively affect the User Experience. Yioop has a rich feature set for collaboration, and the goal of this project was to enhance those features in ways that do not disrupt the current flow. The project does this by:

- Placing all UI components in positions that fit with the existing components, following the same styles
- Only displaying the new UI elements when the necessary conditions were met. If for example the LLM API URL was not configured, the auto-translate and thread summarization buttons were not shown.
- Providing the capability to translate native messages to the user's preferred locale, which is stored in the user settings. Many non-native speakers typically think in their native language and translate their thoughts into the language of communication, which results in lost context and nuance. By providing multiple input methods (text, audio, and translated text), the system is made more usable and comfortable to use.

By supporting multiple input styles (audio, markdown, translated text), the system becomes more comfortable to use.

3.3 Privacy and Security

Another one of the main goals of this project was to avoid sending user data to third party vendors. Since Yioop is an open source project, it can be deployed on-prem in universities, hospitals, or any air-gapped environment (like financial institutions) where sensitive data cannot leave the network. To support these environments, all added features were built with local processing and pluggable architecture in mind:

- **Local Inference:** Transcription, translation, and summarization can utilize self-hosted open-source models, such as Whisper or Aya [13].
- **Vendor Agnostic Backend:** The backend architecture enables the administrator to swap inference providers or turn off network dependent features entirely.
- **Configurable Privacy:** Data remains within the system by default unless an administrator explicitly configures a cloud-based provider.

3.4 Cost Control and Abuse Prevention

LLMs and APIs can cost money quickly; especially since many third-party vendors such as Anthropic and Open AI bill customers per input token and per output token generated. For open source projects, this presents a problem: a single malicious user can increase costs by spamming translation/summarization features.

To address this:

- Translation & summarization features are turned on demand instead of automatic.
- All tasks are run through a locally deployed LLM.

3.5 Structured Prompts for Reliability

A known issue with using LLMs is controlling their output, or reproducibility. The same prompt will generate a different responses every time. Ensuring that the LLMs respond correctly consistently requires figuring out what prompts to use and how to extract information from them. This system utilizes prompt scaffolding. It includes adding tags around the inputs and outputs for the LLM to ensure that the LLM. responds in a predictable format that can be easily parsed. Additionally,

summaries and translations are created in a clean format that is easy to parse and the frontend or php proxy will not break from receiving unexpected tokens.

3.6 Summary

The motivation behind this work is simple: introduce new features that other modern communication platforms have and enhance the features that Yioop already has. These features were chosen because they solve real problems for users: accessibility, ease of use, and communication friction while keeping cost risk and data privacy in mind.

CHAPTER 4

Project Enhancements

The Chapter provides technological design and engineering decisions of integrating the transcription, translation, and summary functions in Yioop. It will also mention what you will already know in order to follow along besides specifics of implementation in the next chapter. The added functionality was added to the Yioop without the need to change the internal structure of Yioop and the existing logic of the data flows.

4.1 Design Principles and Goals

We had a number of goals that we applied when designing our system:

- **Local Inference for privacy:** The features of the AI incorporate local reasoning. They use local servers (Whisper and LLMs through LM Studio) to inquire and do not need to connect to remote cloud APIs.
- **Plugging-in:** Functions are implemented as modules that can send and receive messages to each other across HTTP interfaces. Yioop has not been changed in its core. In this sense, any other person who would wish to introduce new features to Yioop in future can just write his or her own Python and Go server to do the same thing in a different way. As an example, they may use Vosk [14] to do transcriptions instead of Whisper [15].
- **Reuse of Existing Infrastructure:** The implementation makes use of the majority of the same job scheduling, API control routing, CSRF security system, and internationalization support that Yioop already had in place.
- **Minimal invasiveness:** The treatment of transcripts, translations and summaries as auxiliary resources was adopted. There have not been any significant changes in database schema.

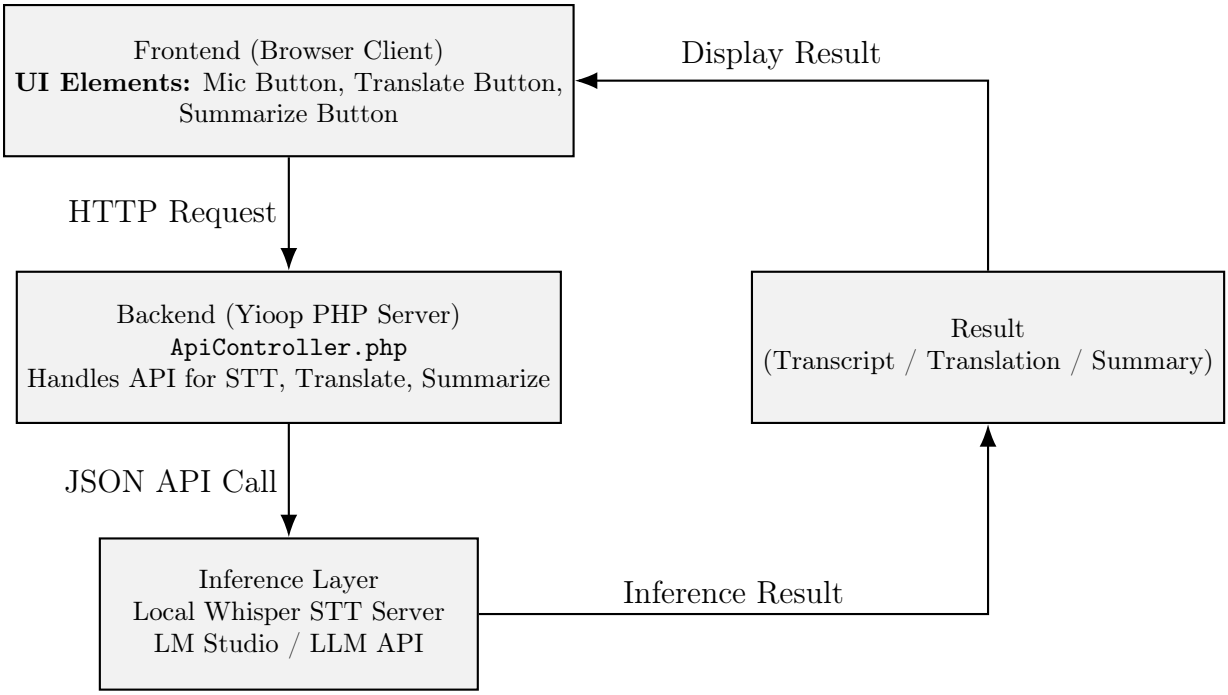


Figure 1: System architecture overview showing the three-tier design and data flow. A three-layer system with frontend browser client at top, backend PHP server in middle, inference layer at bottom, and result box to the right. Arrows indicate flow from frontend to backend, backend to inference layer, inference results to result, and back to frontend.

