SOFTWARE USER PERSONALIZATION THROUGH USER FEEDBACK AND BEHAVIOR

24-25J-296

BSc (Hons) degree in Information Technology Specializing in

Software Engineering

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

August 2024

DECLARATION

We declare that this is our own work, and this Thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other University or institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, We hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my Thesis, in whole or in part in print, electronic or other medium. We retain the right to use this content in whole or part in future works (such as articles or books).

Name	Student ID	Signature
Silva H. S. N	IT21324406	[illegible]
Rosa M.D	IT21215360	[illegible]
Sandun Geemal H.L	IT17117210	[illegible]

The above candidate has carried out this research thesis for the Degree of Bachelor of Science (honors) Information Technology (Specializing in Software Engineering) under my supervision.

Signature of the supervisor
(Ms. Vindhya Kalapuge) Date

Signature of co-supervisor Date
(Ms. Revoni De Silva)

ABSTRACT

With the exponential rise in online content consumption, the need for intelligent, adaptive user interfaces has become more critical than ever. This research presents the design and development of a comprehensive Chrome extension aimed at enhancing the reading experience on Medium.com through advanced Software User Interface (UI) Personalization. The system integrates three major personalization components which are content-based UI personalization, behavior-driven UI personalization, and accessibility-focused UI personalization for visually impaired users. The content-based module empowers users with features such as multi-format article summarization, interactive chatbot support, and dynamic mind map generation, providing personalized cognitive assistance through modern Natural Language Processing (NLP) techniques. The behavior-driven personalization component leverages real-time behavioral signals, including scrolling patterns, reading speed, and interaction metrics, to adaptively adjust UI elements such as font size, content density, and layout without requiring manual intervention. Simultaneously, the accessibility-focused module utilizes user-provided vision attributes such as color blindness or near vision impairments processed through a machine learning-based recommendation system to deliver individualized screen themes that enhance readability and navigation. The entire system is built with a FastAPI and Flask backend, ReactJS frontend, and integrates powerful machine learning and NLP models including XGBoost, TF-IDF, FAISS, OpenAI's GPT-3.5, and spaCy pipelines. By combining content awareness, behavioral intelligence, and accessibility adaptation into a unified platform, this research advances human-centered UI design, offering a more inclusive, personalized, and engaging web experience. The proposed solution not only reduces cognitive load but also sets a new benchmark for adaptive interface technologies, bridging the gap between static web design and truly intelligent, responsive user experiences.

Keywords - Content-Based Summarization, Chatbot, Mind Map Generation, Chrome Extension, User Interface Personalization, Behavior-Driven Design, Accessibility Adaptation, Vision-Based Personalization, Adaptive User Interface, Natural Language Processing, Machine Learning, Personalized Web Experience, Human-Centered Design, Medium Articles.

ACKNOWLEDGEMENT

We would like to express our heartfelt gratitude to our module coordinator, Dr. Jayantha Amararachchi, for his continuous guidance, support, and motivation throughout the course of this project. We are also deeply thankful to our initial supervisor, Mr. Thusithanjana Thilakarathna, whose early advice and encouragement laid a strong foundation for this work. Following his departure, We were fortunate to be supported by our current supervisor, Ms. Vindhya Kalapuge, whose dedication, insightful feedback, and patient guidance helped steer the project in the right direction. We would also like to sincerely thank our co-supervisor, Ms. Revoni, for her technical advice and consistent support during every stage of development. Our appreciation extends to the academic and non-academic staff of SLIIT, including lecturers, assistant lecturers, instructors, and to the group members, all of whom contributed directly or indirectly to the successful completion of this project. Finally, we are truly grateful to our beloved family and friends for their constant encouragement, moral support, and for standing by us throughout this journey, especially during challenging times.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviations	**Description**
SLIIT	Sri Lanka Institute of Information Technology
AI	Artificial Intelligence
API	Application Programming Interface
UI	User Interface
UX	User Experience
NLP	Natural Language Processing
LLM	Large Language Model
GPT	Generative Pre-trained Transformer
RAG	Retrieval-Augmented Generation
TF-IDF	Term Frequency-Inverse Document Frequency
FAISS	Facebook AI Similarity Search
TTS	Text-to-Speech
CSV	Comma-Separated Values
SMOTE	Synthetic Minority Over-sampling Technique
SVO	Subject-Verb-Object
JSON	JavaScript Object Notation
CRUD	Create, Read, Update, Delete
CI/CD	Continuous Integration / Continuous Deployment
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
ML	Machine Learning
SSDLC	Secure Software Development Life Cycle

1. INTRODUCTION

1.1 Background Study and Literature Review

1.1.1 Background Study

In the evolving landscape of Human-Computer Interaction (HCI), User Interface (UI) personalization has emerged as a critical methodology to enhance user experience by tailoring digital environments according to individual preferences, behaviors, and needs. As digital platforms cater to increasingly diverse audiences with varying goals and abilities, static, one-size-fits-all interfaces are no longer sufficient. Modern users expect dynamic, personalized experiences that adapt seamlessly to their reading styles, interaction patterns, and accessibility requirements.

This project focuses on developing a comprehensive UI personalization system that integrates three key dimensions:

1. Content – Based Software UI Personalization
2. Software UI Personalization through User Behavior
3. Software UI Personalization through User Feedback for Visually Impaired Users

Together, these components form a unified framework aimed at delivering personalized, accessible, and intelligent digital reading experiences, particularly on platforms such as **Medium.com**.

Content-based personalization focuses on dynamically tailoring the way information is presented to users, based on the nature of the content and the user's cognitive preferences. In this project, it is achieved through the integration of three intelligent features which are text summarization, an interactive chatbot, and automatic mind map generation.

Text summarization, a major subfield of Natural Language Processing (NLP), condenses lengthy articles into concise formats without losing the core information.

By employing large language models (LLMs) such as OpenAI's GPT-3.5, the system enables users to generate summaries in multiple styles pointwise, short, and long based on their individual reading preferences. This adaptability supports users who seek quick scanning as well as those preferring detailed understanding.

Complementing the summarization feature, an intelligent chatbot transforms the passive act of reading into an active learning experience. Built using a Retrieval-Augmented Generation (RAG) pipeline, combining TF-IDF vectorization, FAISS-based semantic retrieval, and GPT-3.5 generation, the chatbot allows users to ask questions about the article and receive contextually grounded responses. This enables deeper engagement with content and personalized knowledge acquisition. To further aid comprehension, especially for visual learners, the system integrates automatic mind map generation. By extracting subject-verb-object (SVO) relationships using spaCy and visualizing them through react-flow-renderer, the system generates interactive, radial mind maps. These mind maps allow users to intuitively grasp the structure and relationships within an article, offering a more holistic and accessible learning experience.

Together, these three features under the content-based personalization module adapt not just the interface, but the very presentation and interaction with digital content to fit individual cognitive needs and preferences.

Beyond content presentation, personalization can be significantly enhanced by observing real-time user behavior. Unlike static personalization that relies on manual settings, behavior-driven UI personalization dynamically adapts interfaces by analyzing how users interact with the system. Key behavioral signals such as scrolling speed, zoom levels, click patterns, and engagement duration provide rich data that reflect user intent, reading comfort, and attention span. By monitoring these behaviors in real-time, the system can infer user preferences and automatically adjust UI elements such as font size, line spacing, contrast settings, and content layout to optimize the reading experience without requiring explicit user intervention.

For instance, a user who frequently zooms in may automatically receive an increased default font size; a user who scrolls quickly through sections may have the content reorganized for faster navigation. Such dynamic personalization not only enhances usability but also promotes efficiency and accessibility.

Advancements in Artificial Intelligence (AI) and Machine Learning (ML), particularly predictive analytics and user modeling, make it possible to integrate these behavior-aware adaptations into real-world applications. In this project, the ReactiveWeb concept embodies this philosophy by building an intelligent, self-adjusting reading environment that responds to user behavior patterns in real-time, ensuring a smooth, customized digital experience.

Accessibility remains a crucial pillar of modern UI design. Many users, particularly those with visual impairments, face challenges interacting with standard interfaces designed for the "average" user. To address this gap, the third component of the system focuses on UI personalization through explicit user feedback, specifically targeting visually impaired users.

Rather than relying solely on behavioral inference, this module empowers users to provide direct feedback regarding their accessibility needs. Users can specify preferences such as increased text contrast, larger font sizes, simplified layouts, or reduced color complexity. Based on this feedback, the system dynamically adapts the UI to create a more inclusive environment, improving readability and interaction ease for users with varying degrees of visual impairment.

This approach acknowledges that accessibility needs are highly individualized and that the best personalization strategy often combines behavioral observations with explicit, user-driven inputs. By enabling users to actively shape their own interface experiences, the system aligns with inclusive design principles and ensures equitable access to digital content.

Together, the integration of content-based personalization, behavior-driven UI adaptation, and feedback-driven accessibility personalization represents a holistic, user-centric approach to modern interface design. By addressing the cognitive, behavioral, and accessibility dimensions of personalization, the system not only improves user engagement and satisfaction but also sets a precedent for how adaptive, intelligent UI systems can meet the diverse and evolving needs of contemporary digital users. Through this comprehensive design, the project contributes meaningfully to the field of adaptive HCI, personalized information systems, and accessible technology development, highlighting how next-generation digital experiences can and should be tailored for everyone.

1.1.2 Literature Review

In the modern digital era, the need for personalized and adaptive user interfaces (UI) has become more critical than ever. Traditional static UIs, designed with a "one-size-fits-all" approach, often fail to address the diverse needs, behaviors, and accessibility requirements of today’s users. As online content consumption continues to rise, platforms like Medium, where users engage with lengthy articles, reveal the urgent need for intelligent systems that can adapt content, presentation, and interaction flow based on individual preferences and conditions.

This research project introduces a comprehensive personalization framework divided into three major components which are Content-Based Personalization, Behavior-Based Personalization, and Accessibility-Focused Personalization for Visually Impaired Users. The following sections explore the existing works related to each area, identify their limitations, and justify the novelty of the proposed system.

Several existing systems offer content summarization and chatbot capabilities but often in isolation, lacking deep personalization and contextual adaptation.

- SummarizeBot is an AI-powered summarization tool that extracts basic summaries from input texts. However, it offers limited customization options and requires manual input outside the reading flow [1].
- Flipboard, a popular content aggregation platform, curates articles based on topic selection but does not offer real-time summarization or interactive content understanding [2].

- Microsoft Immersive Reader improves reading accessibility through text manipulation features but lacks content summarization or interactive Q&A capabilities [3].

With the rise of Large Language Models (LLMs) like GPT-3.5, abstractive summarization quality has improved dramatically, allowing summaries to be contextually aware, user-driven, and dynamic [4]. Furthermore, Retrieval-Augmented Generation (RAG) models combining semantic search techniques like TF-IDF and FAISS have enabled building chatbots capable of context-specific question answering, especially for long, unstructured texts [5].

However, most real-world implementations either focus solely on summarization or provide basic chatbot services without tight integration with the user's reading context. Additionally, users are rarely allowed to choose their preferred summary formats (e.g., bullet points, short summaries, or detailed explanations).

Thus, integrating multi-format summarization, context-aware chatbots, and mind map visualization within the reading environment presents a significant advancement over previous tools.

While static UI customization options (e.g., changing font size or theme manually) exist on many platforms, real-time UI adaptation based on user behavior remains largely underexplored.

Studies highlight that tracking user interactions like scrolling speed, reading time, click patterns, and zoom behaviors can provide critical insights into user engagement and cognitive load [6]. Systems like BeeLine Reader and Read Aloud extensions have introduced features aimed at enhancing readability, but they require users to manually initiate adjustments [7].

Recent research by Lacic et al. discusses the importance of adapting news article recommendations based on user interaction signals to increase engagement, suggesting that real-time behavioral adjustments could also enhance UI personalization [8].

Despite these findings, practical implementations of live behavioral monitoring combined with UI adaptation are rare. Most existing personalization systems either operate based on static user profiles or require explicit feedback, making them less adaptive and intrusive for users seeking seamless experiences.

Therefore, the proposed behavior-driven personalization engine, which dynamically alters UI elements in response to real-time user interactions without requiring explicit input, fills a critical gap in current HCI research and application.

Accessibility tools like Microsoft Immersive Reader and BeeLine Reader offer basic assistance through font changes, text highlighting, and color adjustments [3][7]. However, they provide limited personalization based on individual user needs and generally apply static settings universally.

Research by Sarangam and Liu et al. highlights the need for adaptive accessibility tools that adjust dynamically based on user-specific impairments and evolving needs [9][10].

In the field of machine learning, adaptive theme recommendation systems have been explored, but rarely integrated into real-time web applications. Recent advances using XGBoost classifiers and SMOTE for handling class imbalance demonstrate how vision-related data (color blindness, near vision, far vision, etc.) can be leveraged to predict and dynamically apply UI themes [11].

The EyeCareAI module developed in this system leverages these techniques by:

- Collecting user vision-related attributes
- Processing them through an XGBoost model
- Applying optimal themes in real-time via a Flask API and frontend JavaScript engine

This dynamic adjustment empowers visually impaired users to experience improved readability and navigation without requiring manual configuration, making the interface more accessible, inclusive, and human-centered.

Table 1: Comparison Table with the existing table

	Reactive Web (Proposed System)	**Summarize Bot (AI Summarization Tool)**	**Flipboard (News Personalization App)**	**BeeLine Reader (Accessibility Tool for Dyslexia & Vision Issues)**	**Microsoft Immersive Reader (Accessibility Tool)**
AI-Based Summarization with UI	**Yes**	No	No	No	No

Personalization					
Multiple Summary Formats (Points, Short, Long)	**Yes**	Predefined formats	No	No	No
Uses OpenAI API for High-Accuracy Summarization	**Yes**	No	No	No	No
Dynamic UI Adaptation Based on Scrolling Behavior	**Yes**	No	No	No	No
Zoom-Based UI Personalization	**Yes**	No	No	Helps with text readability but no AI adaption	Yes
Real-Time UI Customization Through AI	**Yes**	No	No	No	No
Personalization Based on User Feedback	**Yes**	No	Yes	No	Patial
Adaptive UI for Visually Impaired Users	**Yes**	No	No	Yes	Yes
Color Contrast and Text Size Customization	**Yes**	No	No	Yes	Yes
Integration of Accessibility Feedback	**Yes**	No	No	Yes	Yes

into UI Adaptation					
Supports Real-Time Behavioral Adjustment s	**Yes**	No	No	No	No
Chrome Extension for Easy Integration	**Yes**	Yes	No	Yes	No

1.2 Research Gap

Despite significant advancements in AI technologies, natural language processing, behavioral analytics, and accessibility tools, existing systems in the domain of user interface (UI) personalization still exhibit critical limitations that inhibit the realization of truly adaptive, intelligent, and inclusive digital experiences.

Most existing solutions focus independently on isolated functionalities such as content summarization, basic accessibility settings, or manual UI customization. Very few systems offer a comprehensive, real-time, and multi-faceted personalization experience that dynamically adapts to content complexity, user behavior, and accessibility needs especially in browser-based reading environments like Medium.

AI summarization tools like SummarizeBot and manual chatbot interfaces like ChatGPT require users to leave their reading flow, copy-paste content, or manually prompt the system [1][5]. They often offer only generic summaries without considering user preferences for format (points, short, long). Similarly, current question-answering systems are not tightly contextualized to the article users are reading. Mind map generation, an important cognitive tool for visual learners, remains largely absent in mainstream summarization tools. While some research prototypes exist for automated knowledge graphs, no real-time Chrome extension solution integrates summarization, chatbot interaction, and mind map visualization in a seamless, unified interface.

Although platforms like BeeLine Reader and Flipboard offer enhanced readability or curated content feeds [2][7], their personalization strategies are typically static relying on manual adjustments or topic-based recommendations.

Few, if any, systems utilize real-time behavioral monitoring (e.g., scrolling patterns, reading time, zooming behavior) to dynamically adjust the UI at runtime. While studies [6] advocate for behavior-driven personalization to enhance user engagement and cognitive support, practical browser-based implementations remain scarce.

Thus, there exists a gap in delivering live, non-intrusive, real-time UI personalization based on observed user behavior, especially without burdening the user with frequent manual interventions.

Traditional accessibility solutions like Microsoft Immersive Reader [3] and browser plugins often rely on generic, static settings for font sizes, color contrasts, and backgrounds. They do not account for specific vision-related conditions like color blindness, short-sightedness, or distance vision impairment.

Recent machine learning advances such as XGBoost classifiers and adaptive theme generation show promise [11], but they are rarely embedded into real-time browser applications for vision-specific personalization. Most accessibility efforts still require users to manually configure settings, making the experience less intuitive and inclusive for visually impaired individuals.

There is a clear lack of dynamic, machine learning-powered UI adaptation systems that can automatically personalize a web interface based on individual vision profiles without manual setup.

While various individual solutions exist summarization tools, basic chatbots, behavior trackers, and accessibility plugins, no existing system offers a fully integrated, real-time adaptive, and content-aware user experience combining,

- Content-Based UI Personalization (Content summarization [points, short, long], Interactive article-specific chatbot, Dynamic mind map generation)
- Real-time UI adaptation based on user behavior
- Automatic accessibility adaptation based on user feedback for visually impaired users especially for long-form reading platforms like Medium.

This critical research gap inspired the development of ReactiveWeb, an intelligent Chrome extension that bridges these functionalities, offering a holistic, user-centered digital reading experience that is content-aware, behavior-driven, and accessibility-enhanced all in real time and without disrupting the user's natural browsing flow.

Table 2: Research Gap with the Existing Systems

Feature	**Existing Systems**	**Proposed System (ReactiveWeb)**
Multi-format Summarization	Basic, static (SummarizeBot) [1]	Dynamic formats: points, short, long
Real-time Chatbot for Articles	Requires manual input (ChatGPT) [5]	Contextual, embedded chatbot
Mind Map Generation	Rarely availableAI	NLP-based automatic mind map
UI Personalization via Behavior	Static customization [7]	Real-time Behavior-driven adaptation
Accessibility Personalization	Static settings (Immersive Reader) [3]	Machine learning-driven dynamic themes

1.3 Research Problem

In the current digital landscape, users increasingly demand personalized, adaptive, and accessible reading experiences, particularly on content-rich platforms like Medium.com. However, existing solutions, whether focusing on content summarization, UI personalization, or accessibility often operate in isolation and fail to integrate a holistic, real-time, and intelligent personalization framework.
While AI-powered summarization tools like SummarizeBot [1] and general-purpose LLM-based interfaces like ChatGPT [13] can generate summaries, they typically require users to copy-paste articles manually, operate outside the user's natural reading flow, and offer limited customization (such as summary format or style). Additionally, mind map generation, a powerful aid for cognitive mapping and understanding complex articles is missing from most mainstream summarization tools. Existing conversational systems either lack contextual grounding in the specific article the user is reading or require manual feeding of content to initiate question-answering [14]. This disjointed interaction increases cognitive load and interrupts the seamless reading experience that modern users expect.

Tools like Flipboard [2] and BeeLine Reader [7] offer content curation and reading support, but they are static in nature. They do not monitor real-time behavioral signals such as scrolling patterns, zooming actions, or engagement duration to dynamically adapt the user interface on-the-fly. Research emphasizes that real-time behavioral adaptations can significantly enhance usability and engagement [6], but practical implementations in browser-based environments are still limited.

Accessibility-focused tools like Microsoft Immersive Reader [3] primarily allow manual adjustments of font size, color schemes, and contrast settings. However, users with specific vision impairments (such as color blindness, short-sightedness, or distance vision issues) must repeatedly configure these settings themselves. Modern approaches using machine learning (such as XGBoost classifiers [16]) show potential for intelligent accessibility personalization, yet they remain underutilized in browser-native interfaces.