4.2 System Architecture Overview

The system has a general form of 3-layered system as in Figure 1.

1. **Frontend (Browser Client):** This has JavaScript-based controls to record audio. It initiates the process of transcribing the audio recordings, initiate translations and create summaries.

2. **Backend (Yioop PHP Server):** The language tool is offered by defined API endpoint routes in ApiController.php. The MediaJob framework to manage the transcription feature asynchronously is also present.

3. **Inference Layer (External AI Servers):** Uses Whisper to do audio-to-text and large language models to do translations and summarizations. All of them

are locally run using HTTP endpoints.

4.3 Pre-requisites for next chapter

The following chapter is more elaborate on how messages and threads are stored in Yioop, how Media Jobs work.

4.3.1 Thread structure

Yioop conversations are arranged into threads which may be either public forums or private chat. Yioop uses a very simple, but efficient format to store threads, a single table named GROUPITEM. A thread in Yioop starts with a message which is called a thread head and all the replies to that message are collected under the thread head. To ensure that he/she knows what messages are associated with what threads, Yioop stores the database column PARENT_ID.

- A message is a thread head when its PARENT_ID is the same as its ID.
- A message is a response, when the PARENT_ID of the message is the ID of the thread head.

A sample of this hierarchy is as shown below in Table 1

ID	PARENT_ID	GROUP_ID	USER_ID	DESCRIPTION
101	101	5	1	"Hi"
102	101	5	2	"How are you?"
103	101	5	3	"Can we add voice chat?"

Table 1: Simplified structure of a thread in the GROUP_ITEM table.

4.3.2 Message structure

Messages are also stored in the GROUP_ITEM table as indicated above. Yioop forms personal teams where the naming system would be based on the user ID of the members (i.e., 1-2 when user 1 and user 2 are communicating). An example of a message record would be the following:

- `DESCRIPTION`: The main message body.
- `TITLE`: The title that will be appended to the conversation (Most of the times will be generated automatically).
- `USER_ID`: Author of the message.
- `GROUP_ID`: naming of the conversation or group.
- `PUBDATE`: Message timestamp.

Certain group or thread is accessed through the application of the getGroupItems method applied in messages. The solution favors pagination, user-filtering and the option of distinguishing the main in threads and the responses.

Listing 4.1: Adding a message to a group with optional encryption

```
public function addGroupItem($parent_id, $group_id, $user_id,
   $title, $description) {
    $post_time = time();
    if ($this->isGroupEncrypted($group_id)) {
        $key = $this->getGroupKey($group_id);
        $title = $this->encrypt($title, $key);
        $description = $this->encrypt($description, $key);
    }

    $sql = "INSERT INTO GROUP_ITEM (PARENT_ID, GROUP_ID,
       USER_ID, TITLE,
            DESCRIPTION, PUBDATE, TYPE) VALUES (?, ?, ?, ?, ?,
               ?, ?)";
    $db->execute($sql, [$parent_id, $group_id, $user_id, $title
       ,
                        $description, $post_time, C\
                           STANDARD_GROUP_ITEM]);
    return $db->insertID("GROUP_ITEM");
}
```

Messages for a given group or thread are retrieved using the `getGroupItems()` method. This method supports pagination, user-based filtering, and can distinguish between top-level thread posts and nested replies.

Listing 4.2: Fetching messages from a group with optional pagination and user filtering

```
public function getGroupItems($limit, $num, $search_array,
   $user_id, $for_group) {
    $sql = "SELECT GI.ID, GI.GROUP_ID, GI.USER_ID, GI.TITLE,
                   GI.DESCRIPTION, GI.PUBDATE,
                   U.USER_NAME
            FROM GROUP_ITEM GI
            LEFT JOIN USERS U ON GI.USER_ID = U.USER_ID
            WHERE GI.GROUP_ID = ?
            ORDER BY GI.PUBDATE ASC
            LIMIT ?";
    return $db->execute($sql, [$for_group, $num]);
}
```

4.3.3 Media Jobs

Yioop background processing system is known as Media Jobs which is mediated by MediaUpdater daemon. The development of this architecture is done on the basis of enabling computational intensive operations to be performed in non-blocking mode so that the UIs are not blocked by long running processes. The MediaUpdater.php software is an old-time command-line interface (CLI) daemon software that actively reads all the jobs entered into it and runs them whenever all the conditions specified in the requirements are met.

Listing 4.3: Main loop of MediaUpdater daemon processing background jobs asynchronously

```
public function loop()
{
    while (CrawlDaemon::processHandler()) {
        $this->getUpdateProperties(); // Fetch current job list
        foreach ($this->jobs as $job_name => $job) {
            $job->run();
        }
        sleep(self::MINIMUM_UPDATE_LOOP_TIME); // Rate limiting
    }
}
```

An abstract base class of the MediaJob is created based on the logic of every job. The rationale behind determining jobs is a mixture of a process such as checkPrerequisites, nondistributedTasks and doTasks.

Listing 4.4: Abstract MediaJob class defining lifecycle methods for background tasks

```
abstract class MediaJob {
    public function checkPrerequisites() {}
    public function nondistributedTasks() {}
    public function getTasks() {}
    public function doTasks() {}
    public function putTask($task_data) {}
}
```

CHAPTER 5

System Design and Implementation

In this Chapter we will explain how the transcription, translation and summarization functionality was implemented in Yioop. We focus here on the architecture of the system and how it can be integrated into Yioop. To achieve this, our design focuses on three main aspects - modularity, re-usability and as little interference as possible with the core application. For this reason all language processing happens on the device by an inference engine that runs locally, which is in line with the privacy first idea behind the project.

5.1 Transcription Implementation

The transcription feature provides users with automatic speech-to-text functionality on the audio content of their messages that have been previously uploaded to Yioop via Open AI's Whisper model. This functionally has been added to the existing API pattern of Yioop, along with an extension to handle asynchronous media jobs via the Media Job framework.

5.1.1 User Interface

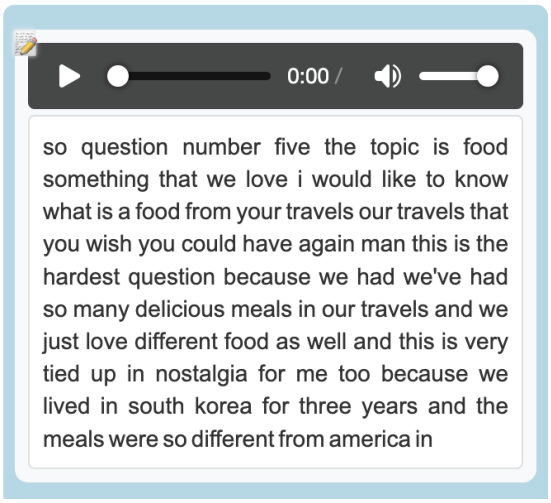
Since we store the transcripts in the same place as the audio resource, the transcription button uses the same logic to fetch the resource by replacing the audio extension with .txt.

Visual Cues: Transcribe buttons appear as overlay icons on audio messages `messages.css:265-290`. (see Figure below for reference)

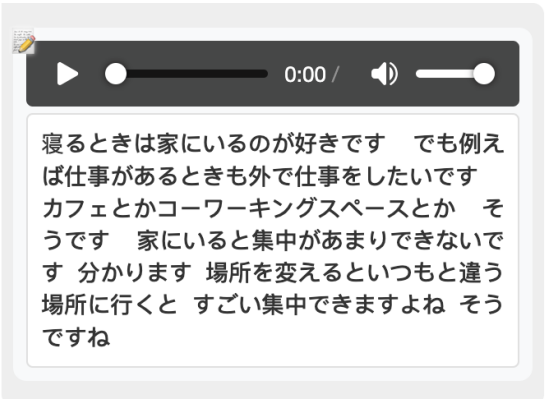
Fetching: Initiated when user clicks on the button.

UI Placing: Transcripts appear below audio players when completed.

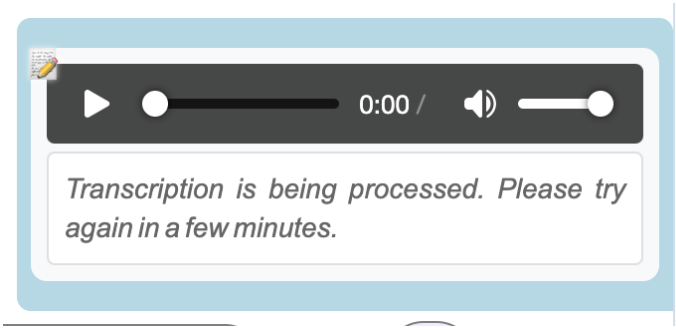
Error Handling: Failed transcriptions show appropriate error messages.



(a) English Audio Transcription



(b) Japanese Audio Transcription



(c) Status Feedback (Processing/Busy)

Figure 2: The transcription user interface results for English and Japanese audio, and system status feedback.

5.1.1.1 Configuration and Prerequisites

The listing below shows the necessary configuration for setting up audio transcription. You can enable auto transcription by setting the Whisper path. There are also other configurable setting available.

Listing 5.1: Transcription configuration constants for Whisper audio processing

```
/** Path to Whisper binary for audio transcription */
nsconddefine('WHISPER', '');
/** Maximum size in bytes for audio files that can be
   transcribed */
nsconddefine('MAX_AUDIO_TRANSCRIBE_SIZE', 100000000);
/** Timeout in seconds for audio transcription processing */
nsconddefine('TRANSCRIPTION_TIMEOUT', 1800);
```

5.1.2 Task Management

Tasks are managed through different kinds of files stores in `SCHEDULES_DIR/audio_transcribe/` directory. There are 3 main kinds of files generated - task, status, and transcript files. Task files, saved as taskid.txt, contain a serialized PHP object which has metadata about a job. Status of each task is tracked in simple text file called Status files, saved as taskid.status. Transcript files are used to collect the final text output in and saved together with the original audio files.

The `transcribe()` method in (`ApiController.php:77-158`) also includes full validation which includes steps like checking the validity of the CSRF token, checking if the audio file actually exists and is within the size limit, and most importantly that Whisper is available at the configured path.

5.1.3 Integration with Background Job Framework

As mentioned earlier, the `AudioTranscriptionJob` class (`src/library/media_jobs/AudioTranscriptionJob.php`) extends Yioop's MediaJob (Media Jobs) framework which is designed for handling background jobs that process media data in this case, Audio Data. here are 2 keys features of this job. Format Conversion feature converts an unsupported audio format to a supported one

provided it exists on the server using FFmpeg. The Retry Logic features allows a job to be retried upto three times, should it fail. The job runs within the MediaUpdater daemon, and therefore, does not require any additional services to manage.

The transcription system maintains Yioop's security standards through 4 steps. All input parameters passed to the job are sanitized using Yioop's 23 clean() method. The API endpoints are protected with valid CSRF tokens by checkCSRFTo-ken(). Audio files are validated to make sure they exist in proper readable format and well within size limits. All shell parameters use escapeshellarg() to prevent the threat of command injection.

5.2 Translation Implementation

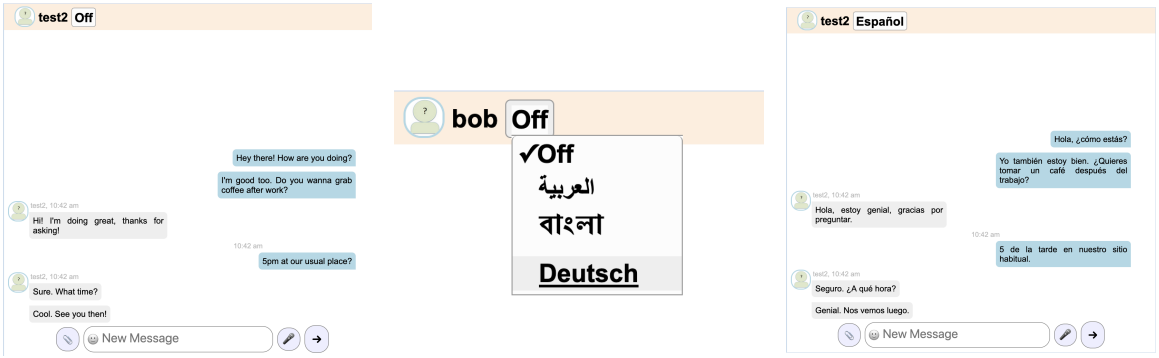
The translation module is based on a synchronous mode and relies on large language models offered by LM Studio or compatible APIs to provide real-time translations of user chat messages. While the transcription system takes time to generate the transcripts, the translation system provides immediate results following the user's input.