Most current solutions treat summarization, interactive questioning, accessibility, and UI adaptation as separate workflows [12][6], forcing users to juggle multiple tools, platforms, or manual settings. This fragmented experience increases cognitive burden, discourages usage, and limits inclusivity.

1.4 Research Objectives

1.4.1 Main Objective

The primary objective of this research is to design, develop, and evaluate a comprehensive Chrome extension that delivers an intelligent, user-centric reading experience on the Medium platform by seamlessly integrating three major personalization strategies: content-based UI personalization, behavior-driven UI personalization, and feedback-based UI personalization for visually impaired users. The system aims to reduce cognitive load, enhance accessibility, and promote deeper user engagement by enabling multi-format article summarization, chatbot-driven interactive learning, and mind map-based visual comprehension. Simultaneously, it dynamically adapts the user interface based on real-time behavioral analysis (such as scrolling speed, zooming actions, and content engagement patterns) and applies machine learning-driven theme adjustments to support users with specific vision impairments like color blindness and short-sightedness. By combining adaptive natural language processing, behavioral intelligence, and accessibility-focused personalization into a unified platform, the project strives to create a holistic, inclusive, and intelligent digital reading environment that evolves fluidly with individual user needs.

1.4.2 Specific Objectives

The following are the sub-objectives of conducting this research:

- Provide an intelligent, real-time personalization system integrated directly into the Medium reading environment.
- Implement multi-format summarization (pointwise, short, long) to support different user comprehension needs.
- Enable chatbot-driven interactive learning by allowing users to ask contextual questions about articles and receive real-time, relevant responses.
- Generate automatic mind maps from article content to visually enhance users' understanding of complex information structures.
- Adapt user interfaces dynamically based on real-time behavioral signals such as scrolling speed, zooming, and interaction patterns.
- Implement a machine learning-driven theme recommendation system to personalize UI settings (font size, background color, zoom level) for visually impaired users.
- Enhance user accessibility, reduce cognitive load, and create a more inclusive browsing experience through adaptive UI strategies.
- Collect user feedback and behavior data to continuously refine personalization models and enhance future user experiences.

1.4.3 Business Objectives

- **Improve Content Engagement and Reading Efficiency**: Enhance the reading experience on Medium by providing users with quick summaries, interactive chatbots, and mind maps that allow faster comprehension, improving user satisfaction and platform engagement.
- **Enhance User Accessibility and Inclusivity**: Offer dynamic UI adaptations and customized screen themes for visually impaired users, expanding accessibility and promoting inclusivity for a wider, diverse audience.
- **Reduce Information Overload and Cognitive Fatigue**: Through intelligent summarization and adaptive UI designs, help users consume large amounts of content more efficiently and with less cognitive effort.
- **Establish Competitive Advantage in Browser Extensions and Accessibility Tools**: Position the system as an innovative Chrome extension that combines AI, behavioral analytics, and accessibility personalization, offering unique, intelligent web experiences beyond standard summarization or reading tools.
- **Enable Future Commercialization and Platform Integration Opportunities**: Lay the foundation for monetization models (premium features, accessibility solutions licensing) and potential partnerships with content platforms such as Medium or education-oriented services by providing scalable, intelligent personalization technology.

2. METODOLOGY

2.1 Methodology

Methodology in research refers to the systematic set of techniques and strategies used to plan, design, implement, and analyze a research project. It ensures the research is conducted in a structured, rigorous, and replicable manner, leading to valid and credible results. For this project, a well-defined methodology guided the selection of development tools, testing techniques, and deployment strategies, supporting the effective realization of all three components of the system.
In this thesis, an Agile methodology was adopted, using a seven-stage development framework to develop the integrated system for content-based summarization and personalization, dynamic UI adaptation through user behavior, and accessibility-driven UI personalization for visually impaired users. Agile's iterative, feedback-driven approach was ideal given the evolving technologies involved, including AI, natural language processing (NLP), machine learning (ML), and user interface (UI) engineering.

1. Agile Methodology for Research Development

Agile methodology was selected for its adaptability, focus on collaboration, and incremental development style, all of which are critical when managing complex systems that integrate multiple AI-driven and behavior-driven personalization strategies.

The project was divided into multiple sprints, each focusing on implementing and refining a major module:

- Content-based summarization, chatbot, and mind map generation.
- Real-time behavior-based dynamic UI personalization.
- Vision-based accessibility UI adaptation for visually impaired users.

Each sprint involved regular team meetings, backlog prioritization, testing, and user feedback collection to guide continuous improvements. This allowed the system to evolve iteratively with maximum responsiveness to technical challenges and user expectations.

SCRUM MASTER
24 HOURS
PRODUCT OWNER
REVIEW
IMPLEMENTATION
2-4 WEEKS
RETROSPECT
USER STORIES
SPRINT BACKLOG
PLANNING
DEFINITION OF DONE

Figure 1: Agile Scrum Framework

2. Seven-Stage Development Framework

The project followed a structured seven-stage development lifecycle:

- Requirement Gathering: Conducted user surveys, stakeholder interviews, and literature reviews to gather requirements for content summarization, chatbot interactions, mind map generation, dynamic UI adaptation based on user behavior, and personalized UI themes for visually impaired users.
- System Design: Designed a modular architecture integrating three primary components:
  - A React.js Chrome extension frontend with tabs for summaries, chatbot interactions, and mind map visualization, dynamic UI personalization through user behavior.
  - A FastAPI backend for summarization, chatbot, and mind map generation.
  - A Flask API and XGBoost model for vision theme prediction.
  - Backend services incorporated OpenAI APIs, FAISS vector search, TF-IDF, and spaCy NLP pipelines.
- Development (Content-Based Personalization): Implemented summarization (pointwise, short, long), chatbot (semantic Q&A using RAG techniques), and mind map generation using dependency parsing (spaCy) and visualization via react-flow-renderer.

- Development (Behavior-Based Personalization): Developed modules to capture user behavior metrics (scrolling speed, zoom level, time spent, content engagement) and dynamically adjust UI settings such as font size, layout density, and contrast based on real-time behavior patterns.
- Development (Accessibility-Based Personalization): Built a Flask-based backend powered by an XGBoost classifier trained on vision-related attributes to recommend and apply the best personalized screen theme dynamically, improving accessibility for users with vision impairments.
- Testing: Unit tests, integration tests, and system testing were conducted across all components. Manual testing validated usability and personalization effectiveness. Real-world testing was performed on Medium articles for comprehensive validation.
- Deployment: The Chrome extension was deployed on a cloud-hosted backend (Azure) ensuring real-time data processing and API integration. Post-deployment and analytics were collected post-deployment for further system refinements.

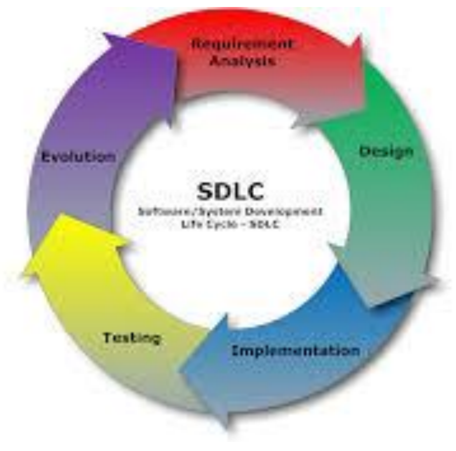


Figure 2: Software Development Life Cycle

3. Iterative Development and Collaboration

The development process was highly iterative, with each major feature being built incrementally over multiple sprints. Early sprints focused on setting up the summarization pipeline, chatbot embeddings, and basic behavior monitoring mechanisms. Subsequent sprints expanded functionalities to include mind map generation, dynamic UI recalibration, and vision-based theme recommendation. Regular sprint reviews ensured team synchronization, early bug detection, and user-driven refinements. Collaboration between frontend and backend developers allowed seamless integration of personalized features into a cohesive, user-friendly Chrome extension.

4. **Flexibility and Responsiveness**

Agile's flexibility allowed quick pivots whenever technical or functional challenges arose:

- For example, early behavior detection models were enhanced by adding threshold-based UI adaptation mechanisms to better handle scrolling and zooming irregularities.
- Mind map quality was improved by switching from simple key phrase extraction to dependency-based SVO relationship mapping.
- Accessibility model performance was boosted by applying SMOTE techniques to balance the training dataset for vision theme prediction.

Technical risks like API rate limits, NLP inaccuracies, and UX inconsistencies were addressed promptly through iterative testing and development, ensuring smooth system evolution without project derailments.

2.1.1 Feasibility Study/ Planning

A feasibility study assesses the practicality and viability of a proposed project before full-scale implementation. It determines whether the project is technically, economically, legally, operationally, and socially feasible within the given timeframe. This research project focused on the development of a comprehensive Chrome extension integrating content-based summarization, chatbot, mind map generation, dynamic UI adaptation based on user behavior, and accessibility-focused UI personalization for visually impaired users.

The feasibility of the system was evaluated across multiple dimensions, as detailed below:

- **Technical Feasibility**

The proposed system was found to be highly technically feasible due to the availability and maturity of the technologies used for all three components. The frontend was built using React.js, supporting dynamic UI rendering, real-time user behavior tracking, and seamless Chrome extension integration. The backend was implemented using FastAPI for summarization, chatbot, and mind map services, and Flask for the accessibility-focused vision theme recommendation module. OpenAI's GPT-3.5 API was used for high-quality summarization and chatbot responses. FAISS was utilized for fast vector similarity search based on TF-IDF and MiniLM embeddings. spaCy and newspaper3k powered the mind map generation through subject-verb-object (SVO) extraction. XGBoost, along with SMOTE and scikit-learn, were used for developing the vision theme prediction model for visually impaired users. Frontend theme adjustments based on the accessibility model's prediction were handled dynamically through JavaScript and CSS manipulation. All selected technologies are well-documented, open-source (or affordable), scalable, and compatible with modern web platforms, ensuring the robustness and extensibility of the solution. Browser compatibility, backend performance, API responsiveness, and the adaptive UI behaviors were tested extensively to ensure smooth operation on devices supporting Google Chrome.

- **Economic Feasibility**

The project was economically feasible. Most technologies used were free or open-source which are React.js, FastAPI, Flask, scikit-learn, spaCy, FAISS, and react-flow-renderer were used at no licensing cost. The main cost was associated with OpenAI API usage. Token consumption was minimized using prompt optimization, chunking strategies for article embeddings, and controlling summary lengths. The mind map feature and behavior-based UI adaptation required no extra costs, leveraging only free libraries. The vision theme prediction model for visually impaired users was developed using a self-collected or open dataset and hosted via cost-effective Flask APIs. The solution targets a broad audience (Medium readers, users needing accessibility support), offering opportunities for future monetization via premium versions (e.g., advanced mind map exports, pro user behavior profiles, or customizable accessibility themes). Development and deployment costs during the undergraduate project phase remained minimal, with strong potential for commercial scalability later.

- **Legal and Ethical Feasibility**

The system adheres to all necessary legal and ethical standards. No unauthorized user data collection was performed. All summarization, chatbot conversations, and mind map generations occur within the user's currently active Medium article, ensuring no breach of copyright or unauthorized scraping. User behavior metrics are collected anonymously and are used exclusively for real-time UI adjustments without storing sensitive personal data. The vision-based accessibility system asks for explicit consent when gathering user-specific vision attributes. Data storage (such as user feedback) is secured and handled in compliance with privacy best practices. Third-party API usage (OpenAI, FAISS) follows respective service agreements. Ethical considerations regarding accessibility and inclusion are fully respected by offering tailored reading experiences for users with vision impairments.

- **Operational Feasibility**

The system requires minimal technical expertise for users to operate. Users simply install the Chrome extension and can access content summarization, chatbot interaction, mind map visualization, dynamic behavior-based UI adjustments, and accessibility theme recommendations with a few clicks. Backend services automate article parsing, summarization, chatbot Q&A, mind map generation, and vision theme prediction without user intervention. The behavior monitoring is unobtrusive and real-time. Personalized accessibility themes are applied instantly without manual configuration, improving usability for visually impaired users. Operationally, the system demonstrates a high degree of usability, stability, and adaptability for long-term deployment and scaling.

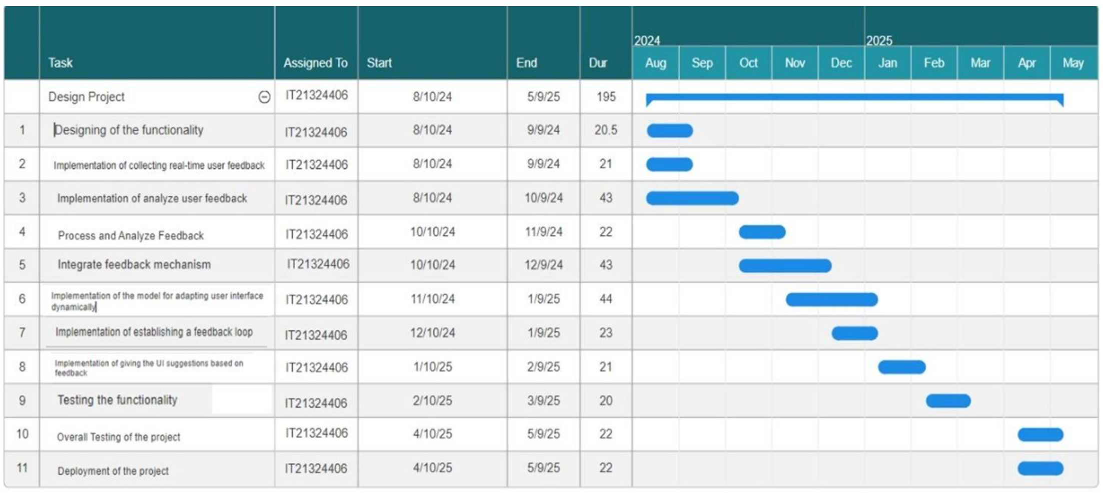
- **Time/Schedule Feasibility**

The project was successfully completed within the academic timeline, following an Agile methodology with clearly defined sprints. Separate development tracks for content personalization, behavior-based personalization, and accessibility-based personalization allowed parallel progress. Weekly standups, sprint reviews, and user testing cycles ensured adherence to deadlines. GitHub version control and modular backend/frontend design enabled smooth integration of all components. Mid-project adjustments, such as improving mind map clarity and refining UI adaptation thresholds based on user feedback, were handled without major project delays. Final testing and real-world user evaluations confirmed that all features performed reliably in the expected timeframe.

- **Social and Cultural Feasibility**

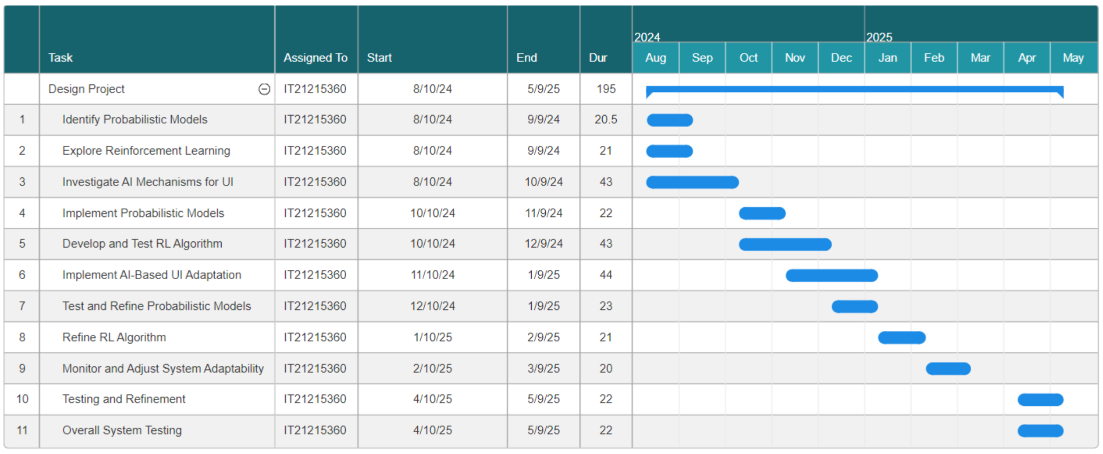
The system is socially and culturally feasible, addressing growing digital reading trends, accessibility needs, and personalized user experiences. In Sri Lanka (the primary survey region), 73% of users found the concept highly useful, particularly emphasizing interest in summarized content, customizable UI, and accessibility enhancements. The behavior-based UI personalization ensures inclusivity for users with varied reading speeds and content interaction patterns. The accessibility module supports users with vision impairments, a critical social inclusion aspect. Future multilingual support is planned, allowing broader cultural relevance. No cultural taboos or legal risks were encountered in the system's design or functionality.

Overall, the system aligns with the global movement toward intelligent, user-centric, inclusive digital experiences.



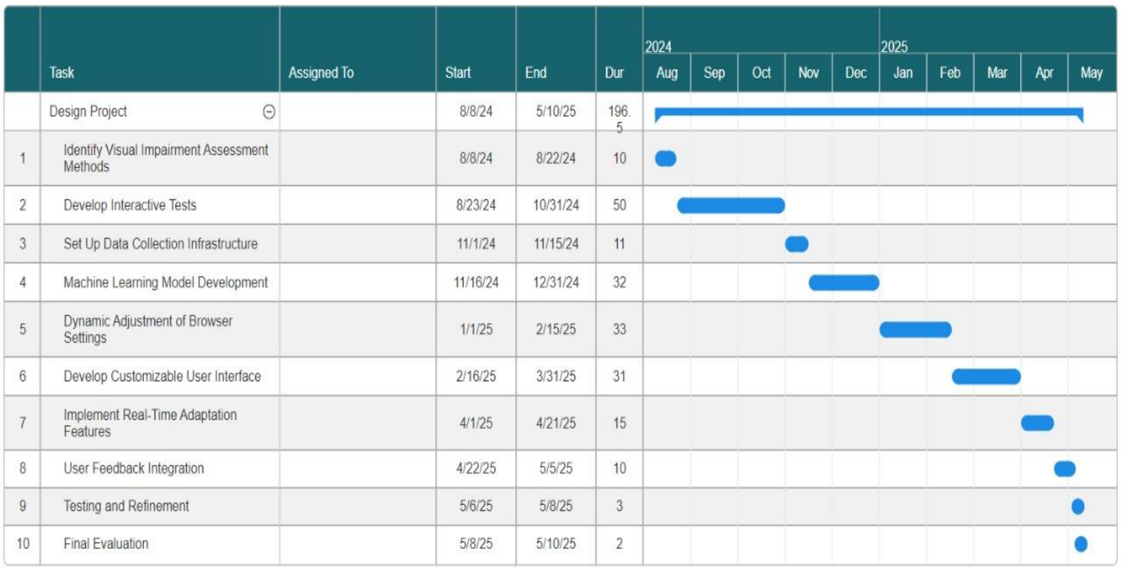
	Task	Assigned To	Start	End	Dur
	Design Project	IT21324406	8/10/24	5/9/25	195
1	Designing of the functionality	IT21324406	8/10/24	9/9/24	20.5
2	Implementation of collecting real-time user feedback	IT21324406	8/10/24	9/9/24	21
3	Implementation of analyze user feedback	IT21324406	8/10/24	10/9/24	43
4	Process and Analyze Feedback	IT21324406	10/10/24	11/9/24	22
5	Integrate feedback mechanism	IT21324406	10/10/24	12/9/24	43
6	Implementation of the model for adapting user interface dynamically	IT21324406	11/10/24	1/9/25	44
7	Implementation of establishing a feedback loop	IT21324406	12/10/24	1/9/25	23
8	Implementation of giving the UI suggestions based on feedback	IT21324406	1/10/25	2/9/25	21
9	Testing the functionality	IT21324406	2/10/25	3/9/25	20
10	Overall Testing of the project	IT21324406	4/10/25	5/9/25	22
11	Deployment of the project	IT21324406	4/10/25	5/9/25	22

Figure 3: Gantt Chart of Content-Based UI Personalization Component



	Task	Assigned To	Start	End	Dur
	Design Project	IT21215360	8/10/24	5/9/25	195
1	Identify Probabilistic Models	IT21215360	8/10/24	9/9/24	20.5
2	Explore Reinforcement Learning	IT21215360	8/10/24	9/9/24	21
3	Investigate AI Mechanisms for UI	IT21215360	8/10/24	10/9/24	43
4	Implement Probabilistic Models	IT21215360	10/10/24	11/9/24	22
5	Develop and Test RL Algorithm	IT21215360	10/10/24	12/9/24	43
6	Implement AI-Based UI Adaptation	IT21215360	11/10/24	1/9/25	44
7	Test and Refine Probabilistic Models	IT21215360	12/10/24	1/9/25	23
8	Refine RL Algorithm	IT21215360	1/10/25	2/9/25	21
9	Monitor and Adjust System Adaptability	IT21215360	2/10/25	3/9/25	20
10	Testing and Refinement	IT21215360	4/10/25	5/9/25	22
11	Overall System Testing	IT21215360	4/10/25	5/9/25	22

Figure 4: Gantt Chart of Behavior-Based UI Personalization Component



	Task	Assigned To	Start	End	Dur
	Design Project		8/8/24	5/10/25	196.5
1	Identify Visual Impairment Assessment Methods		8/8/24	8/22/24	10
2	Develop Interactive Tests		8/23/24	10/31/24	50
3	Set Up Data Collection Infrastructure		11/1/24	11/15/24	11
4	Machine Learning Model Development		11/16/24	12/31/24	32
5	Dynamic Adjustment of Browser Settings		1/1/25	2/15/25	33
6	Develop Customizable User Interface		2/16/25	3/31/25	31
7	Implement Real-Time Adaptation Features		4/1/25	4/21/25	15
8	User Feedback Integration		4/22/25	5/5/25	10
9	Testing and Refinement		5/6/25	5/8/25	3
10	Final Evaluation		5/8/25	5/10/25	2

Figure 5: Gantt Chart of Accessibility UI Personalization Component

Other than these feasibility studies, the risk management plan and communication management plan have been done.