Since translating all the messages in real time would render the UI unusable, Yioop only showed their users the last 100 messages. To resolve this issue we added an infinite-scroll feature where users are shown messages in a batch of 15 every time they scroll to the top. This pagination-like mechanism allows us to perform real-time conversions without it affecting the User Experience of texting with other users.

5.2.1 User Interface

The translation feature is designed to work at the conversation (i.e., message) level to provide a clean, clutter-free user experience (see Figure below). As such, the

language control, is positioned in the chat-header (`UsermessagesElement.php:54`) This allows users to select a preferred translation option without having to leave their conversation window. When a user selects a new preferred language, the `messages.js:93` client side logic will then perform a batch update of all currently visible messages so that the conversation view will be updated. While the messages are being translated, an individual loading icon will appear in the message body indicating that the message is being translated. To provide a consistent experience between browser sessions, the selected translation option will be stored on the server so the same preferred translation setting is available when the user returns to their conversation.



(a) Original (English) (b) Language Selection (c) Translated (Spanish)

Figure 3: The translation workflow: original message, language selection, and dynamic update in Spanish.

5.2.2 Language Mapping and Locale Integration

The locale codes used by Yioop were mapped to names of languages that are understandable to LLMs (in `ApiController.php:73-83`). For example : "`zh_CN`" was translated to "`Chinese (Simplified)`" and "`fr_FR`" was translated to "`French (France)`". The LLM returns data using XML-like structured tags (< and >) so we

could safely parse the output.

5.2.3 Prompt Engineering

In order for the language model to accurately translate chat messages into the users preferred language, it was necessary to develop a translation prompt which would address the unique nature of informal messaging. The model was therefore instructed to decode HTML entities prior to translating the message; the model was further instructed to maintain formatting placeholders (e.g., %s) in their original position to avoid damaging the message's layout while rendering. The model was also instructed to frame the message as "conversational" as opposed to "formal" since this is the typical nature of chat messages. Finally, the model was instructed to generate its translation in a structured tagging format to enable safe and consistent parsing by the application. As a result of these design decisions, the development team has been able to greatly improve the accuracy and quality of translations of short, conversational messages with respect to generic translation prompts.

5.2.4 Conversation-Level Translation Preferences

Rather than translating individual messages, the system implements conversation-level translation preferences stored in the `GROUP_ITEM` table as JSON metadata (`GroupModel.php:59`). The translation language for a conversation is chosen by a user, and then saved when the user chooses it using the `GroupModel::setTranslationLanguage()` method. Once the preference has been saved, the client side JavaScript will update the current thread with the new translation for all messages being viewed by the user, using batch processing for the messages already being displayed; any messages received after the selection of the translation

language will be translated using the previously saved preference, and therefore the translation choice will not require additional user intervention for consistency. Since these preferences are stored in the database as opposed to the browser, the user's translation choice will persist across sessions, providing a consistent experience for the user.

5.2.5 Frontend Integration

The following files were modified:

- **SocialComponent.php:136-145**: Adds available locales and current translation settings to message views
- **UsermessagesElement.php:54**: Provides translation language selection dropdown in chat interface
- **messages.js:93**: Handles batch translation API requests and manages the corresponding UI state during translation, including loading behavior.
- **messages.css:14**: Defines the styles for translation controls and loading states

5.2.6 API Integration and Error Handling

The translation request uses the the `sendLLMRequest()` infrastructure as for summarization (same timeouts, same reporting of errors). The system also propagates errors to the UI when LLM services are unavailable.

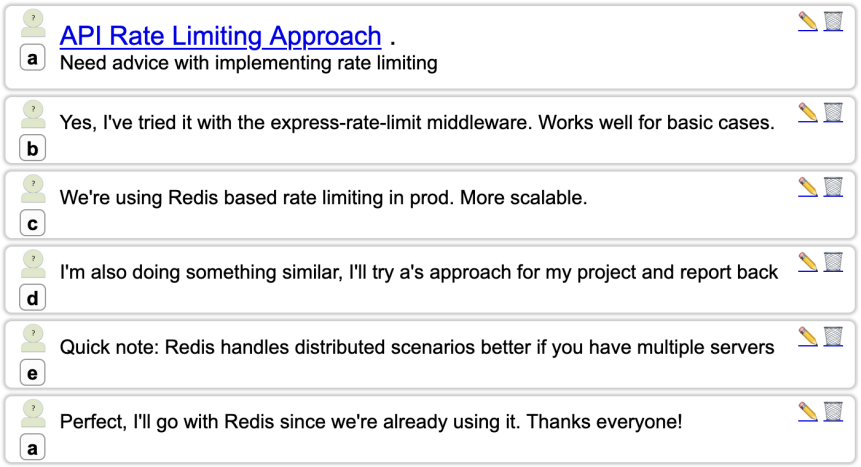
5.3 Summarization Implementation

The summarization feature enables thread summaries to be generated by large language models, enabling users to gain a quick understanding of the conversation

context without having to read through each post in the thread. In addition to this, the implementation of the summarization feature is designed to generate the summary on demand rather than caching it so that new posts are always included and users do not have to re-generate the summary every time a new message is added to the thread.