- **Risk Management Plan**

Risk management is an essential component of any software development process, aimed at identifying, assessing, and mitigating potential threats that could impact the successful delivery of the project. For this system, which integrates content summarization, chatbot interaction, mind map generation, behavior-driven UI personalization, and vision-based UI adaptation, a comprehensive risk management plan was devised. The goal of this plan is to outline proactive strategies to handle unforeseen challenges and ensure that the development team can continue progress

with minimal disruptions across all three components. By anticipating and preparing for common technical, operational, and integration risks such as OpenAI API rate limits, user behavior data misinterpretation, accessibility adaptation inaccuracies, parsing inconsistencies in Medium articles, ML model prediction errors, or frontend rendering issues the team-maintained control over the project lifecycle. This structured approach significantly contributed to the successful implementation of the full ReactiveWeb Chrome extension with all its personalization features.
Table 3 presents the detailed risk management plan

Table 3: Risk Management Plan

Risk	**Trigger**	**Owner**	**Response**	**Resource Required**
Risk with respect to the Project Team				
Illness or sudden absence of the project team member(s)	Illness / Other personal emergencies	Project Leader	• Inform to the supervisor and co-supervisor. • * Development team divides the functions with equal scope.	• Project Schedule Plan/Gantt Chart • Backup resources
Risk with respect to the Panel/ Supervisor(s)				
Panel Requests changes	Not satisfied with the product/presentation/ outcome	Project Leader	• Do the necessary changes immediately. • Update the changes in all required documents. • Update the changes to the required persons.	• Project Schedule Plan/Gantt Chart • Product Backlog • Meeting Log
Supervisor(s) Request changes	Not satisfied with the product/presentation/ outcome	Project Leader		

Panel/Supervisor(s) is not at the scheduled meetings	Illness / Other personal emergencies	Project Leader	• Inform it to the required persons immediately. • Reschedule the meeting/ do necessary alternatives	• Meeting Log • Proper Email

- **Communication Management Plan**

Effective communication is critical to the success of any research or development project, especially when managing a multi-component system like ReactiveWeb. Throughout the development of the Chrome extension, which integrates summarization, chatbot, mind map generation, behavior-driven UI adaptation, and accessibility-focused personalization, a structured communication plan was meticulously followed. This ensured smooth collaboration among team members, supervisors, and stakeholders, minimizing misunderstandings and enabling timely delivery of project milestones across all three major modules.

✓ **Communication Objectives:**

- To ensure all stakeholders, including supervisors and team members, are continuously informed about the project’s status, challenges, and progress.
- To enable smooth and effective coordination among developers working on different components (content personalization, behavior-driven adaptation, and vision-based UI personalization).
- To provide timely updates and feedback loops with supervisors, ensuring that any concerns or requirements are addressed early.
- To document and archive all key decisions, technical changes, sprint outcomes, and architectural updates throughout the research and development lifecycle.
- To facilitate the early identification and collaborative resolution of conflicts, risks, or delays, avoiding bottlenecks.

- To maintain transparency and traceability across all project activities, ensuring accountability for deliverables.

✓ **Communication Media:**

The communication media that will be used for the project are:

- Email (Microsoft Outlook) - for formal communications, progress reports, and milestone updates.
- Chat Applications (WhatsApp, MS Teams) - for daily check-ins, clarifications, and immediate issue reporting.
- Video Conferencing (MS Teams, Google Meet) - for weekly sprint reviews, supervisor meetings, and team collaboration discussions.
- Shared Documents (Google Docs, Microsoft OneDrive) - for collaborative documentation of project requirements, testing reports, and user feedback analysis.
- Project Management Tools (Trello, Microsoft Planner) - for task assignment, progress tracking, and backlog management.
- Version Control Platforms (GitHub) - for code collaboration, issue tracking, and maintaining consistent project versions across frontend and backend teams.

Table 4: Communication Management Plan

Communication Media	Purpose	Frequency	Participants	Description
Email	Formal updates, documentation sharing	As needed (at least weekly)	Supervisor, Co-supervisor, Team Members	Used for sharing project status reports, meeting summaries, and formal communication
Weekly Meetings	Status updates, issue resolution	Weekly	Supervisor, Co-supervisor, Project Team	Scheduled virtual or physical meetings to discuss progress, address issues,

				and plan upcoming tasks
Project Management Tools (e.g., Trello)	Task tracking and progress monitoring	Continuous	Project team	Enables tracking of project tasks, deadlines, and deliverables, ensuring the team stays on schedule
Shared Drive/Cloud Storage (e.g., Google Drive, OneDrive)	Document storage and sharing	Continuous	Project team	Centralized repository for storing project documents, code, designs, and other important files
GitHub/Git	Version control and code sharing	Continuous	Project team	Used for managing code versions, collaborative development, and code reviews.

2.1.2 Requirement Gathering & Analysis

The Requirement Gathering and Analysis phase was critical in shaping the integrated system, ensuring it effectively addressed the personalized content consumption, dynamic UI adaptation, and accessibility enhancement goals. This phase involved collecting detailed insights through surveys, stakeholder interviews, user observations, and literature review, enabling the identification of precise functional and non-functional requirements for each major component which are Content-Based UI Personalization, User Behavior-Based UI Personalization, and Accessibility-Based UI Personalization.

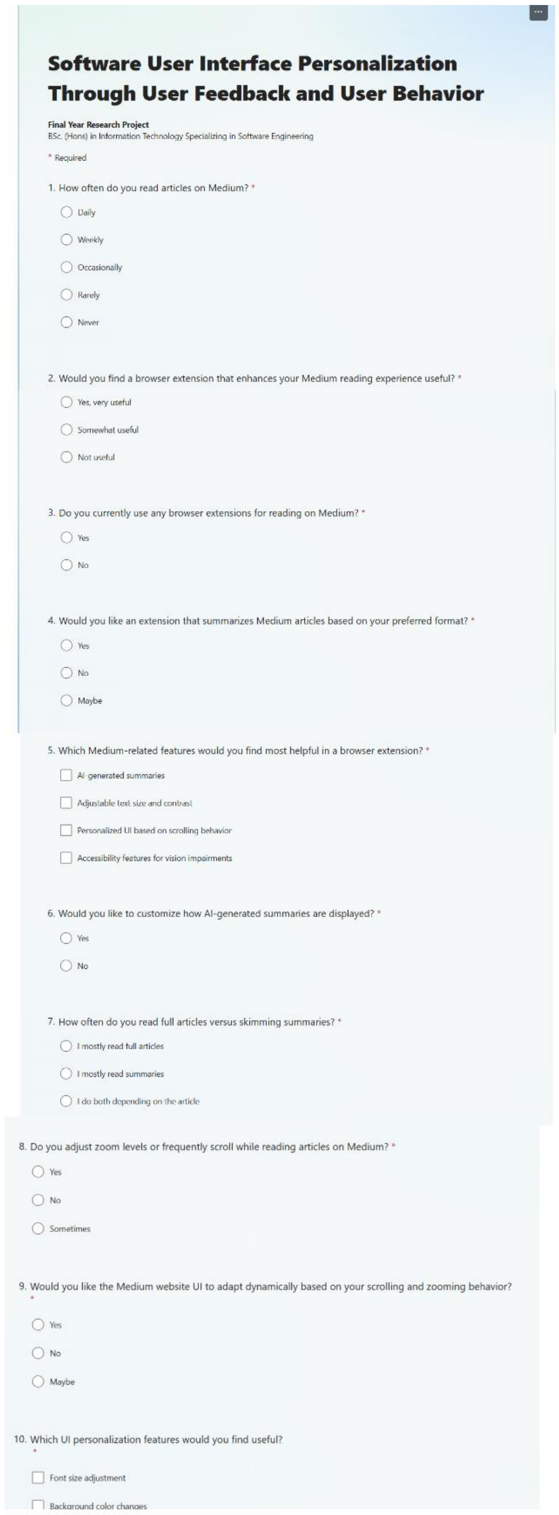
2.1.2.1. Functional Requirements

The proposed system incorporates several critical functional requirements designed to deliver a highly personalized and adaptive reading experience for Medium article readers, catering to users' cognitive, behavioral, and accessibility needs. The system seamlessly integrates personalized content summarization, an intelligent chatbot, a visual mind map generator, dynamic UI adaptation based on user behavior, and vision-based UI personalization for visually impaired users. Each function is carefully crafted to promote engagement, comprehension, and accessibility.

To understand users’ needs and expectations, an online survey was conducted among around 50 participants, primarily Medium.com users and individuals requiring reading support (e.g., visually impaired readers). Key insights gathered include:

- **73%** of users expressed the need for article summarization options (bullet points, short, long).
- **68%** showed strong interest in a chatbot capable of answering article-specific questions.
- **65%** preferred mind maps for easier information comprehension.
- **58%** highlighted the necessity of dynamic UI adaptation based on their reading habits (scrolling speed, reading focus).
- **42%** of users, including visually impaired participants, indicated that default web interfaces lacked adequate customization for their vision needs.

This feedback directly influenced system requirements, ensuring a user-centric design across all modules.



Software User Interface Personalization Through User Feedback and User Behavior

Final Year Research Project
BSc. (Hons) in Information Technology Specializing in Software Engineering

* Required

1. How often do you read articles on Medium? *
   - ○ Daily
   - ○ Weekly
   - ○ Occasionally
   - ○ Rarely
   - ○ Never

2. Would you find a browser extension that enhances your Medium reading experience useful? *
   - ○ Yes, very useful
   - ○ Somewhat useful
   - ○ Not useful

3. Do you currently use any browser extensions for reading on Medium? *
   - ○ Yes
   - ○ No

4. Would you like an extension that summarizes Medium articles based on your preferred format? *
   - ○ Yes
   - ○ No
   - ○ Maybe

5. Which Medium-related features would you find most helpful in a browser extension? *
   - ☐ AI generated summaries
   - ☐ Adjustable text size and contrast
   - ☐ Personalized UI based on scrolling behavior
   - ☐ Accessibility features for vision impairments

6. Would you like to customize how AI-generated summaries are displayed? *
   - ○ Yes
   - ○ No

7. How often do you read full articles versus skimming summaries? *
   - ○ I mostly read full articles
   - ○ I mostly read summaries
   - ○ I do both depending on the article

8. Do you adjust zoom levels or frequently scroll while reading articles on Medium? *
   - ○ Yes
   - ○ No
   - ○ Sometimes

9. Would you like the Medium website UI to adapt dynamically based on your scrolling and zooming behavior? *
   - ○ Yes
   - ○ No
   - ○ Maybe

10. Which UI personalization features would you find useful? *
    - ☐ Font size adjustment
    - ☐ Background color changes

Figure 6: Survey Form

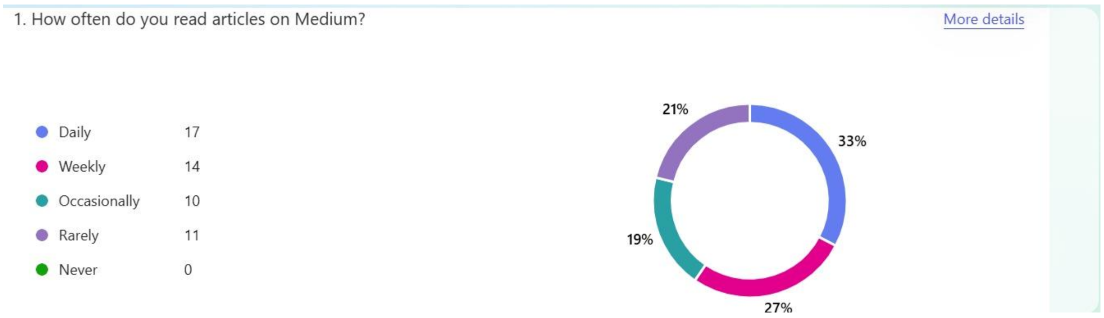


Figure 7: User's Medium Article Preference

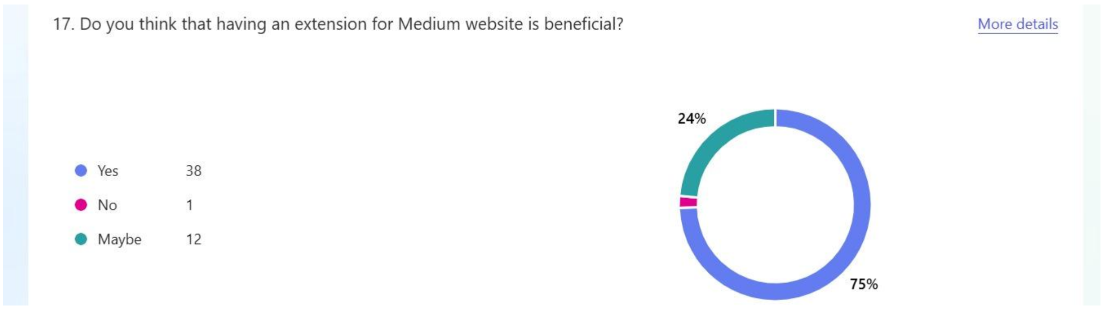


Figure 8: User's feedback on the Extension

Users are enabled to generate summaries in three different formats bullet points, short paragraphs, or long summary according to their individual reading preferences. Users click the "Generate Summary" button on the Chrome extension, which extracts the current Medium article content and sends it to the backend built on FastAPI. The backend leverages OpenAI's GPT-3.5 model to produce contextually accurate summaries and returns them to the frontend for display. The extension features a "Chat" tab where users can ask article-specific questions. The chatbot responds with semantically accurate answers using TF-IDF embeddings, FAISS-based semantic search, and OpenAI GPT-3.5 generation. Chatbot interaction features include Copy Response, Regenerate Answer, and Read Aloud (TTS) to improve usability for diverse audiences. The "Mind Map" feature allows users to visually explore the article's structure. Using NLP techniques (spaCy dependency parsing and subject-verb-object (SVO) triplet extraction), the system identifies key concepts and their relationships. The mind map is dynamically rendered using react-flow-renderer, supporting zooming, panning, dragging nodes, and exporting as a PDF/Image.

Real-time tracking of user behaviors like scrolling speed, reading duration, and navigation flow dynamically adjusts UI parameters such as font size, content density, and layout structure. The system continuously monitors and recalibrates the interface based on implicit behavior, enhancing reading comfort without manual configuration. Visually impaired users receive a personalized screen theme based on their input about vision impairments (e.g., color blindness, short/long sightedness). A Flask-based backend processes user inputs through a trained XGBoost classifier model, predicts the best-suited theme (font size, contrast, background color), and dynamically applies it via JavaScript in real-time. Accessibility preferences and themes are stored for consistent experiences across sessions. All key user actions including summary generation, chatbot queries, mind map creations, scrolling behavior, zoom events, and vision theme applications are logged into a MongoDB database. This enables future data-driven enhancements and deeper personalization through analytics.