5.3.1 User Interface

The summary button is located at the top of each thread, as opposed to adjacent to each individual message, so that users who want to use it can intuitively find it but the users who do not want to use it do not find the UI intrusive. (see figure below)



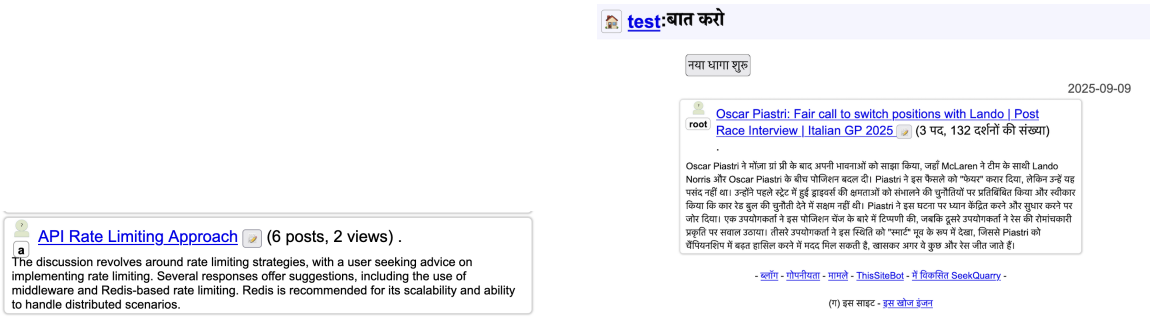
(a) The Thread Context

(b) The "Summarize" Trigger

Figure 4: The summarization entry point. (a) The user views a long thread of messages. (b) The "Summarize" button is available at the top of the interface.

When activated, summaries are presented in a modal overlay so that they do not interfere with the messages. (see figure below) Users can generate summaries again

any time to include new messages in the summaries. All styling for the summarization interface was developed using Yioop's existing styles from `messages.css`.



(a) Standard Summary (English) (b) Localized Summary

Figure 5: The summarization results: (a) A generated summary of the thread in English. (b) A thread summarized in the user's preferred locale, demonstrating multilingual support.

5.3.2 On-Demand Generation

In contrast to the caching of summaries, which may be outdated as conversations evolve, the summarization system ensures that each generated summary accurately reflects the most current state of the thread at the time of the request. Due to their dynamic generation, the system avoids problems with the stale data, and it does not require the complex cache invalidation logic when a new message is posted to an active thread. Additionally, generating summaries at request time results in better contextual accuracy due to the fact that recent developments in the conversation are always included in the generated summary.

However, there is a tradeoff for this methodology; namely, the slightly longer response times (typically 3-5 seconds). Given the advantages of improved accuracy in the summaries and the simplicity of the overall architecture, this delay is deemed acceptable. Although caching could potentially be used in the future if the performance

requirements were to change, it would likely introduce unnecessary complexity into the implementation of the current use case.

5.3.3 Message Retrieval and Preprocessing

The `summarize()` method (`ApiController.php:67-128`) utilizes Yioop's existing `GroupModel::getGroupItems()` infrastructure to fetch messages. To ensure that the correct messages are retrieved for the targeted thread, the messages are filtered based on the parent_id field. For larger conversations, the system can process up to MAX_SUMMARIZE_MESSAGES messages, where MAX_SUMMARIZE_MESSAGES is configured to 10,000 by default but can be modified via configuration. During this process, the original chronological ordering of messages is preserved so that the language model receives the conversation in the correct sequence.

Before sending the formatted messages to the language model, each message is formatted as `"On {timestamp}, {username} wrote: ¨{content}¨`. This consistent format of formatting the messages provides the language model with a clear speaker attribution and conversational context, both of which assist the language model in producing more accurate summaries.

5.3.4 Prompt Engineering

The summarization prompt was created to be consistent, concise and will always be parseable by the client. It explicitly tells the model to produce a summary in the requested target language and to return the summary enclosed in and tags. Because of the strict output constraint, the model cannot return any explanation, comment or additional metadata that would have to be parsed by the client.

```
$summarize_prompt = sprintf(
  "Summarize the following thread concisely in %s, capturing
     the key points:%s IMPORTANT: You MUST output ONLY the
     summary
between <out></out> tags. Do not include any other text,
   commentary, or explanations outside these tags.",
  $target_language, $input_text);
```

The structured approach using XML-like tags (`<out></out>`) ensures reliable parsing even when dealing with multilingual content or complex conversation topics. The system-level messages reinforce the formatting constraints and help minimize the risk of malformed responses and parsing errors.

5.3.5 Error Handling

Before processing the user's summarization request, the summarization system checks whether the submitted thread contains any messages and displays an error message to the user if the thread is empty. The system also implements safe-fail functionality for scenarios in which the LLM service is unavailable (e.g., connection timeout), and therefore leaves the user with no indication of what occurred. In cases in which the returned response is formatted in ways that are not expected or valid, the system will perform additional validation to determine if there were errors in parsing the response and provide the user with feedback about the failure. As with the rest of the system, all error messages are internationalized using the `tl()` function so that they can be displayed in the user-selected language.

5.3.6 Integration with Thread Interface

The summarization feature is integrated into the thread interface, and uses the same UI design patterns throughout Yioop. The summary button is located at the

thread level so that users understand that the summary operation is applied to the entire conversation rather than to individual messages. The system also makes use of asynchronous JavaScript calls for all of the summarization operations so that the user can continue to interact with the interface while waiting for the model to process the request. We also hide the summary button if the server does not have LLM enabled.

5.4 Audio Recording Implementation

Audio messaging uses the browser's native `MediaRecorder` API to capture voice messages. The implementation in `src/scripts/basic.js:166` provides a natural recording experience. The button only shows up in browsers that support `MediaRecorder`, and the button changes from record to recording when the user starts recording the message. When it is done recording, it allows users to play it back or delete in case they do not want to send it. We also have client side validation which prevents excessively large recordings. See the figure below for details

(a) Idle State (b) Recording (c) Review (Send/Delete)

Figure 6: The audio recording workflow: (a) Microphone button ready. (b) Visual feedback during recording. (c) Review state allowing the user to send or delete the clip.

5.5 Configuration and Deployment

The language features do not require a lot of changes to existing installations.

5.5.1 External Service Dependencies

There are two types of services that Yioop needs to use the discussed features. The first is Whisper, OpenAI's speech recognition model, which performs transcriptions on audio recordings. The second is an LLM (Large Language Model) API Server, for translation and summarization.

5.5.2 Configuration Parameters

All configuration is handled through Yioop's standard `LocalConfig.php` mechanism using `nsconddefine()` constants:

```
// Transcription configuration
nsconddefine('WHISPER', '/opt/homebrew/bin/whisper');
nsconddefine('FFMPEG', '/opt/homebrew/bin/ffmpeg');
nsconddefine('MAX_AUDIO_TRANSCRIBE_SIZE', 100000000);

// LLM API configuration
nsconddefine('LLM_API_URL', 'http://localhost:1234/v1/chat/
   completions');
nsconddefine('LLM_MODEL', 'aya-23-8b');
```

CHAPTER 6

Results and Evaluation

In this Chapter we evaluate the three language-related functions that have been added to Yioop: speech-to-text transcription; message translation; and conversation summarization. We want to see whether or not the language-related functions are performing as intended when using them in "real world" situations (with an emphasis on functionality, system reliability, time it takes for responses and usability). We tested the language-related functions in real-world usage within a self-hosted environment. All tests were run on a local deployment of Yioop, utilizing the environment described below.

- Apple MacBook Pro (M2 Pro, 10-core CPU, 16-core GPU)
- 16GB unified memory
- macOS sequoia 15.3.1
- Whisper and LM Studio hosted locally

Evaluation criteria included:

1. Functional correctness
2. Edge case behavior
3. Latency and performance
4. Qualitative user experience

6.1 Overview

A set of tests were run in a structured manner to assess how well the new, AI-based tools functioned. Testing involved automated scripts run against standard datasets

and user feedback sessions. Two Large Language Models (LLMs) were evaluated as backend processors for these features: **AYA-23 (8B)** and **Llama-3.1 (8B)**.

6.2 AI Translation

To translate text as discussed.

6.2.1 Functional Testing

The goal in this study was to ensure that the translation tool worked correctly and reliably . Unit and integration tests confirmed that the translation API endpoint is reachable, accepts requests with appropriate parameters, interacts correctly with the LLM backend, parses the LLM response (including extracting text from <out> tags), and returns the translated text to the frontend. We found that the core functionality works as expected.

6.2.2 Performance Testing (Latency)

This was to measure the time to receive a translation from the API. Automated scripts were created to submit 100 translation requests to the API for each language pair (English to Spanish, English to French, English to Hindi, English to Tamil, English to Telugu, English-Chinese and vice versa). The average latency for all submitted requests for each pair was calculated.

Results: Please see results tabulated in the table below.

As we can see from the results, For Spanish, French, and Hindi, the models provided translations in approximately 2 to 4 seconds. AYA-23 produced excessive latency for Telugu (greater than two minutes) and Tamil (approximately one minute), indicating difficulties in processing or timeouts for these languages. On the other

Table 2: Average Translation Latency (ms) per Language Pair

Language Pair	Avg Latency (ms) - AYA-23	Avg Latency (ms) - Llama-3.1
en-es	1859	2232
en-fr	2040	2349
en-hi	3599	2868
en-ta	56864	20758
en-te	120073	9990
en-zh	1548	103071

hand, Llama-3.1 produced significantly lower latency for Tamil and Telugu than did AYA-23. Additionally, Llama-3.1 demonstrated substantially higher latency for Chinese (greater than 1.5 minutes) whereas AYA-23 processed it quickly. Finally, Llama-3.1 was slightly slower than AYA-23 for both Spanish and French.

6.2.3 Quality Evaluation

The goal here was to evaluate the quality of translations using standard reference translations that have existing evaluation metrics. Using the same test set used in the previous section, the generated translations were compared to the reference translations using both COMET [16](a metric correlating well with human judgment) and BERTScore F1 [17] (measuring semantic similarity). Only scores of successful translations were considered.

Results:

Analysis:

AYA-23 Demonstrated strong quality (COMET > 0.7, BERTScore > 0.9) for Spanish, French, Hindi, and Chinese. Performance dropped significantly for Tamil and Telugu, correlating with the high latency and lower success rates (97% and 90% respectively). **Llama-3.1** Showed comparable quality to AYA-23 for Spanish and

Table 3: COMET Scores (Higher is better) per Language Pair

Language Pair	AYA-23	Llama-3.1
en-es	0.8448 (100/100)	0.8085 (100/100)
en-fr	0.7797 (100/100)	0.7296 (100/100)
en-hi	0.7251 (100/100)	0.6961 (100/100)
en-ta	0.5437 (97/100)	0.6082 (99/100)
en-te	0.4754 (90/100)	0.7089 (100/100)
en-zh	0.8145 (100/100)	0.4533 (82/100)

Table 4: BERTScore F1 (Higher is better) per Language Pair

Language Pair	AYA-23	Llama-3.1
en-es	0.9507 (100/100)	0.9445 (100/100)
en-fr	0.9276 (100/100)	0.8680 (100/100)
en-hi	0.9135 (100/100)	0.8711 (100/100)
en-ta	0.8583 (97/100)	0.7126 (99/100)
en-te	0.8566 (90/100)	0.8874 (100/100)
en-zh	0.9241 (100/100)	0.5869 (82/100)

Hindi. It performed better than AYA-23 on Telugu and slightly better on Tamil (COMET). However, it performed notably worse on French and especially Chinese (COMET < 0.5, BERTScore < 0.6, with only 82% success rate), correlating with its high latency for Chinese. Overall AYA-23 appears superior for most tested languages except Tamil/Telugu where its latency and success rate were problematic. Llama-3.1 struggled significantly with Chinese quality and latency.

6.2.4 Usability Testing

The Goal here was to Gather initial feedback on the feature's ease of use and perceived quality. A small pilot study was simulated with 5 internal users. They were asked to perform common tasks involving translation within the application interface. We included both existing and new users to Yioop in this study.

Results: Users generally found the translation option easy to locate and activate. In practice, users found the translation speed to be bearable for all languages that were tested. Perceived translation quality was generally rated as "good" or "very good" for common European and Asian languages. Some minor awkward phrasing or unnatural word choices were occasionally noted, but the core meaning was usually conveyed.

6.3 AI Summarization

This feature enables users to create a summary of group discussion threads.

6.3.1 Functional Testing

The purpose is to test basic functionality brought by the patch. The integration tests ensured that the button "Summarize Thread" checked invokes the appropriate JavaScript process (toggleAISummary).It in turn invokes the new `/api/summarize` endpoint with the threadid. Server-side tests asserted that messages(using `getMessages`) are properly retrieved by the ApiController.summarize action.

We verified that Yioop makes the API to the LLM (sendLLMRequest), processes the response (removes text enclosed in <out> tags), and outputs a JSON containing summarization or errors in response. Display logic of the frontend of hiding/ showing the summary div was checked as well.

Result: Essential functionality is in place. Automated testing (below) indicated large API reliability.