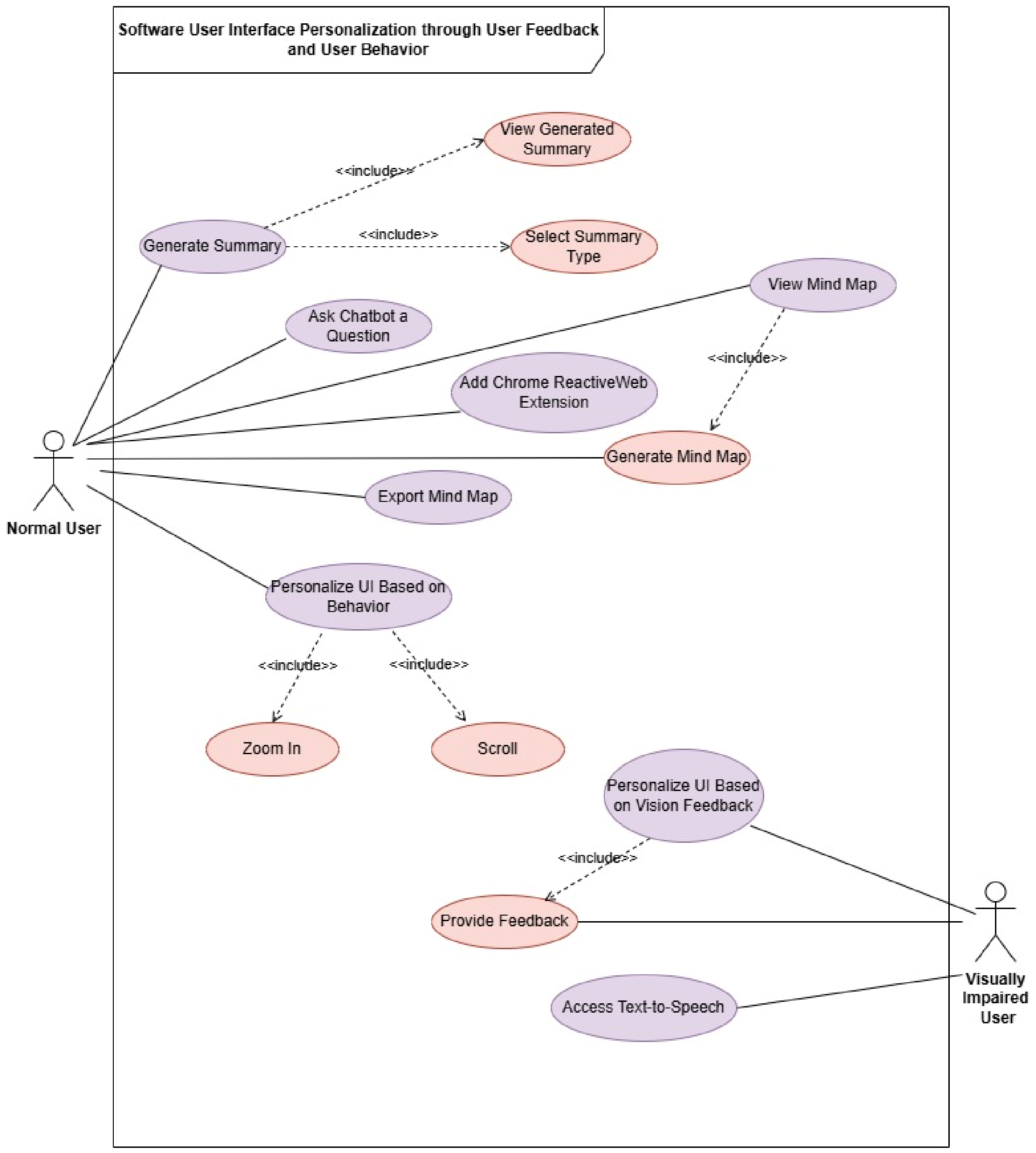
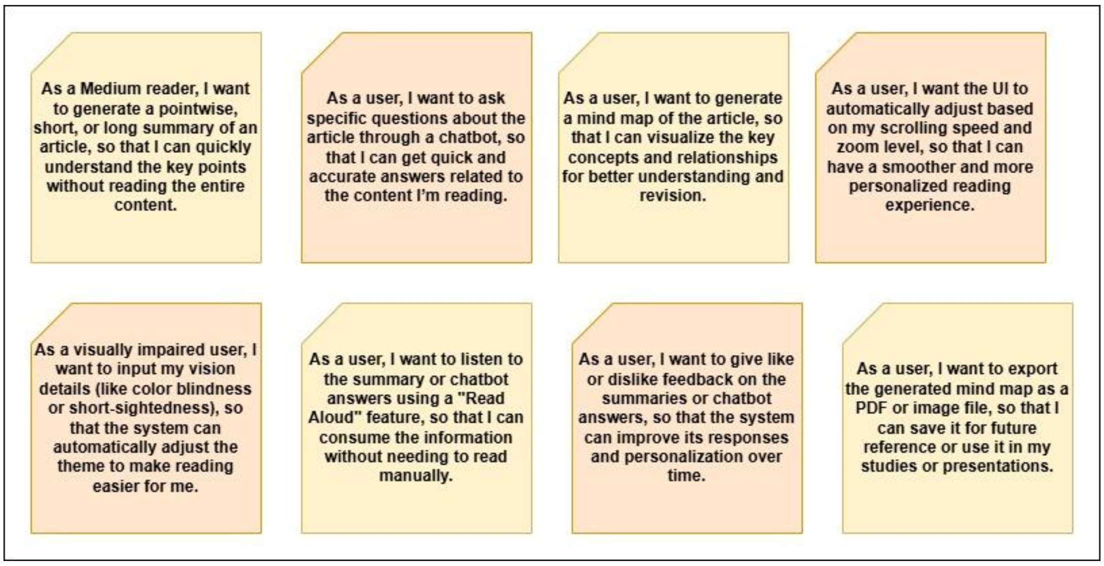


Figure 9: Use Case Diagram


As a Medium reader, I want to generate a pointwise, short, or long summary of an article, so that I can quickly understand the key points without reading the entire content.

As a user, I want to ask specific questions about the article through a chatbot, so that I can get quick and accurate answers related to the content I'm reading.

As a user, I want to generate a mind map of the article, so that I can visualize the key concepts and relationships for better understanding and revision.

As a user, I want the UI to automatically adjust based on my scrolling speed and zoom level, so that I can have a smoother and more personalized reading experience.

As a visually impaired user, I want to input my vision details (like color blindness or short-sightedness), so that the system can automatically adjust the theme to make reading easier for me.

As a user, I want to listen to the summary or chatbot answers using a "Read Aloud" feature, so that I can consume the information without needing to read manually.

As a user, I want to give like or dislike feedback on the summaries or chatbot answers, so that the system can improve its responses and personalization over time.

As a user, I want to export the generated mind map as a PDF or image file, so that I can save it for future reference or use it in my studies or presentations.

Figure 10: User Stories

2.1.2.2. Non-Functional Requirements

To ensure a seamless and high-quality user experience across all personalization modules, the proposed system adheres to several critical non-functional requirements. The summarization engine is optimized to generate summaries (pointwise, short, or long) within 3 to 5 seconds. The chatbot module provides contextually accurate responses within 5 seconds of user input. The mind map generation engine processes and renders clear visual diagrams within a few seconds after activation, maintaining minimal delay even for lengthy articles. Similarly, dynamic UI adjustments based on user behavior (scrolling, zooming) and real-time accessibility theme adaptations occur instantly without noticeable lag.

The front-end interface, built using React.js, is lightweight and highly responsive. It ensures full compatibility with all major Chromium-based browsers (such as Google Chrome, Microsoft Edge, and Brave), maintaining smooth performance without hindering the standard browsing experience on Medium articles.

All user interactions, including article parsing, summary generation, chatbot communication, and behavior monitoring, are securely transmitted over HTTPS. No sensitive user information (such as personally identifiable data) is permanently stored without consent, and security best practices are enforced across both frontend and backend components.

The system is designed to scale and handle at least 50 concurrent users without experiencing latency or service degradation. MongoDB is utilized for flexible and scalable storage, managing summaries, chatbot conversations, user behavior logs, accessibility settings, and mind map data. The UI has been carefully designed with accessibility standards in mind. Features such as adjustable font sizes, dynamic zoom controls, high-contrast themes, and text-to-speech (TTS) support are integrated, enabling a more inclusive experience for visually impaired users and users with cognitive challenges. The system follows modular architecture across both frontend and backend layers, ensuring that new features such as additional summarization styles, chatbot enhancements, expanded mind map visualizations, or new behavior-driven adaptations can be easily incorporated. This extensibility guarantees long-term maintainability and smooth future integrations into broader content platforms or educational tools.

2.1.3 Designing

Designing is a critical phase in software development that transforms user requirements into a concrete blueprint for implementation. In this research, the design process involved careful consideration of both functionality and user experience, especially since the solution is integrated as a Chrome extension aimed at Medium readers. The system architecture was planned to support three major personalization components which are **Content-Based UI Personalization** (summarization, chatbot, mind map), **Behavior-Based UI Personalization**, and **Accessibility-Based UI Personalization for Visually Impaired Users**.

The backend was designed using a modular architecture with FastAPI and Flask, allowing flexible communication between multiple components including summarization, chatbot, mind map generation, behavioral adaptation engine, and accessibility adjustment engine. Each service is loosely coupled, ensuring independent scalability and maintainability.

At the frontend, a React.js-based Chrome extension was developed with a structured tab interface:

- Summary Tab for generating article summaries in multiple formats.

- Chat Tab for interacting with an article-specific AI chatbot.
- Mind Map Tab for visualizing article concepts.
- Behavioral Personalization Layer that adjusts font size, layout, and contrast dynamically based on user behavior (scrolling, zooming, engagement time).
- Accessibility Personalization Module for applying personalized screen themes (font size, contrast, zoom) based on vision impairment prediction.

For Content-Based UI Personalization, the mind map feature was designed to extract subject-verb-object (SVO) relationships from article text using spaCy, which are then visualized with react-flow-renderer to produce an interactive, expandable mind map. The summarization and chatbot modules share the same context, ensuring seamless, coherent interaction based on the active article content.

Behavior-Based Personalization was designed to monitor real-time scrolling speed, zooming actions, and content focus times. This behavioral data triggers adaptive UI adjustments (e.g., increasing font size for slow readers, reducing clutter for fast scrollers) without requiring explicit user input.

Accessibility-Based Personalization collects user vision data (e.g., color blindness, short-sightedness) and predicts an optimal theme using a Flask backend powered by a trained XGBoost classifier. The frontend dynamically applies the personalized theme for better readability and inclusiveness.

User feedback mechanisms such as thumbs-up/down on summaries and chatbot answers, and theme satisfaction ratings for accessibility personalization, were incorporated to drive future system refinements. Security, modularity, and minimal latency were prioritized throughout the design to ensure a reliable, accessible, and user-friendly experience across all personalization dimensions.

System Architecture Diagram:

The system architecture of the proposed solution is designed to support Content-Based Personalization, Behavior-Based Personalization, and Accessibility-Based Personalization in a unified Chrome extension. It follows a modular, layered architecture consisting of:

- Frontend Layer (React.js Chrome Extension):

    - Tab-based interface: Summary, Chat, Mind Map
    - Real-time behavioral monitoring engine
    - Dynamic UI adjustment module
    - Accessibility theme application engine
    - Interaction handlers for summaries, chatbot queries, mind map generation
- Backend Layer (Microservice):
    - FastAPI Server for Summarization, Chatbot, Mind Map generation:
        - Summarization using OpenAI GPT-3.5
        - Chatbot interaction using TF-IDF + FAISS + OpenAI GPT-3.5
        - Mind Map generation using spaCy SVO extraction
    - Flask Server for Vision Theme Prediction (Accessibility Personalization):
        - XGBoost ML model serving
        - Preprocessing using Label Encoding, Standard Scaler
        - SQLite-based Theme Storage
- External Services:
    - OpenAI APIs for language understanding and generation
    - FAISS for vector similarity search
    - MongoDB for storing user data, summaries, mind map nodes, behavioral logs, and accessibility settings
    - Cloud Storage (optional) for future data scaling
- Workflow Overview:
    - The frontend extracts the article DOM content.
    - Sends data to FastAPI/Flask endpoints depending on user action.
    - Backend processes the input (summarization, embedding, NLP parsing, ML prediction).
    - Backend returns summarized text, chatbot answers, mind map data, or theme settings.
    - Frontend dynamically renders and applies results in real-time.

This architecture ensures that all three major personalization components work independently yet cohesively, providing a seamless and adaptive user experience within the browser environment. The modular design also supports easy scalability for future features such as multilingual summarization, deeper behavior-based learning models, or more complex accessibility options.

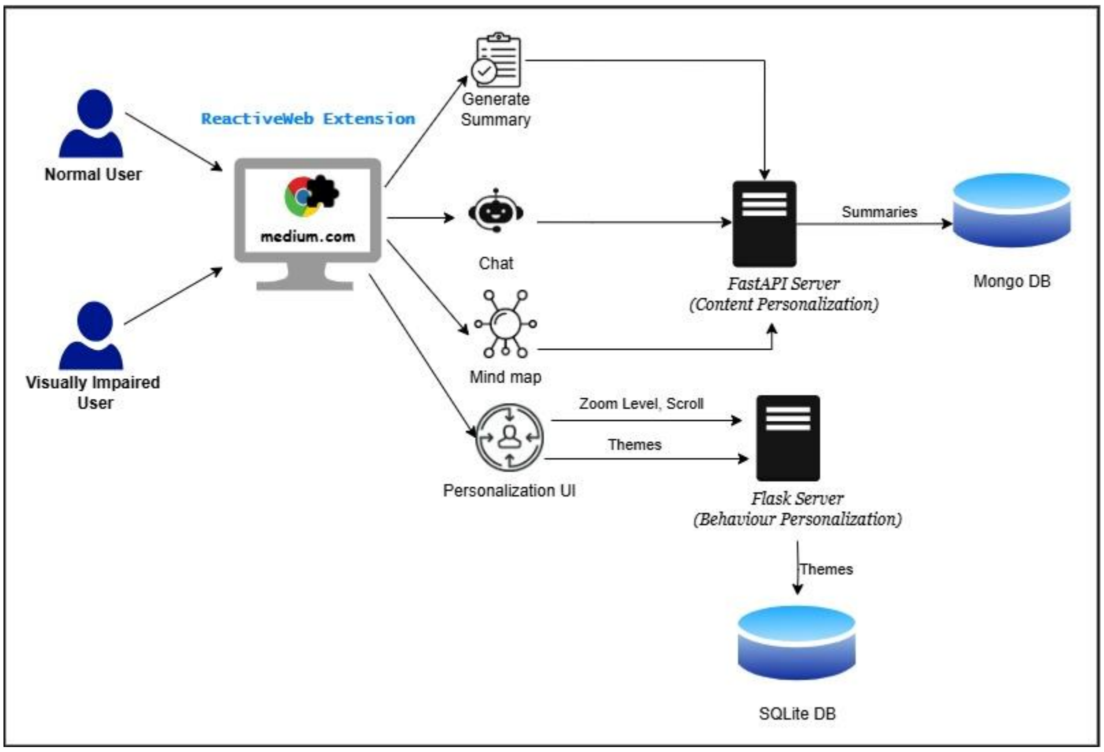


Figure 11: System Overview Diagram

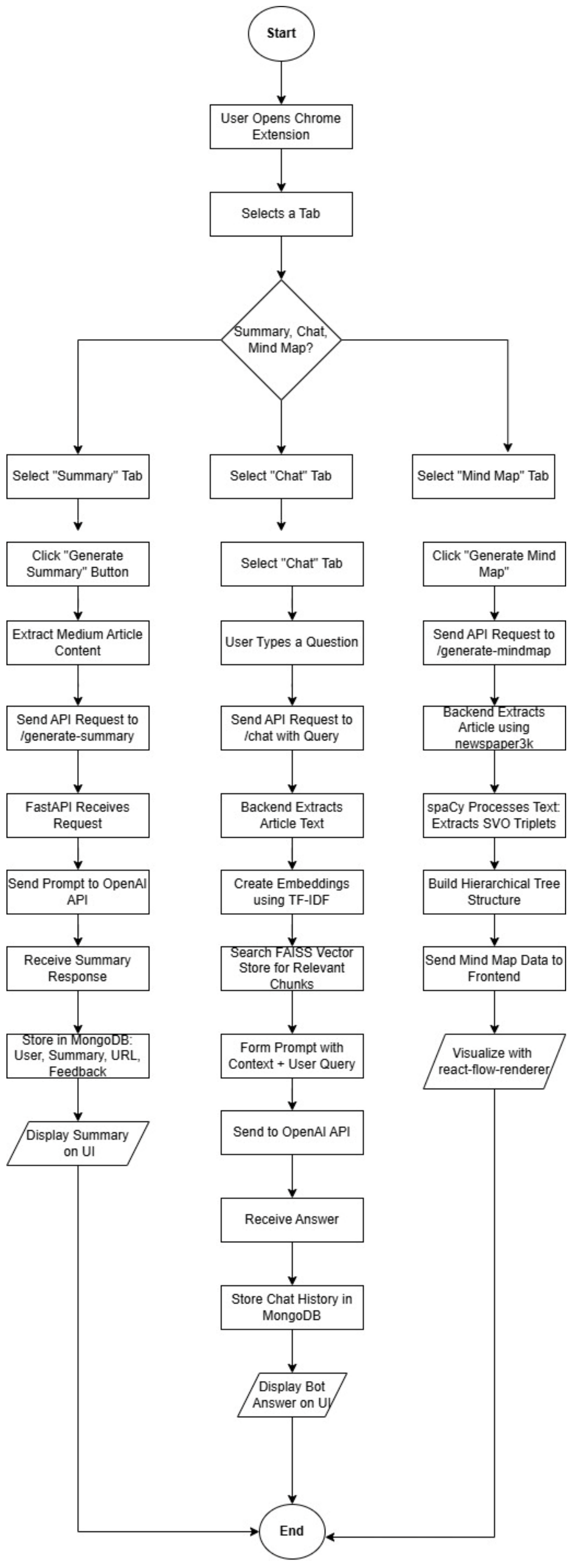

Start
User Opens Chrome Extension
Selects a Tab
Summary, Chat, Mind Map?
Select "Summary" Tab
Select "Chat" Tab
Select "Mind Map" Tab
Click "Generate Summary" Button
Extract Medium Article Content
Send API Request to /generate-summary
FastAPI Receives Request
Send Prompt to OpenAI API
Receive Summary Response
Store in MongoDB: User, Summary, URL, Feedback
Display Summary on UI
Select "Chat" Tab
User Types a Question
Send API Request to /chat with Query
Backend Extracts Article Text
Create Embeddings using TF-IDF
Search FAISS Vector Store for Relevant Chunks
Form Prompt with Context + User Query
Send to OpenAI API
Receive Answer
Store Chat History in MongoDB
Display Bot Answer on UI
Click "Generate Mind Map"
Send API Request to /generate-mindmap
Backend Extracts Article using newspaper3k
spaCy Processes Text: Extracts SVO Triplets
Build Hierarchical Tree Structure
Send Mind Map Data to Frontend
Visualize with react-flow-renderer
End

Figure 12: Flow of Content-Based UI Personalization

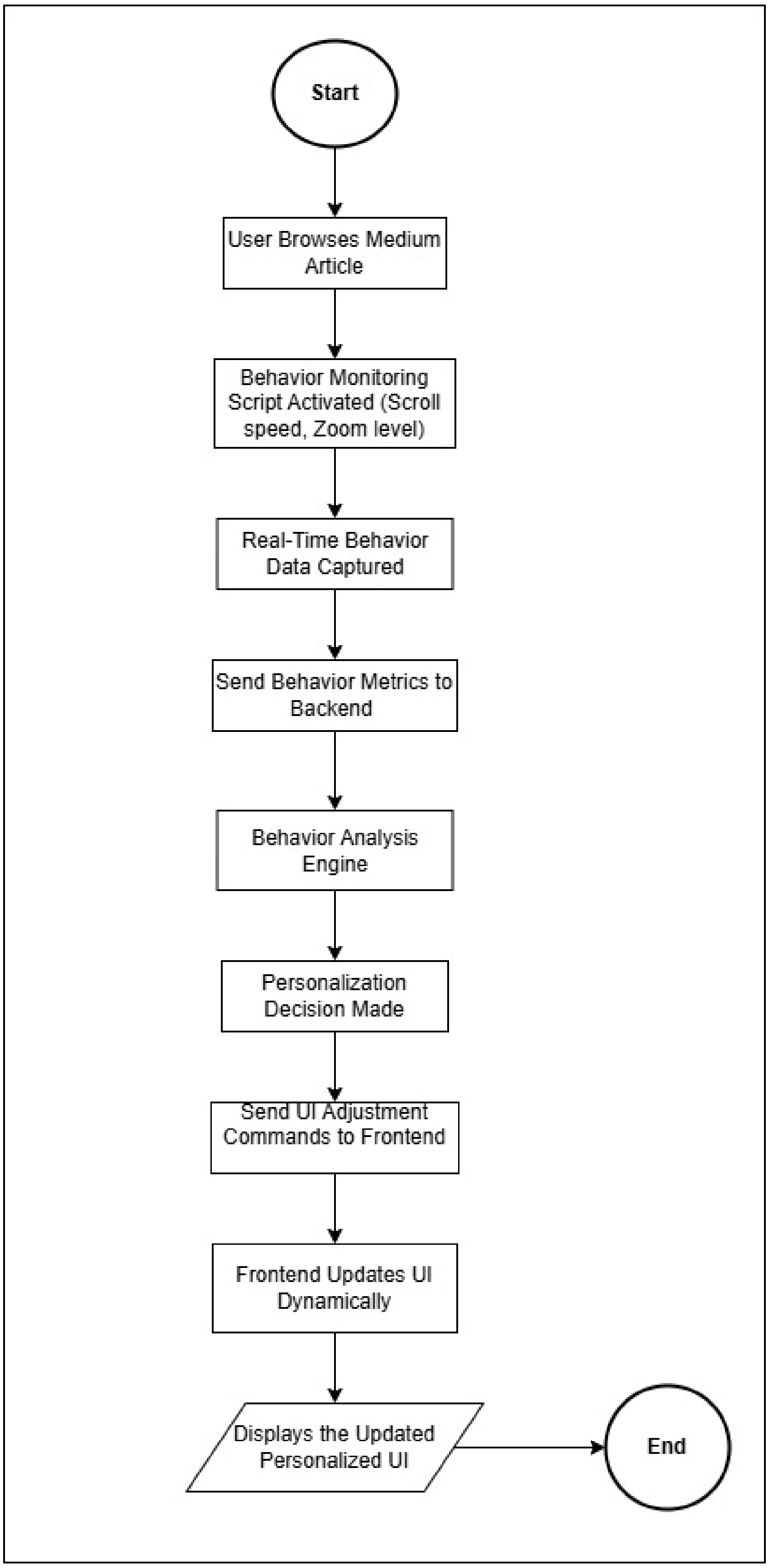

Start
User Browses Medium Article
Behavior Monitoring Script Activated (Scroll speed, Zoom level)
Real-Time Behavior Data Captured
Send Behavior Metrics to Backend
Behavior Analysis Engine
Personalization Decision Made
Send UI Adjustment Commands to Frontend
Frontend Updates UI Dynamically
Displays the Updated Personalized UI
End

Figure 13: Flow of Behavior-Based UI Personalization

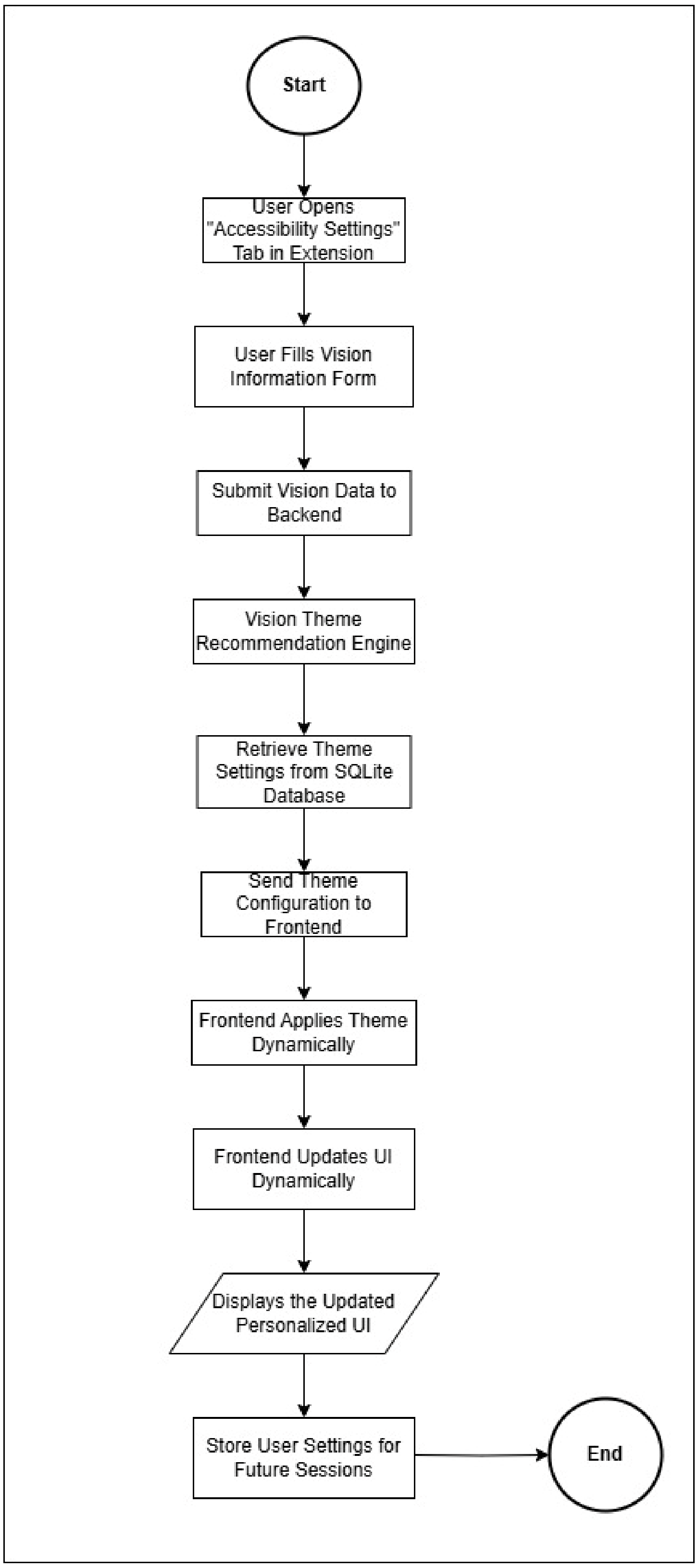

Start
User Opens "Accessibility Settings" Tab in Extension
User Fills Vision Information Form
Submit Vision Data to Backend
Vision Theme Recommendation Engine
Retrieve Theme Settings from SQLite Database
Send Theme Configuration to Frontend
Frontend Applies Theme Dynamically
Frontend Updates UI Dynamically
Displays the Updated Personalized UI
Store User Settings for Future Sessions
End

Figure 14: Flow of Accessibility feature

2.1.4 Implementation

The Implementation phase of this research project marked a pivotal transition from the design architecture to the realization of a fully functional, modular Chrome extension aimed at enhancing the reading experience on Medium through AI-powered content summarization, intelligent chatbot interaction, mind map visualization, dynamic behavioral personalization, and accessibility adaptation for visually impaired users. This phase was meticulously carried out using a structured approach involving task breakdown, sprint-based development, and the application of modern tools and frameworks such as React.js, FastAPI, Flask, XGBoost, and OpenAI APIs.

Task Breakdown and Project Management:

Prior to development, a detailed task breakdown was undertaken to ensure a clear division of work across the three major components:

- Content-Based UI Personalization (Summarization, Chatbot, Mind Map)
- Dynamic UI Personalization based on User Behavior
- Accessibility-Focused UI Personalization for Visually Impaired Users

The development was organized into three main sprints, each focusing on one of the major functional modules, while maintaining integration compatibility across all components.

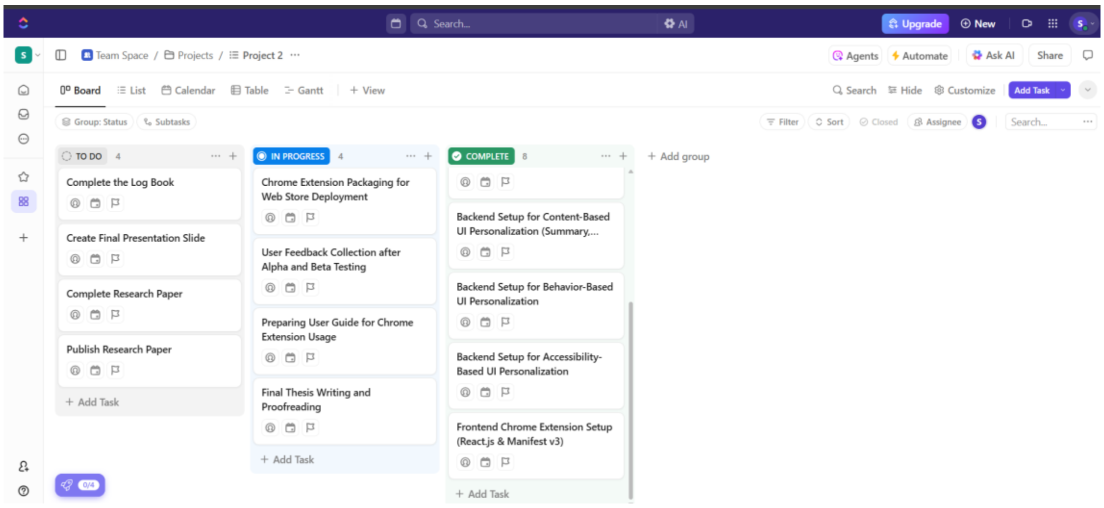
- Sprint 1: Development of the Chrome extension's frontend base structure, summary generator, chatbot system, and OpenAI integration.
- Sprint 2: Implementation of behavioral tracking and dynamic UI adaptation engine based on user actions (scrolling, zooming, reading time).
- Sprint 3: Construction of accessibility-focused UI theme recommendation through a machine learning model using user vision profiles.

Microsoft Planner was used for project management, tracking user stories, task progress, sprint goals, and milestone reviews. Each task was logged with expected deliverables, deadlines, and dependency links to other system parts, ensuring smooth collaboration and time-bound delivery.

The Work Breakdown Structure (WBS) was built around:

- Frontend (React.js) development tasks: Chrome extension setup, tab management (Summary, Chat, Mind Map, Settings).

- Backend (FastAPI, Flask) microservice setup tasks: APIs for summarization/chatbot/mind map (Server 1), dynamic behavior personalization (Server 2), accessibility personalization (Server 3).
- Integration tasks: Connecting frontend with the appropriate backend services, user state management.
- Testing and Deployment tasks: Unit, integration, and acceptance testing, Azure deployment.



TO DO 4
- Complete the Log Book
- Create Final Presentation Slide
- Complete Research Paper
- Publish Research Paper
- + Add Task

IN PROGRESS 4
- Chrome Extension Packaging for Web Store Deployment
- User Feedback Collection after Alpha and Beta Testing
- Preparing User Guide for Chrome Extension Usage
- Final Thesis Writing and Proofreading
- + Add Task

COMPLETE 8
- Backend Setup for Content-Based UI Personalization (Summary,...
- Backend Setup for Behavior-Based UI Personalization
- Backend Setup for Accessibility-Based UI Personalization
- Frontend Chrome Extension Setup (React.js & Manifest v3)
- + Add Task

Figure 15: Click Up Board of the Project

Development Environment and Tools:

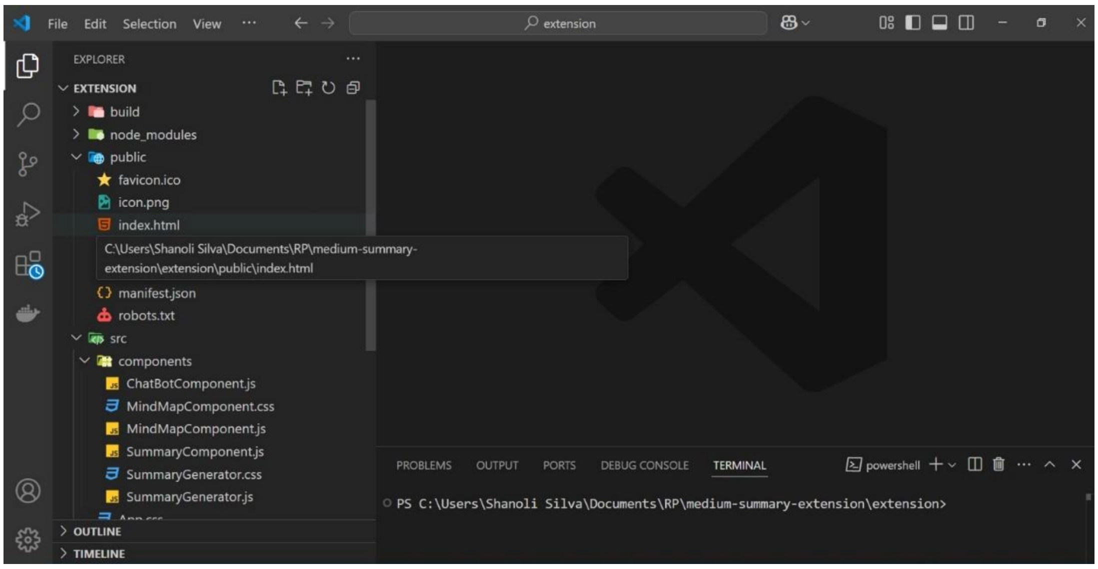
The frontend of the system was developed using React.js, which provided a dynamic, component-based structure essential for a responsive Chrome extension. The extension followed the Manifest V3 standard, which is the latest specification recommended by Google Chrome for building secure and efficient extensions. The interface was structured and styled using a combination of HTML5, CSS3, and JavaScript, ensuring lightweight rendering and smooth interactions for users browsing Medium articles. To visualize the mind map feature dynamically, react-flow-renderer was utilized, enabling hierarchical and interactive graph layouts that represented key concepts extracted from articles.

The backend architecture was developed using a microservices approach, split across three servers. The FastAPI framework powered the first microservice (Server 1), responsible for handling article content processing, summarization, chatbot query management, and mind map generation. Summarization and chatbot functionalities leveraged OpenAI's GPT-3.5 API, ensuring high-quality, context-aware text generation. For mind map creation, spaCy was employed to extract subject-verb-object (SVO) relationships, which were then structured into hierarchical formats for frontend rendering. FAISS was used for efficient semantic vector search during chatbot question retrieval, while TF-IDF and MiniLM embeddings helped embed article chunks semantically for accurate similarity matching. The third server (Server 3) hosted a Flask application that handled accessibility personalization. This microservice used a pre-trained XGBoost classifier to recommend UI themes for visually impaired users based on inputs such as vision type, age, and gender. SMOTE was applied during model training to mitigate class imbalance in the vision dataset, ensuring higher prediction accuracy and robustness. For storage, MongoDB Atlas served as the primary database solution, recording user interactions such as summary generations, chatbot queries, mind map visualizations, and behavioral engagement logs. Deployment was handled through Azure Cloud Services, where each backend microservice was deployed independently via Azure App Services, ensuring modular scaling, resilience, and security. GitHub Actions was set up for continuous integration and deployment (CI/CD), automating the process of code building, testing, and releasing.

2.1.4.1 Frontend Implementation

Frontend Implementation:

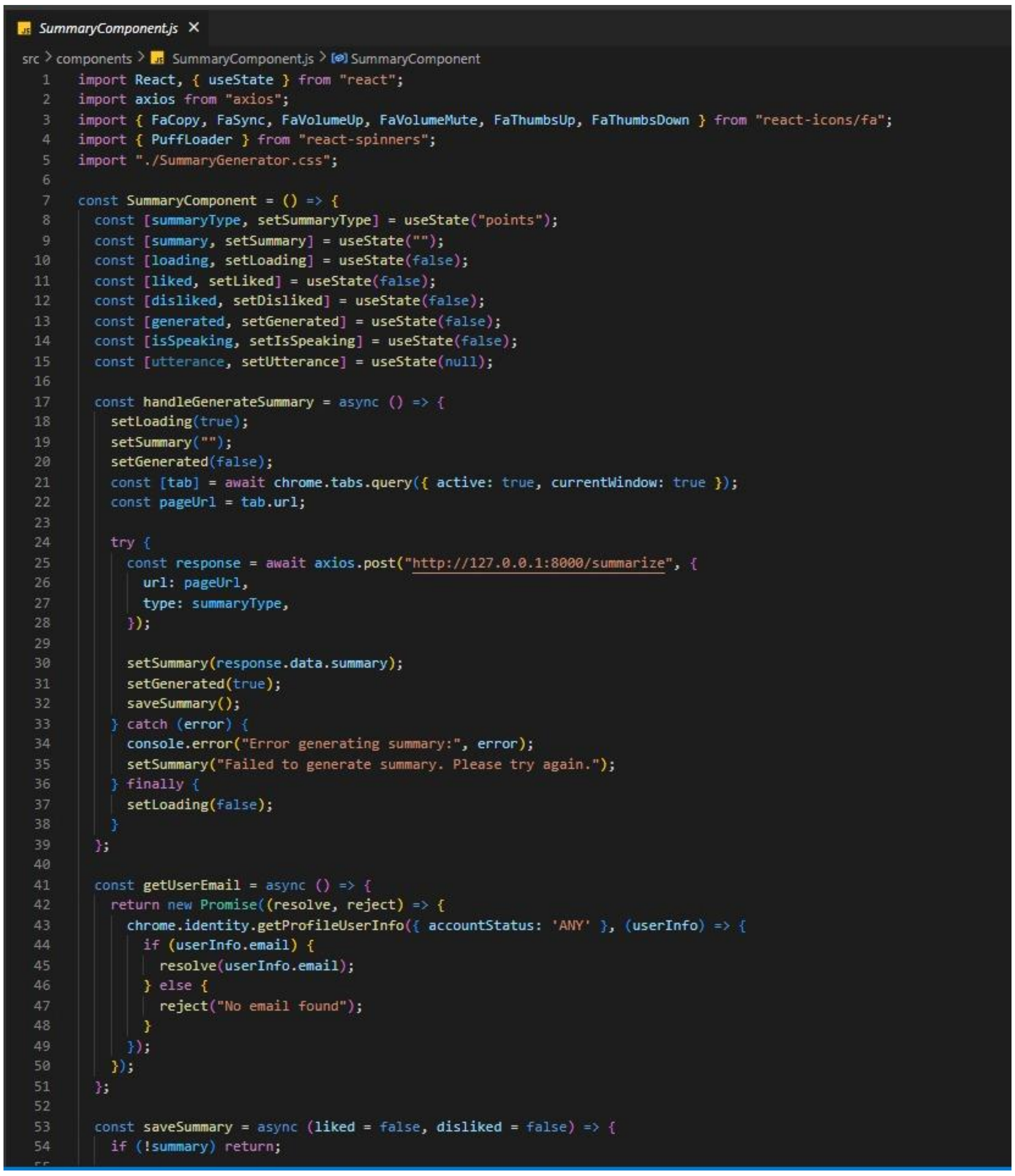
The summary generation interface was developed using React.js and integrated into the Chrome extension popup as the default tab. A dropdown menu allows users to select one of three summary formats: bullet points, short paragraph, or long form. A button labeled "Generate Summary" initiates the summarization process. The UI dynamically renders the result in a scrollable, styled box with options to like/dislike the output for feedback purposes.