6.3.2 Performance Testing (Latency)

The goal now was to time how long it will take to click the button until the summary appears. Procedure included simulating 500 threads of user clicks in automated scripts. (assuming that the IDs used are those of dialoguesum [18] thread IDs). Mean end-to-end latency was obtained.

Results:

Table 5: Average Summarization Latency in milliseconds for each model

Model	Avg Latency (ms)	Notes
AYA-23	6513	Based on 499 successful calls
Llama-3.1	5348	Based on 492 successful calls

Comparison Llama-3.1 was significantly faster with an average of 5.3 seconds compared to 6.5 seconds for AYA-23. Both are a physical wait time towards the user.

6.3.3 Quality Assessment (Automated Measures)

Goal was to measure the summary quality with reference summaries and standard metrics. With the dialogue_summary_dataset.jsonldataset (500 items) created overviews of the API calls were contrasted with a reference summary using ROUGE [19] (1, 2, L) and BERTScore F1 [17].

Results:

Table 6: Average summarization quality metrics (F1 Scores) for each model

Model	Avg ROUGE-1 F1	Avg ROUGE-2 F1	Avg ROUGE-L F1	Avg BERTScore F1	Notes
AYA-23	0.289	0.099	0.225	0.651	Based on 499 valid metrics
Llama-3.1	0.273	0.092	0.205	0.639	Based on 492 valid metrics

The two models have low ROUGE scores, especially, ROUGE-2 (0.1). This means that there is a huge absence of lexical redundancy (common words and phrases)

between the generated summaries and reference summaries in the dataset. The two models record poor BERTScore F1 scores (around 0.64-0.65). This indicates moderate semantic similarity—the summaries generated. abscond part of the meaning but are much different in the references. AYA-23 has been found to be marginally better than Llama-3.1 on all measures, though it is a slight difference. In general, when compared to these metrics against the given standard references, both models are rated medium-to-low in terms of quality.

6.3.4 Usability Testing

Goal was to get a first impression about how useful the function is for users, how fast it works, and what kind of quality of summaries can be expected. A small test scenario has been developed based on the model, which involved five internal users who communicated and worked together on various tasks of varying length, similar to the other studies.

Results: The concept was valued by the users, and the button is convenient to make use of getting an overview of long threads. The wait time of 5-7 seconds was considered as acceptable considering the amount of time saved, even though it would be desirable to be faster. The critiques on the quality of summaries were varied. The summaries were commonly found acceptable by the users, but sometimes missed helpful notes that got buried by the noise from irrelevant messages. One user was critical about the feature since the AI had not quite grasped the subtlety of the conversation, but they agreed that it was easy to ignore the button if they did not want to use it, so it did not ruin the User Experience.

6.3.5 Model Comparison Summary

AYA-23: Typically translations of a slightly higher quality (except Ta/Te). and summaries using automated metrics. Demonstrated extremely high reliability but occasionally had problems of latency. It did not do a good job translating to Tamil and Telugu, and was slower than Llama-3.1 in summarizing.

Llama-3.1: Holds much superior latency on summarization and on the other model in Tamil/Telugu translation. However, it underperformed on Chinese translation (both quality and latency) and scored a little lower than AYA-23 on the quality metrics of a summary. It had a lower API reliability (98.4%) success rate as well.

6.3.6 Conclusion and Discussion

Translating and Summarizing services with local LLMs AYA-23 and Llama-3.1 is proven to be successful and is highly reliable. Models and tasks were seen to have differences in terms of performance. Llama-3.1 offers faster summarization, whereas AYA-23 has better speed on most translation pairs.except Tamil and Telugu. Both models are characterized with high latency of particular language pairs. (Te/Ta on AYA-23, Zh on Llama-3.1), which may affect the user experience. Although the quality of translation is strong in both in most cases, summarization quality when compared to reference summaries by ROUGE models is still only mediocre. The poor ROUGE results indicate that the models produce summaries and using very different phrases than the references.

According to these preliminary findings, AYA-23 seems a bit better based on two main reasons. It had a better overall translation quality and a slight improvement in summarization scores, despite its specific latency issues.

The next steps that can be recommended are: Carrying out more extensive

usability testing. Researching into engineering methods to enhance the quality of summaries (e.g., focus, detail level). We can also try finding more optimized models in dialogue summarization.

6.4 Audio Transcription

The automatic transcription is achieved by utilizing an asynchronous job processing system that uses Whisper as the back-end (turbo model).

6.4.1 Functional Testing

This testing was mainly done to ensure the complete end-to-end automatic transcription asynchronous workflow. The main workflow was confirmed by the successful creation of jobs and transcript retrieval of 33 test files sent to the system in FLAC-formatted audio files, which initiated a MediaJob. Furthermore, it was tested that errors messages were shown when there were no ground truths. The tested base workflow was asynchronous and therefore found to be working as per the expected file formats. There is a need to test error conditions further (e.g. invalid input, processing errors).

6.4.2 Performance Testing (Transcription Time)

The aim was to determine the duration of Whisper backend to transcribe audio files. Measurement of the time in transcription did not encompass time in queue since it became submitted. The evaluation script measured the duration of the completion of the call to the whisper.transcribe() script to complete the entire 33 FLAC files in a subset of a simulated LibriSpeech LibriSpeech[20] like dataset. Each of the audio files tested had average transcription times of approximately 5.0 s (164.27 s / 33 Files) on

average. The average file times were 4.5s-6.1s. As a rule, the above shows that the general performance of the Whisper Backend with very short audio clips is relatively constant regarding processing time with these files. The total processing time time of the users within an asynchronous environment will however be longer than the processing time itself because the delay factors are further included due to the delay in job handling and queuing.

6.4.3 Quality Evaluation

This evaluation was mainly aimed at evaluating the quality of transcriptions compared to actual transcripts.

The actual transcripts (ground truths) and the generated transcripts of the Whisper model were compared by Word Error Rate (WER) at total count of 33 FLAC files based on LibriSpeech which is a typical speech database and has its own ground truth transcripts to compare with generated ones.

Average WER = 0.0157 (or 1.57%) over the 33 files.

This is a very, very low number in automatic speech recognition, and hence shows that this model was very good in terms of accuracy on our test set. Evaluation of more challenging audio like audio recorded in a noisy place or when multiple speakers with different accent profiles are present may give different results.