C:\Users\Shanoli Silva\Documents\RP\medium-summary-extension\extension\public\index.html

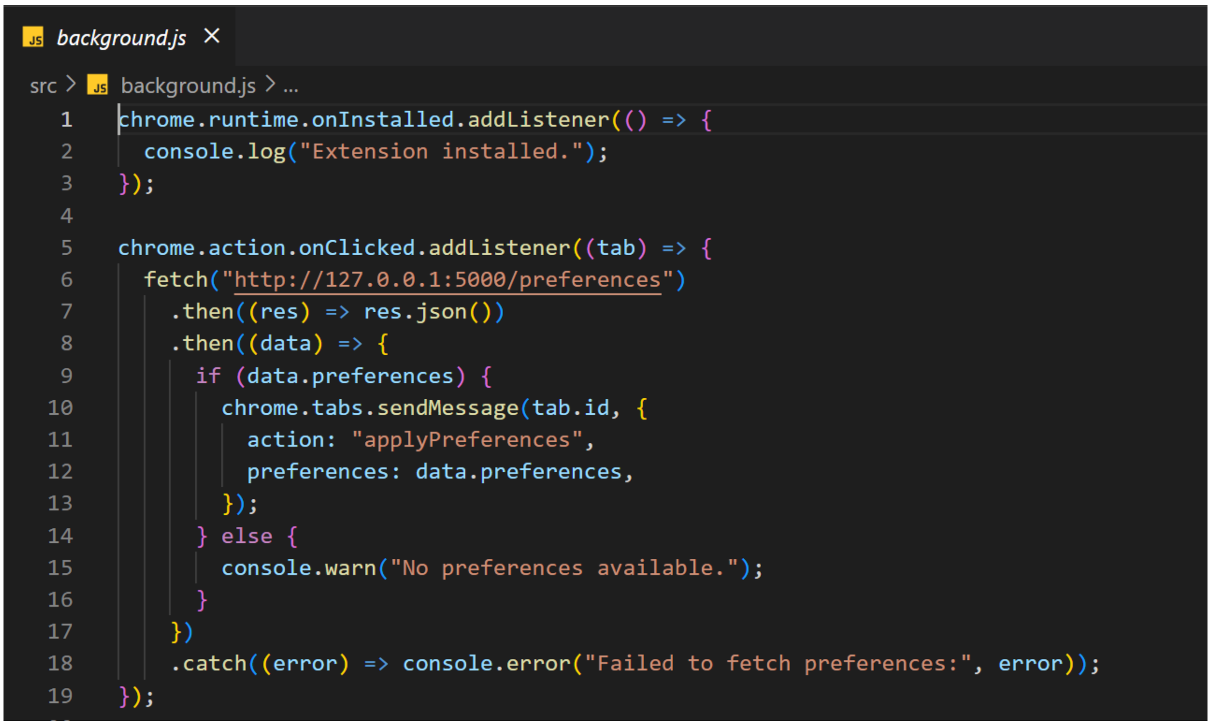
```
PS C:\Users\Shanoli Silva\Documents\RP\medium-summary-extension\extension>
```

Figure 16: Frontend Project Structure



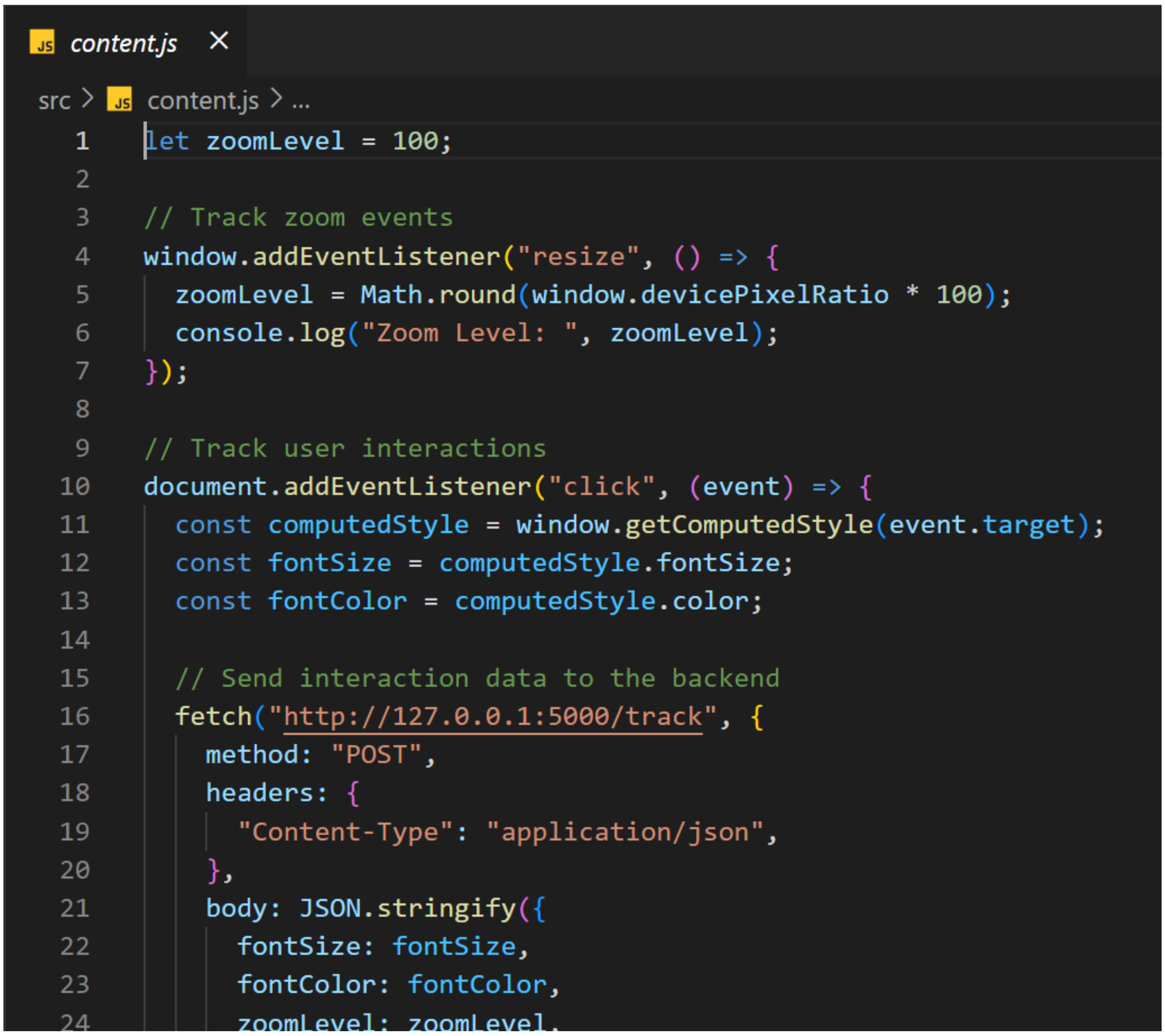
```javascript
import React, { useState } from "react";
import axios from "axios";
import { FaCopy, FaSync, FaVolumeUp, FaVolumeMute, FaThumbsUp, FaThumbsDown } from "react-icons/fa";
import { PuffLoader } from "react-spinners";
import "./SummaryGenerator.css";

const SummaryComponent = () => {
  const [summaryType, setSummaryType] = useState("points");
  const [summary, setSummary] = useState("");
  const [loading, setLoading] = useState(false);
  const [liked, setLiked] = useState(false);
  const [disliked, setDisliked] = useState(false);
  const [generated, setGenerated] = useState(false);
  const [isSpeaking, setIsSpeaking] = useState(false);
  const [utterance, setUtterance] = useState(null);

  const handleGenerateSummary = async () => {
    setLoading(true);
    setSummary("");
    setGenerated(false);
    const [tab] = await chrome.tabs.query({ active: true, currentWindow: true });
    const pageUrl = tab.url;

    try {
      const response = await axios.post("http://127.0.0.1:8000/summarize", {
        url: pageUrl,
        type: summaryType,
      });

      setSummary(response.data.summary);
      setGenerated(true);
      saveSummary();
    } catch (error) {
      console.error("Error generating summary:", error);
      setSummary("Failed to generate summary. Please try again.");
    } finally {
      setLoading(false);
    }
  };

  const getUserEmail = async () => {
    return new Promise((resolve, reject) => {
      chrome.identity.getProfileUserInfo({ accountStatus: 'ANY' }, (userInfo) => {
        if (userInfo.email) {
          resolve(userInfo.email);
        } else {
          reject("No email found");
        }
      });
    });
  };

  const saveSummary = async (liked = false, disliked = false) => {
    if (!summary) return;
```

Figure 17: Implementation of the Summary Component



```javascript
chrome.runtime.onInstalled.addListener(() => {
  console.log("Extension installed.");
});

chrome.action.onClicked.addListener((tab) => {
  fetch("http://127.0.0.1:5000/preferences")
    .then((res) => res.json())
    .then((data) => {
      if (data.preferences) {
        chrome.tabs.sendMessage(tab.id, {
          action: "applyPreferences",
          preferences: data.preferences,
        });
      } else {
        console.warn("No preferences available.");
      }
    })
    .catch((error) => console.error("Failed to fetch preferences:", error));
});
```

Figure 18: Implementation of background.js

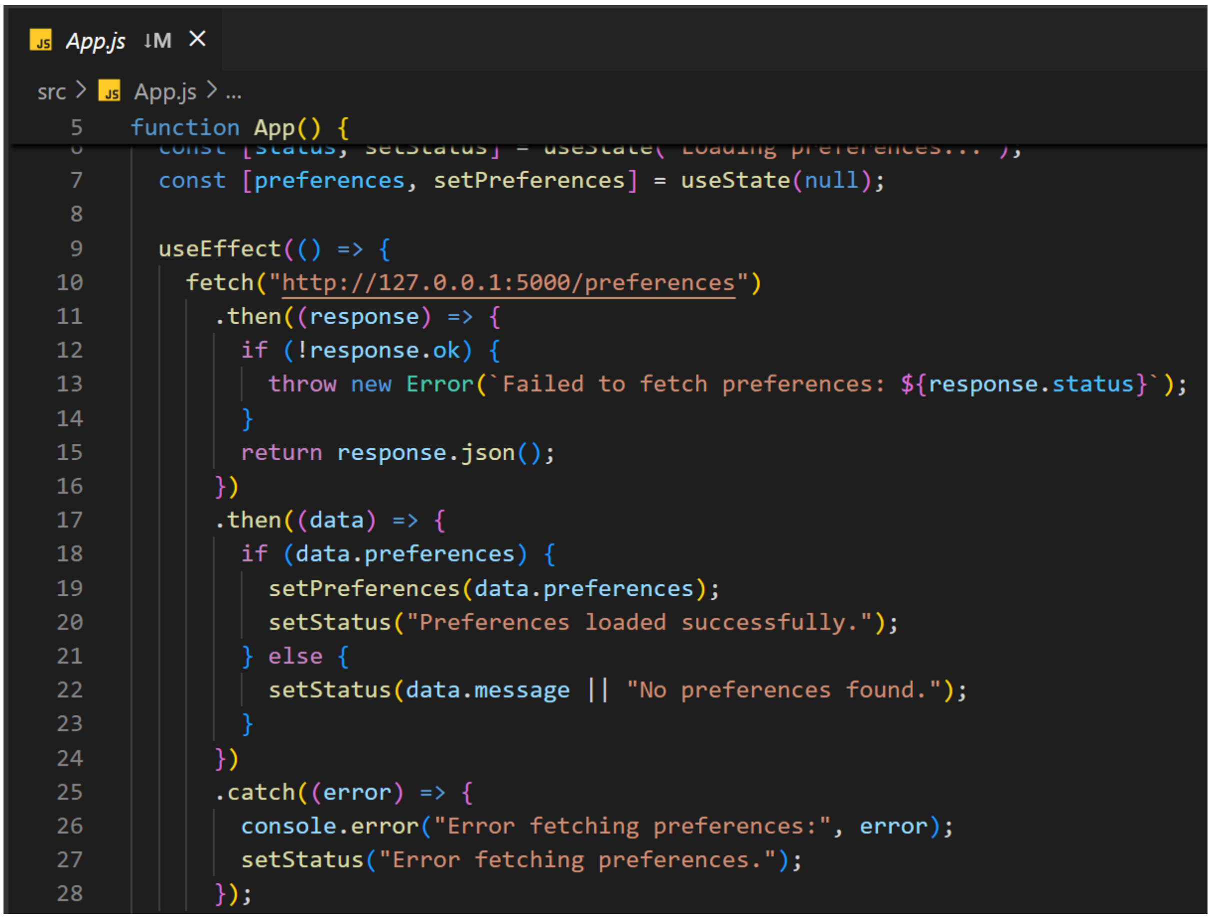


content.js ×

src > content.js > ...

```javascript
let zoomLevel = 100;

// Track zoom events
window.addEventListener("resize", () => {
  zoomLevel = Math.round(window.devicePixelRatio * 100);
  console.log("Zoom Level: ", zoomLevel);
});

// Track user interactions
document.addEventListener("click", (event) => {
  const computedStyle = window.getComputedStyle(event.target);
  const fontSize = computedStyle.fontSize;
  const fontColor = computedStyle.color;

  // Send interaction data to the backend
  fetch("http://127.0.0.1:5000/track", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      fontSize: fontSize,
      fontColor: fontColor,
      zoomLevel: zoomLevel,
```

Figure 19: Implementation of content.js



```javascript
function App() {
  const [status, setStatus] = useState("Loading preferences...");
  const [preferences, setPreferences] = useState(null);

  useEffect(() => {
    fetch("http://127.0.0.1:5000/preferences")
      .then((response) => {
        if (!response.ok) {
          throw new Error(`Failed to fetch preferences: ${response.status}`);
        }
        return response.json();
      })
      .then((data) => {
        if (data.preferences) {
          setPreferences(data.preferences);
          setStatus("Preferences loaded successfully.");
        } else {
          setStatus(data.message || "No preferences found.");
        }
      })
      .catch((error) => {
        console.error("Error fetching preferences:", error);
        setStatus("Error fetching preferences.");
      });
```

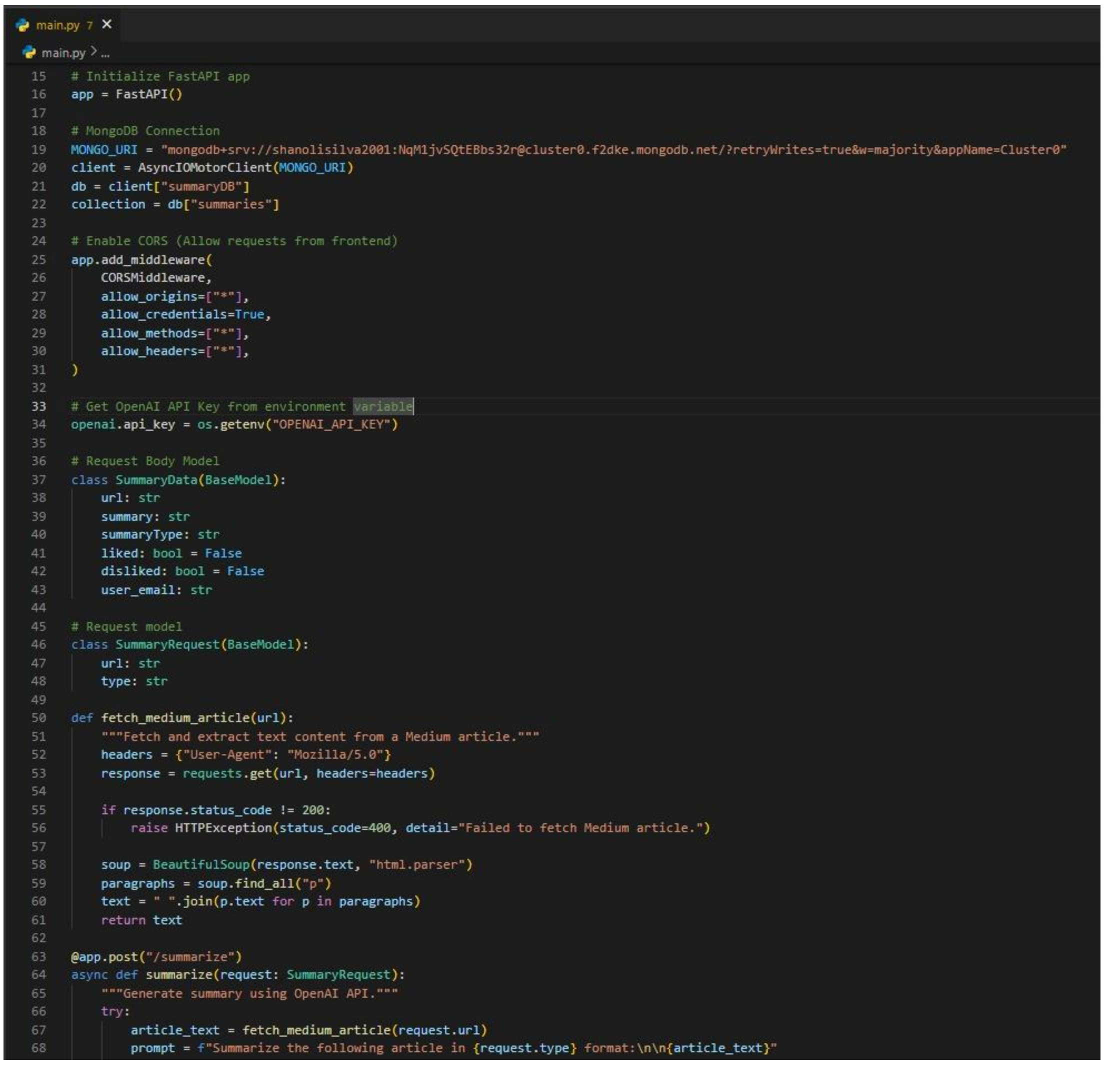
Figure 20: Implementation of app.js

2.1.4.1 Backend Implementation

Content-Based UI Personalization Implementation:

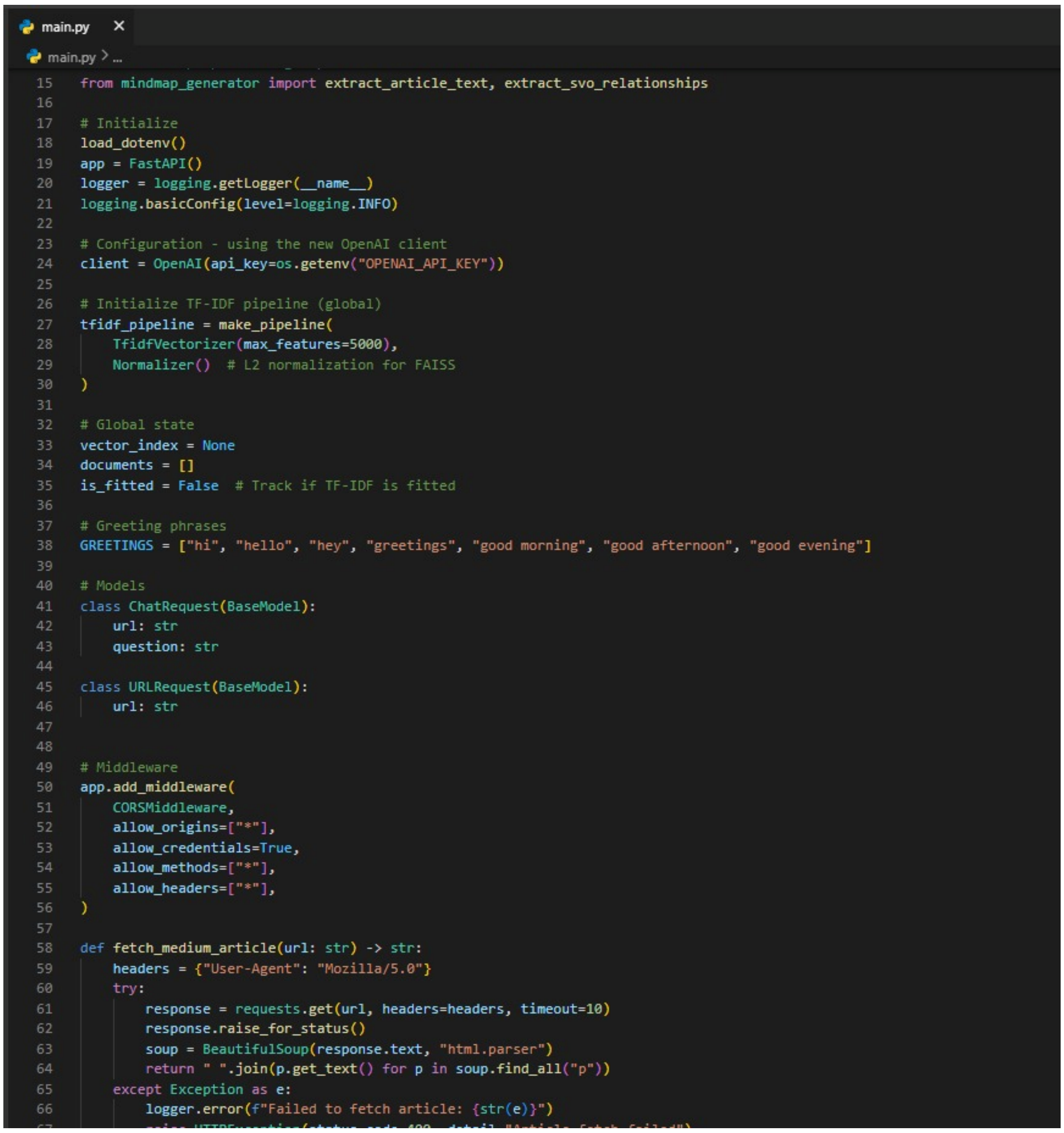
The backend implementation for the content-based UI personalization module was carried out using FastAPI, providing a microservice that handled three core functionalities: summarization, chatbot interaction, and mind map generation. For summarization, the service exposed a /summarize endpoint, where it scraped Medium article content using newspaper3k, applied format-specific prompt engineering, and queried the OpenAI GPT-3.5 API to generate bullet point, short, or long summaries. For the chatbot, a /ask_question endpoint processed user queries by embedding article chunks with TF-IDF and MiniLM, storing them into a FAISS vector database, retrieving the top-matching sections, and using OpenAI GPT-3.5 to formulate semantically rich, article-aware responses. The /generate_mindmap endpoint handled mind map generation by parsing articles through spaCy to extract subject-verb-object

(SVO) relationships and returning structured data for visualization in the frontend. All user interactions, feedback, and usage logs were stored in MongoDB, ensuring personalization learning and future system improvements.



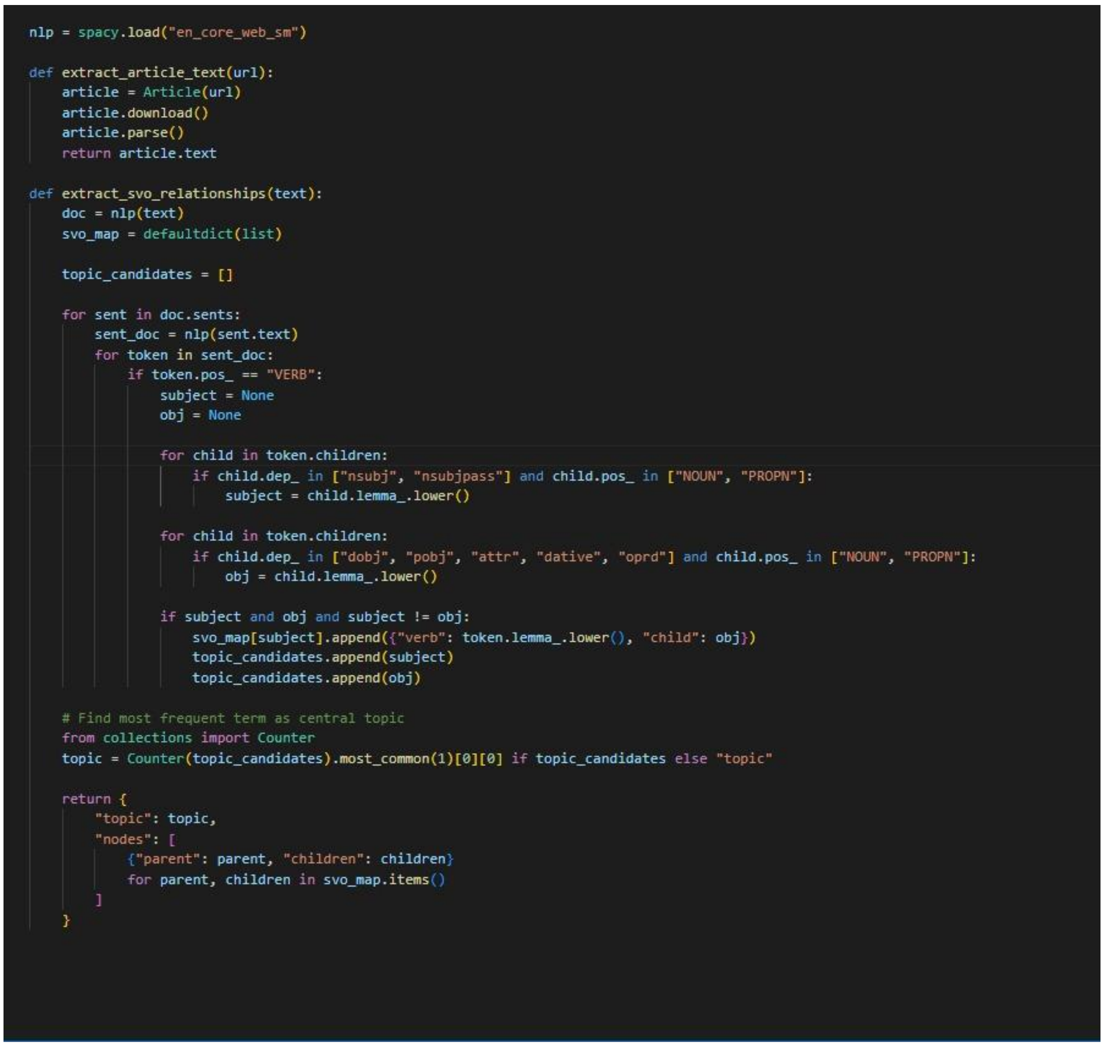
```python
# Initialize FastAPI app
app = FastAPI()

# MongoDB Connection
MONGO_URI = "mongodb+srv://shanolisilva2001:NqM1jvSQtEBbs32r@cluster0.f2dke.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
client = AsyncIOMotorClient(MONGO_URI)
db = client["summaryDB"]
collection = db["summaries"]

# Enable CORS (Allow requests from frontend)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Get OpenAI API Key from environment variable
openai.api_key = os.getenv("OPENAI_API_KEY")

# Request Body Model
class SummaryData(BaseModel):
    url: str
    summary: str
    summaryType: str
    liked: bool = False
    disliked: bool = False
    user_email: str

# Request model
class SummaryRequest(BaseModel):
    url: str
    type: str

def fetch_medium_article(url):
    """Fetch and extract text content from a Medium article."""
    headers = {"User-Agent": "Mozilla/5.0"}
    response = requests.get(url, headers=headers)

    if response.status_code != 200:
        raise HTTPException(status_code=400, detail="Failed to fetch Medium article.")

    soup = BeautifulSoup(response.text, "html.parser")
    paragraphs = soup.find_all("p")
    text = " ".join(p.text for p in paragraphs)
    return text

@app.post("/summarize")
async def summarize(request: SummaryRequest):
    """Generate summary using OpenAI API."""
    try:
        article_text = fetch_medium_article(request.url)
        prompt = f"Summarize the following article in {request.type} format:\n\n{article_text}"
```

Figure 21: Backend Implementation of Summary



```python
from mindmap_generator import extract_article_text, extract_svo_relationships

# Initialize
load_dotenv()
app = FastAPI()
logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)

# Configuration - using the new OpenAI client
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# Initialize TF-IDF pipeline (global)
tfidf_pipeline = make_pipeline(
    TfidfVectorizer(max_features=5000),
    Normalizer()  # L2 normalization for FAISS
)

# Global state
vector_index = None
documents = []
is_fitted = False  # Track if TF-IDF is fitted

# Greeting phrases
GREETINGS = ["hi", "hello", "hey", "greetings", "good morning", "good afternoon", "good evening"]

# Models
class ChatRequest(BaseModel):
    url: str
    question: str

class URLRequest(BaseModel):
    url: str


# Middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

def fetch_medium_article(url: str) -> str:
    headers = {"User-Agent": "Mozilla/5.0"}
    try:
        response = requests.get(url, headers=headers, timeout=10)
        response.raise_for_status()
        soup = BeautifulSoup(response.text, "html.parser")
        return " ".join(p.get_text() for p in soup.find_all("p"))
    except Exception as e:
        logger.error(f"Failed to fetch article: {str(e)}")
            raise HTTPException(status_code=400, detail="Article fetch failed")
```

Figure 22: Backend Implementation of Chatbot

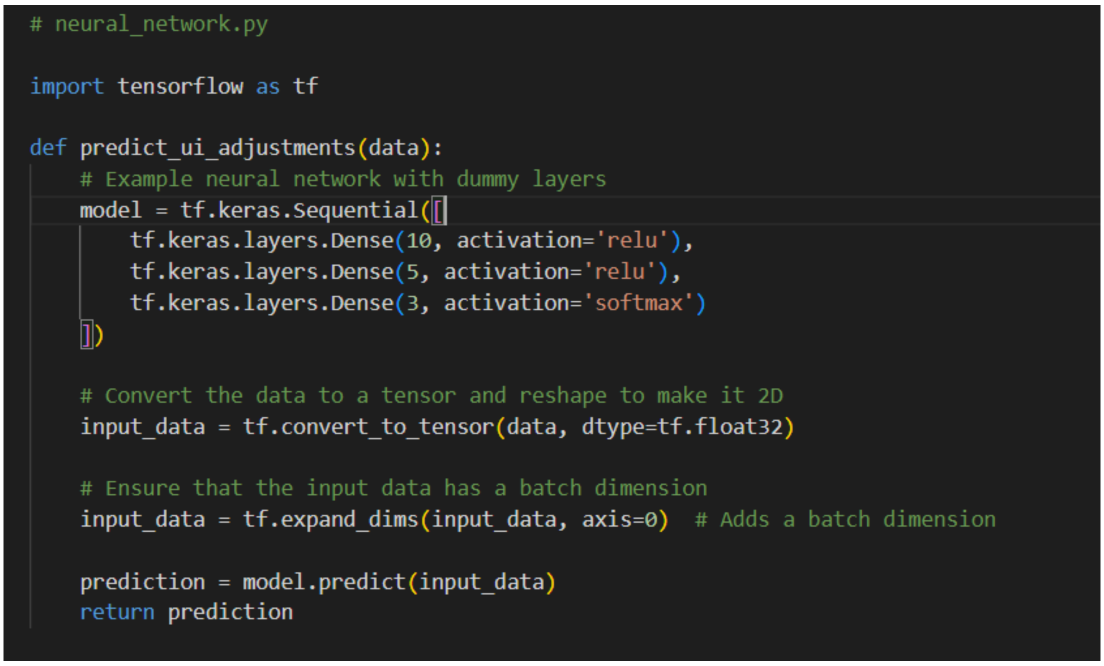


```python
nlp = spacy.load("en_core_web_sm")

def extract_article_text(url):
    article = Article(url)
    article.download()
    article.parse()
    return article.text

def extract_svo_relationships(text):
    doc = nlp(text)
    svo_map = defaultdict(list)

    topic_candidates = []

    for sent in doc.sents:
        sent_doc = nlp(sent.text)
        for token in sent_doc:
            if token.pos_ == "VERB":
                subject = None
                obj = None

                for child in token.children:
                    if child.dep_ in ["nsubj", "nsubjpass"] and child.pos_ in ["NOUN", "PROPN"]:
                        subject = child.lemma_.lower()

                for child in token.children:
                    if child.dep_ in ["dobj", "pobj", "attr", "dative", "oprd"] and child.pos_ in ["NOUN", "PROPN"]:
                        obj = child.lemma_.lower()

                if subject and obj and subject != obj:
                    svo_map[subject].append({"verb": token.lemma_.lower(), "child": obj})
                    topic_candidates.append(subject)
                    topic_candidates.append(obj)

    # Find most frequent term as central topic
    from collections import Counter
    topic = Counter(topic_candidates).most_common(1)[0][0] if topic_candidates else "topic"

    return {
        "topic": topic,
        "nodes": [
            {"parent": parent, "children": children}
            for parent, children in svo_map.items()
        ]
    }
```

Figure 23: Backend Implementation of Mind Map Generation

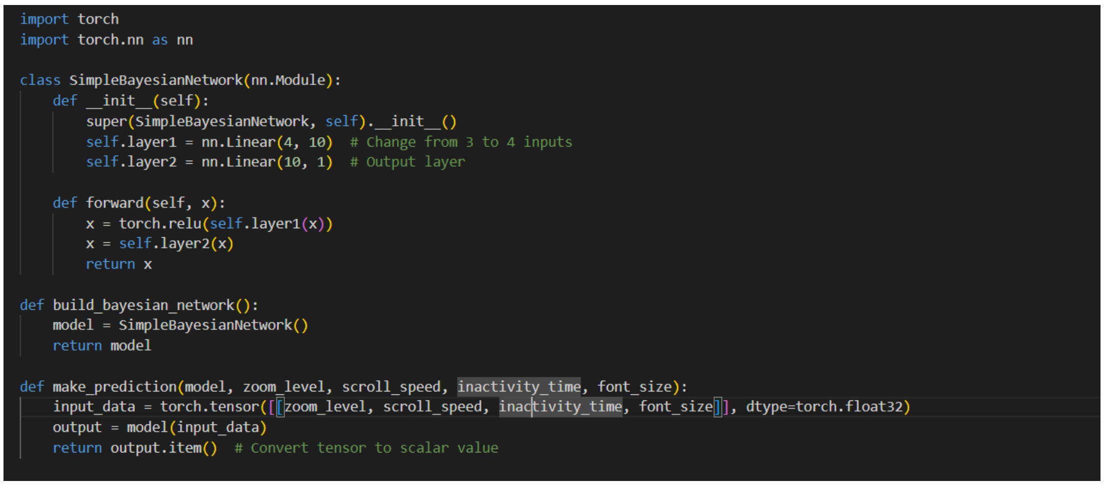
Behavior-Based UI Personalization Implementation:

The backend implementation for behavior-based UI personalization focused on building intelligent models that dynamically adjust user interface (UI) parameters like zoom level, font size, and scroll sensitivity based on real-time user behavior. Three major modules were developed independently and integrated via API services to communicate with the Chrome extension frontend. First, a Neural Network model was built using TensorFlow with a sequential architecture comprising input, hidden, and output layers to predict immediate UI adjustments from real-time session data such as scrolling velocity, zoom level, inactivity duration, and font size settings. Second, a Bayesian Network model was created using PyTorch to capture probabilistic relationships between user behaviors and UI adaptation preferences, offering a confidence score for recommending interface changes. Finally, a Q-Learning reinforcement agent was developed, setting up browsing states and UI adjustment actions, and using a dynamic Q-table updated through reward signals to learn the optimal personalization policy over time. These three models—predictive, probabilistic, and reinforcement-based—worked together to create a self-adaptive web experience, with lightweight deployment to ensure real-time responsiveness and flexibility in updating user interface elements based on evolving user interaction patterns.



```python
# neural_network.py

import tensorflow as tf

def predict_ui_adjustments(data):
    # Example neural network with dummy layers
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(10, activation='relu'),
        tf.keras.layers.Dense(5, activation='relu'),
        tf.keras.layers.Dense(3, activation='softmax')
    ])

    # Convert the data to a tensor and reshape to make it 2D
    input_data = tf.convert_to_tensor(data, dtype=tf.float32)

    # Ensure that the input data has a batch dimension
    input_data = tf.expand_dims(input_data, axis=0)  # Adds a batch dimension

    prediction = model.predict(input_data)
    return prediction
```

Figure 24: Neural Network Model

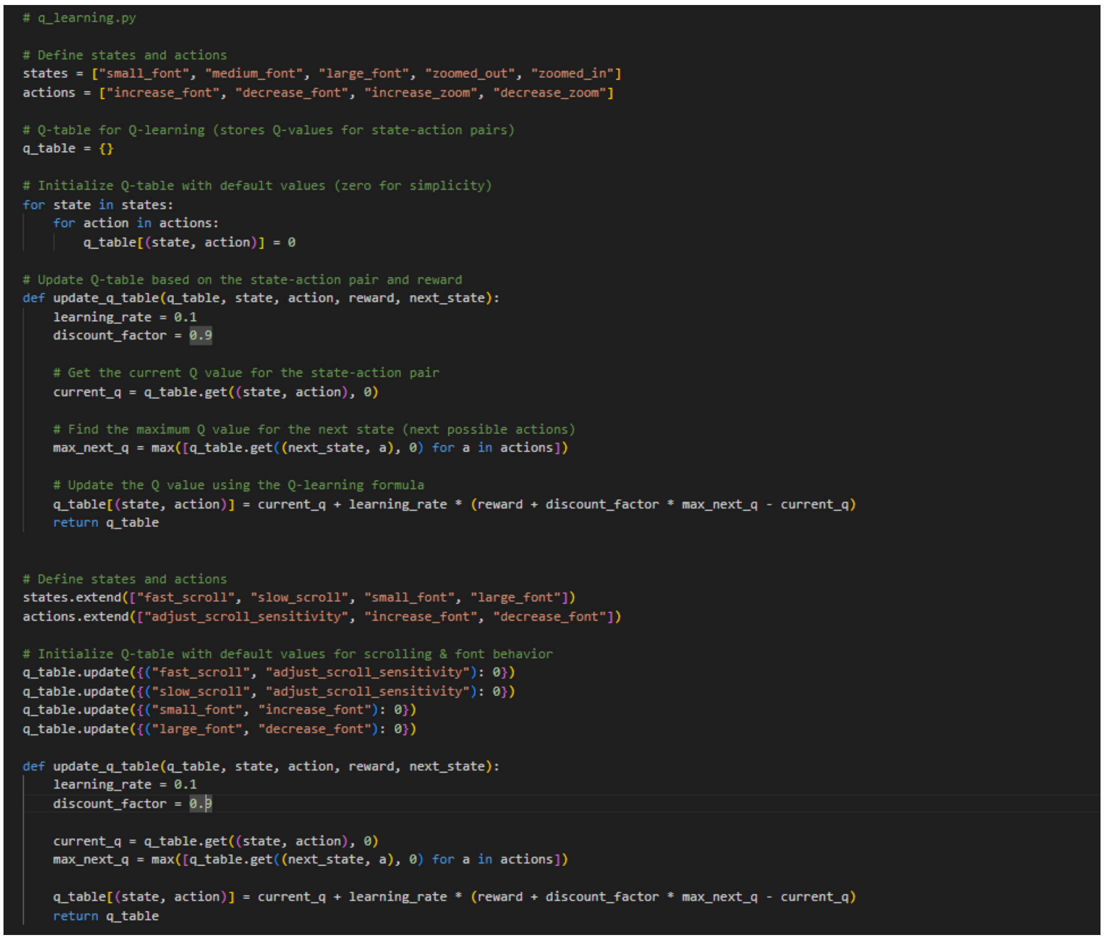


```python
import torch
import torch.nn as nn

class SimpleBayesianNetwork(nn.Module):
    def __init__(self):
        super(SimpleBayesianNetwork, self).__init__()
        self.layer1 = nn.Linear(4, 10)  # Change from 3 to 4 inputs
        self.layer2 = nn.Linear(10, 1)  # Output layer

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = self.layer2(x)
        return x

def build_bayesian_network():
    model = SimpleBayesianNetwork()
    return model

def make_prediction(model, zoom_level, scroll_speed, inactivity_time, font_size):
    input_data = torch.tensor([[zoom_level, scroll_speed, inactivity_time, font_size]], dtype=torch.float32)
    output = model(input_data)
    return output.item()  # Convert tensor to scalar value
```

Figure 25: Bayesian Network Model

$$Q(s,a) \leftarrow Q(s,a) + \alpha \times (r + \gamma \times \max_{a'} Q(s',a') - Q(s,a))$$