6.4.4 Usability Testing

The main aim of this study was to review the usability of the new workflow, it's perceived accuracy, and general utility.

Like the other studies, 5 internal users were included. List of audio file uploads (some good quality recordings and others with noise in the background) was provided

to the users. The users received their transcripts once they became available. They were also encouraged to

Uploading audio files was easy to accomplish as indicated by all users. The majority of users knew that the system is asynchronous but system can be improved to better communicate the time to completion, and maybe also notify the users when the transcript is available.

The users believed that the system did a decent job and was comparatively fast when it came to the shorter audio clip jobs. As stated above, the users would like to see an approximate or estimated turnaround time for longer audio clips.

The users felt that accuracy of the transcription was reduced when background noise in the audio file was high or when there were more than two speakers.

CHAPTER 7

Conclusion

This project successfully enhanced the method of communicating through Yioop. It includes voice messages, translation of conversations in real time, and making summaries of conversations. All features kept privacy in mind and enabled users to have control over their information. This project is one way existing open-source platforms can be adapted to incorporate AI-based collaboration tools without ruining their design.

7.1 Key Contributions

The key contribution of our project is improving the user experience of Yioop. We created an audio messaging system that also automatically transcribes it accurately. It also introduces a conversation translation system that assists in bridging the language barrier. WE also added an on demand summary tool which lets users glance long conversation threads.

This project also shows how new AI-driven workflows can be introduced to old systems by modular design. It is also a good example of privacy-preserving AI design that can be followed by other open source projects.

7.2 Lessons Learned

One of the key lessons learned is that the type of model you choose impacts the performance of your application a lot. The trade-offs between processing speed and output quality are something you need to consider for your use case. To guarantee the accuracy of the decoding of the results of the models it is important to give the AI models specific instructions on how to structure the output. We did this by using prompt engineering to get XML-like tags, but other ways can be considered.

While implementing chat translation and audio message transcription, we learned the importance of asynchronous design. Users leave the application if the interface is blocked for even half a minute, but are okay with the delay if the system shows them information or error messages.

Another key lesson was the importance of good User Experience. No matter how good a feature is, a user will not use it if it does not have a good User Interface or helpful interactions (like operation errored out, try again)

7.3 Future Directions

This research could be expanded in many directions. From the perspective of the features already added, the immediate two goals that come to mind are stress testing the system and using different models to find general purpose models that give better results. Other things that future contributors could explore are adding speaker diarization or using alternative methods for audio transcription. Another challenging feature to explore is adding caching to the summaries. Since AI agents return different outputs on the same prompts every time you run them, we could potentially have 5 summarize calls for the same thread and only store the summary with the best score. A lot of research can go into this score calculation as well.

If we talk about adding new features, there are a lot of AI enabled features that we can explore. We have already established the process for adding backend, frontend, and AI inference components to Yioop, so it should be easy for new contributors to add more features. Since Yioop supports Video Calls, a good addition in line with the work done would be real-time captions in calls. Yioop also has its own Chatbot framework, so having a Generative AI chatbot that the users can chat with to use as a personal assistant would be immensely helpful.

LIST OF REFERENCES

[1] C. Pollett, ‘‘Yioop! open source search engine,’’ https://www.seekquarry.com, 2009--2025, accessed: 2025-12-04.

[2] J. Oikarinen and D. Reed, ‘‘Internet Relay Chat Protocol,’’ RFC 1459, 1993. [Online]. Available: https://www.rfc-editor.org/info/rfc1459

[3] PubNub, ‘‘What is xmpp? extensible messaging and presence protocol,’’ https://www.pubnub.com/guides/xmpp/, accessed: 2025-12-04.

[4] XMPP Standards Foundation, ‘‘History of xmpp,’’ https://xmpp.org/about/history/, accessed: 2025-12-04.

[5] I. Fette and A. Melnikov, ‘‘The WebSocket Protocol,’’ RFC 6455, 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6455

[6] R. L. Daft and R. H. Lengel, ‘‘Organizational information requirements, media richness and structural design,’’ *Management Science*, vol. 32, no. 5, pp. 554--571, 1986.

[7] ‘‘Social presence and trust in audio messaging,’’ *Journal of Computer-Mediated Communication*, 2025, doi: 10.1080/1553118X.2025.2515277.

[8] e. a. Aslan, ‘‘Speejis: Enhancing user experience of mobile voice messaging,’’ in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2025.

[9] ADOC Studio, ‘‘Markup versus wysiwyg,’’ https://www.adoc-studio.app/blog/markup-versus-wysiwyg, accessed: 2025-12-04.

[10] Human Science Co., Ltd., ‘‘Docs as code,’’ https://www.science.co.jp/en/document_jamstack_blog/41130/, accessed: 2025-12-04.

[11] e. a. Hofbauer, ‘‘Large-scale collaborative writing: Technical challenges and recommendations,’’ 2023.

[12] J. Sweller, ‘‘Cognitive load theory,’’ *Psychology of Learning and Motivation*, 2011. [Online]. Available: https://psycnet.apa.org/record/2011-17503-002

[13] Cohere For AI, ‘‘Aya 23: Open weight releases to further multilingual progress,’’ https://cohere.com/research/aya, 2024, accessed: 2025-12-04.

[14] Alpha Cephei Inc., ‘‘Vosk offline speech recognition api,’’ https://alphacephei.com/vosk/, 2025, accessed: 2025-12-04.

[15] OpenAI, ‘‘Whisper: Speech recognition,’’ https://github.com/openai/whisper, 2022, accessed: 2025-12-04.

[16] R. Rei, C. Stewart, A. C. Farinha, and A. Lavie, ‘‘COMET: A neural framework for MT evaluation,’’ in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 2685--2702.

[17] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and M. Artetxe, ‘‘Bertscore: Evaluating text generation with bert,’’ in *International Conference on Learning Representations*, 2020.

[18] Y. Chen, Y. Liu, L. Chen, and Y. Zhang, ‘‘Dialogsum: A real-life scenario dialogue summarization dataset,’’ in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 5062--5074.

[19] C.-Y. Lin, ‘‘Rouge: A package for automatic evaluation of summaries,’’ in *Text Summarization Branches Out*. Association for Computational Linguistics, 2004, pp. 74--81.

[20] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, ‘‘Librispeech: an asr corpus based on public domain audio books,’’ in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, 2015, pp. 5206--5210.