Figure 26: Q-Learning Update Rule

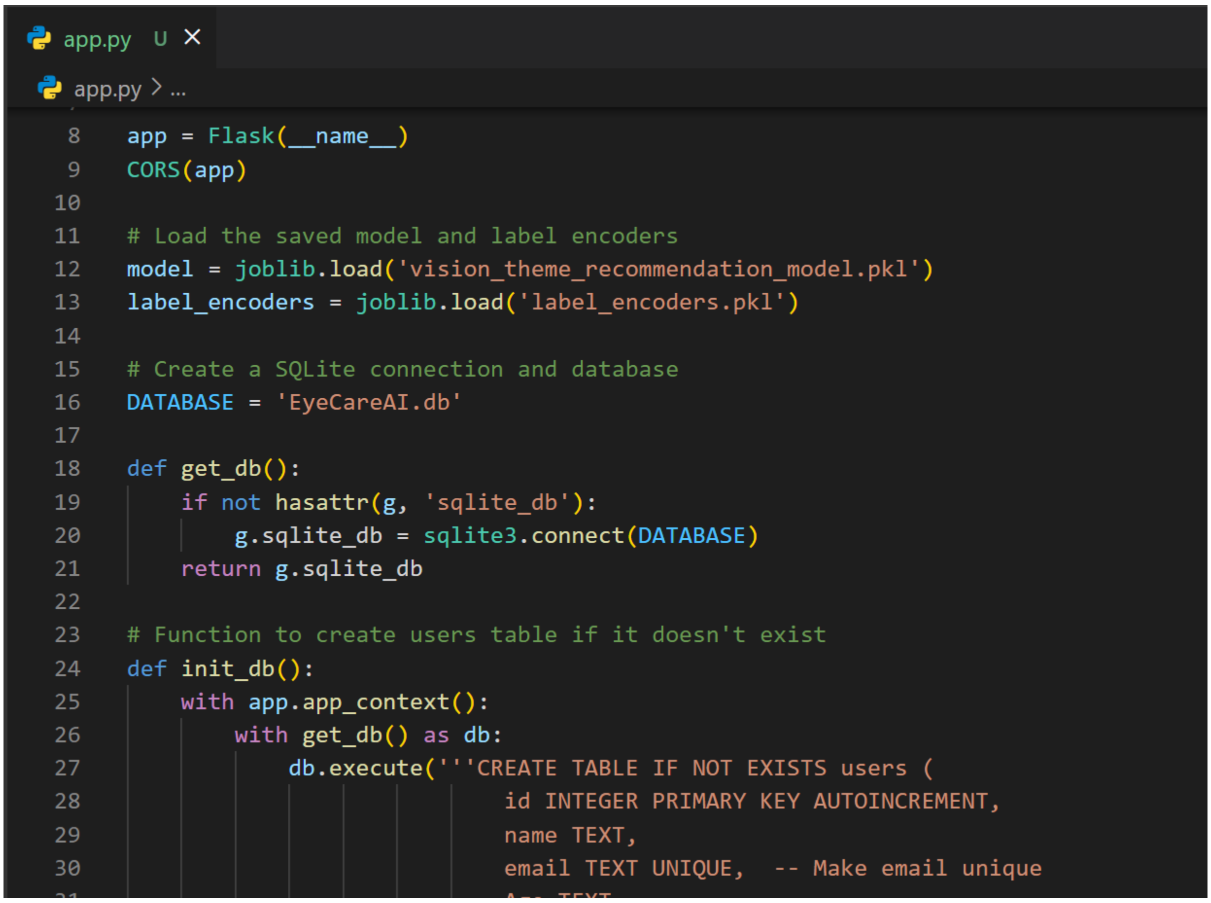


```python
# q_learning.py

# Define states and actions
states = ["small_font", "medium_font", "large_font", "zoomed_out", "zoomed_in"]
actions = ["increase_font", "decrease_font", "increase_zoom", "decrease_zoom"]

# Q-table for Q-learning (stores Q-values for state-action pairs)
q_table = {}

# Initialize Q-table with default values (zero for simplicity)
for state in states:
    for action in actions:
        q_table[(state, action)] = 0

# Update Q-table based on the state-action pair and reward
def update_q_table(q_table, state, action, reward, next_state):
    learning_rate = 0.1
    discount_factor = 0.9

    # Get the current Q value for the state-action pair
    current_q = q_table.get((state, action), 0)

    # Find the maximum Q value for the next state (next possible actions)
    max_next_q = max([q_table.get((next_state, a), 0) for a in actions])

    # Update the Q value using the Q-learning formula
    q_table[(state, action)] = current_q + learning_rate * (reward + discount_factor * max_next_q - current_q)
    return q_table


# Define states and actions
states.extend(["fast_scroll", "slow_scroll", "small_font", "large_font"])
actions.extend(["adjust_scroll_sensitivity", "increase_font", "decrease_font"])

# Initialize Q-table with default values for scrolling & font behavior
q_table.update({("fast_scroll", "adjust_scroll_sensitivity"): 0})
q_table.update({("slow_scroll", "adjust_scroll_sensitivity"): 0})
q_table.update({("small_font", "increase_font"): 0})
q_table.update({("large_font", "decrease_font"): 0})

def update_q_table(q_table, state, action, reward, next_state):
    learning_rate = 0.1
    discount_factor = 0.9

    current_q = q_table.get((state, action), 0)
    max_next_q = max([q_table.get((next_state, a), 0) for a in actions])

    q_table[(state, action)] = current_q + learning_rate * (reward + discount_factor * max_next_q - current_q)
    return q_table
```

Figure 27: Q-Learning Reinforcement Agent

Accessibility-Based UI Personalization for Visually Impaired Users Implementation:

The backend implementation for accessibility-focused UI personalization was built using Flask and focused on delivering adaptive screen themes for users with visual impairments. A dedicated /predict_theme endpoint accepted user-specific vision inputs such as age, gender, color blindness, short-sightedness, distance vision, near vision, and color discrimination issues. These inputs were preprocessed using Label Encoding and Standard Scaling before being fed into a pre-trained XGBoost Classifier. This model, optimized using techniques like SMOTE for handling class imbalance, predicted a personalized UI theme including font size, background color, text color, and zoom level, mapped via entries stored in an SQLite database. The Flask API returned these theme configurations to the frontend dynamically, enabling real-time updates without user intervention. All user data, theme predictions, and accessibility feedback were logged securely to facilitate continuous learning and experience consistency across browsing sessions.



```python
app = Flask(__name__)
CORS(app)

# Load the saved model and label encoders
model = joblib.load('vision_theme_recommendation_model.pkl')
label_encoders = joblib.load('label_encoders.pkl')

# Create a SQLite connection and database
DATABASE = 'EyeCareAI.db'

def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = sqlite3.connect(DATABASE)
    return g.sqlite_db

# Function to create users table if it doesn't exist
def init_db():
    with app.app_context():
        with get_db() as db:
            db.execute('''CREATE TABLE IF NOT EXISTS users (
                        id INTEGER PRIMARY KEY AUTOINCREMENT,
                        name TEXT,
                        email TEXT UNIQUE,  -- Make email unique
```

Figure 28: Model Implementation

2.1.5 Testing

The Testing phase was critical in validating the functionality, performance, and user experience of the overall Chrome extension, which integrates content-based UI personalization (summarization, chatbot, mind map), behavior-based UI personalization, and accessibility-focused UI personalization. A multi-layered testing strategy was followed, encompassing unit testing, integration testing, system testing, and acceptance testing to ensure robustness across each feature individually and collectively.

Unit Testing:

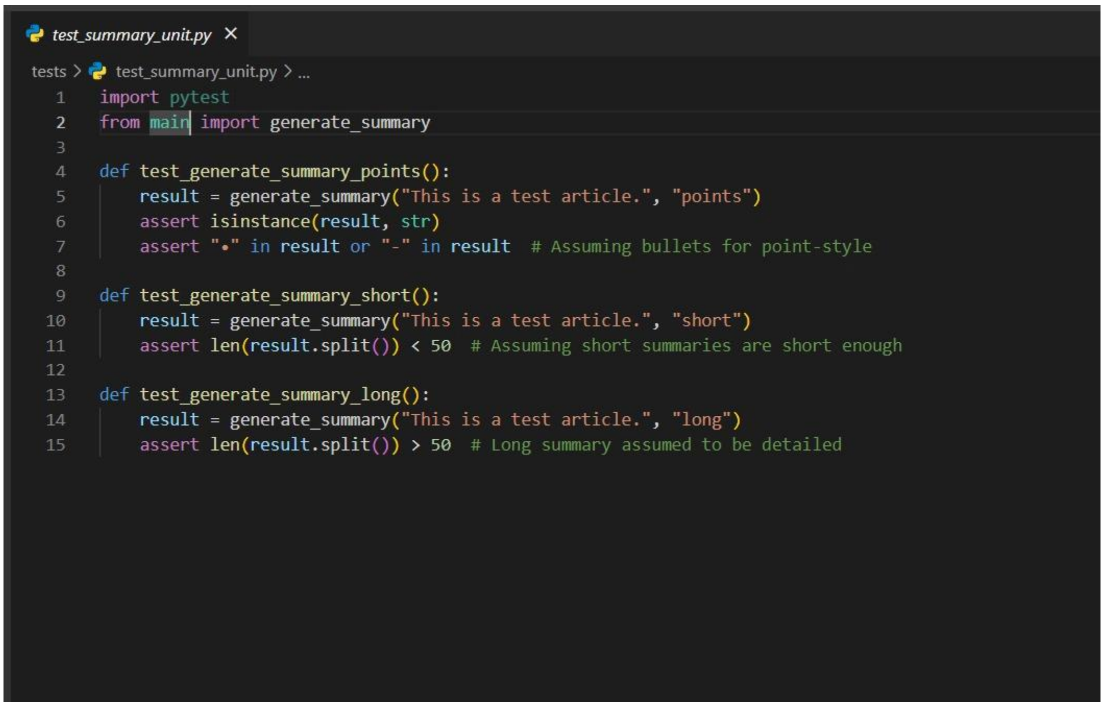
Unit testing was conducted to validate the behavior of individual backend functions and frontend components independently. For the backend microservices, the Python unittest library was used to test APIs responsible for summary generation, chatbot responses, mind map generation, behavioral prediction, and accessibility theme recommendation. Test cases were created to verify that:

- The summary API returned results within 5 seconds for valid Medium articles.
- The chatbot API provided semantically accurate answers based on embedded document chunks.
- The mind map API generated a valid node-link structure based on extracted SVO relationships.
- The behavior analysis models (neural network, Bayesian, Q-learning agent) correctly predicted necessary UI adaptations from mock user interaction data.
- The accessibility Flask API predicted accurate UI themes when vision condition inputs were provided.

On the frontend, Jest and React Testing Library was used to validate React components. Tests ensured that:

- Button click events triggered the correct API calls.
- Summaries, chatbot conversations, and mind maps rendered appropriately based on user inputs.
- User behavior tracking (scroll detection, zoom, font change) correctly logged real-time changes without causing crashes.
- Accessibility adjustments like font resizing and background color changes dynamically reflected the backend-predicted themes.

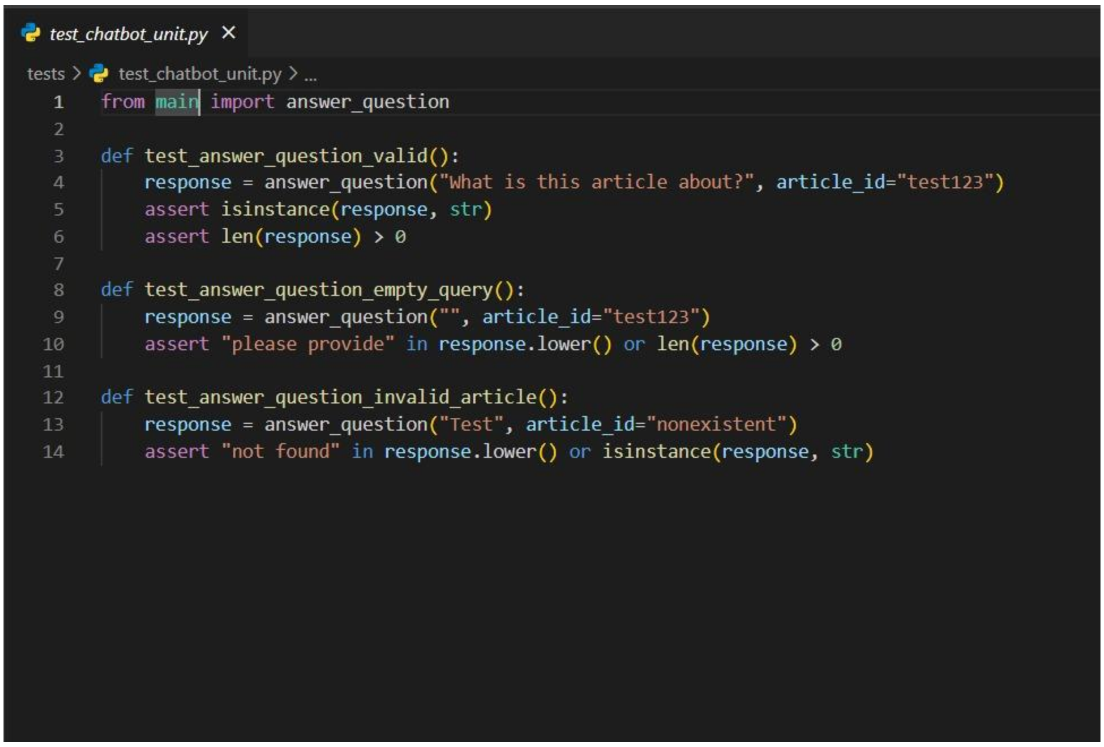
Sample unit test cases are shown in Figures 27–32, covering the summary generator, chatbot functionality, mind map node rendering, behavioral prediction handlers, and accessibility theme switchers.


test_summary_unit.py ×

tests > test_summary_unit.py > ...

```python
import pytest
from main import generate_summary

def test_generate_summary_points():
    result = generate_summary("This is a test article.", "points")
    assert isinstance(result, str)
    assert "•" in result or "-" in result  # Assuming bullets for point-style

def test_generate_summary_short():
    result = generate_summary("This is a test article.", "short")
    assert len(result.split()) < 50  # Assuming short summaries are short enough

def test_generate_summary_long():
    result = generate_summary("This is a test article.", "long")
    assert len(result.split()) > 50  # Long summary assumed to be detailed
```

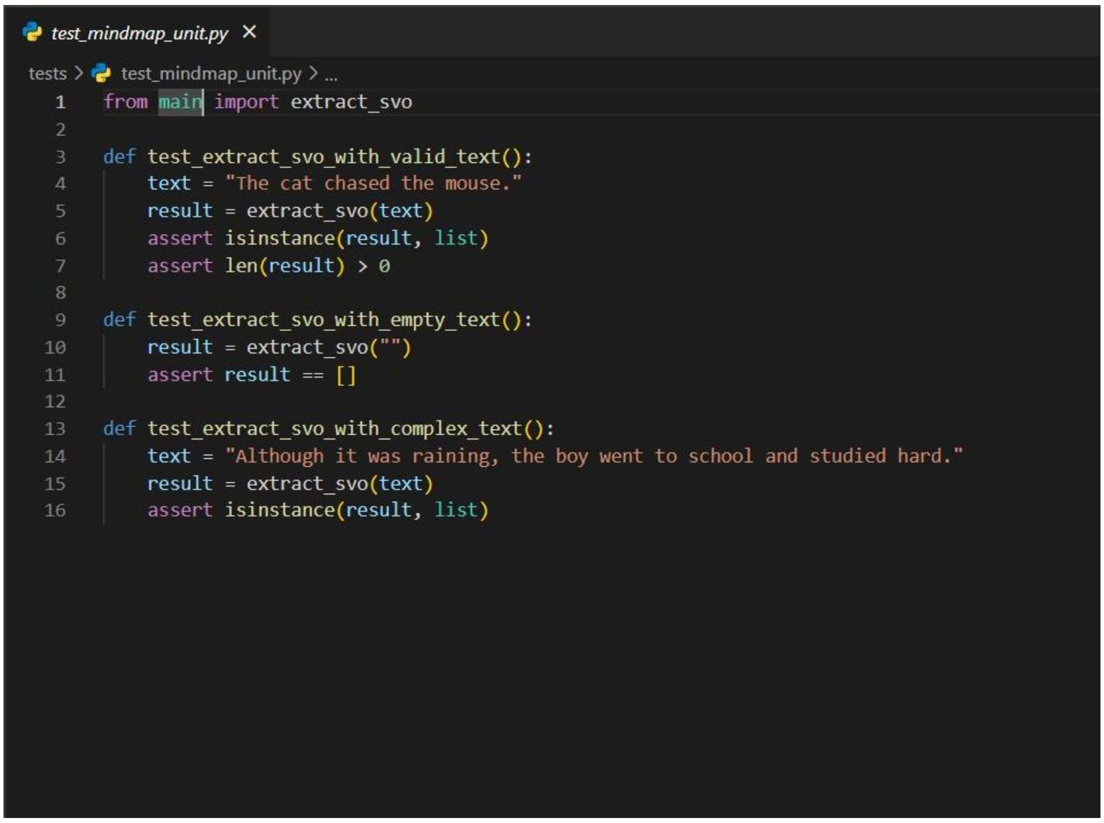
Figure 29: Unit Testing of Summary Generation


test_chatbot_unit.py ×

tests > test_chatbot_unit.py > ...

```python
from main import answer_question

def test_answer_question_valid():
    response = answer_question("What is this article about?", article_id="test123")
    assert isinstance(response, str)
    assert len(response) > 0

def test_answer_question_empty_query():
    response = answer_question("", article_id="test123")
    assert "please provide" in response.lower() or len(response) > 0

def test_answer_question_invalid_article():
    response = answer_question("Test", article_id="nonexistent")
    assert "not found" in response.lower() or isinstance(response, str)
```

Figure 30: Unit Testing of Chatbot


test_mindmap_unit.py ×

tests > test_mindmap_unit.py > ...

```python
from main import extract_svo

def test_extract_svo_with_valid_text():
    text = "The cat chased the mouse."
    result = extract_svo(text)
    assert isinstance(result, list)
    assert len(result) > 0

def test_extract_svo_with_empty_text():
    result = extract_svo("")
    assert result == []

def test_extract_svo_with_complex_text():
    text = "Although it was raining, the boy went to school and studied hard."
    result = extract_svo(text)
    assert isinstance(result, list)
```

Figure 31: Unit Testing of Mind Map Generation

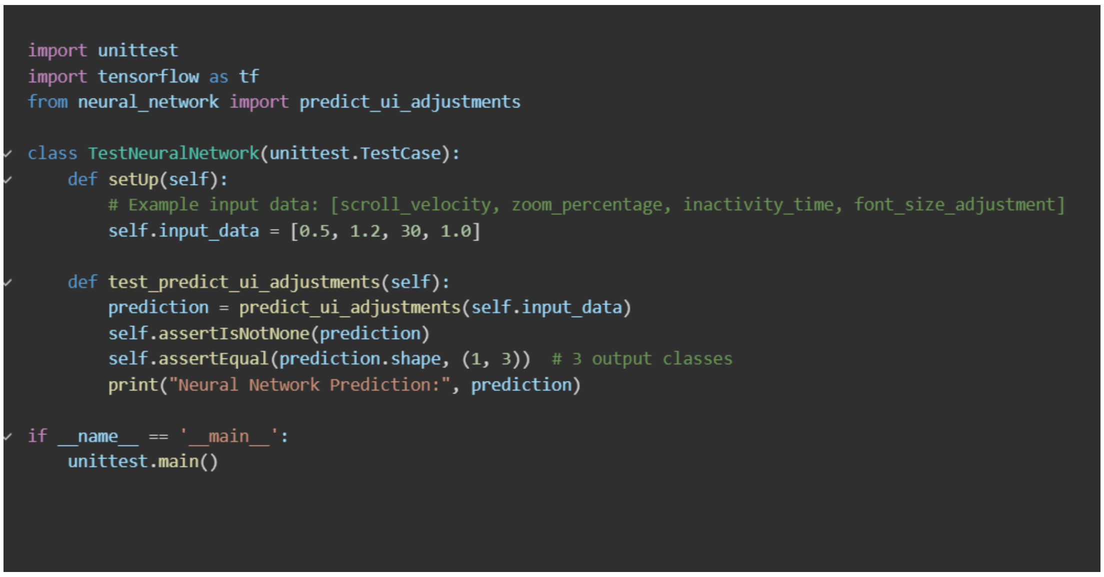

```python
import unittest
import tensorflow as tf
from neural_network import predict_ui_adjustments

class TestNeuralNetwork(unittest.TestCase):
    def setUp(self):
        # Example input data: [scroll_velocity, zoom_percentage, inactivity_time, font_size_adjustment]
        self.input_data = [0.5, 1.2, 30, 1.0]

    def test_predict_ui_adjustments(self):
        prediction = predict_ui_adjustments(self.input_data)
        self.assertIsNotNone(prediction)
        self.assertEqual(prediction.shape, (1, 3))  # 3 output classes
        print("Neural Network Prediction:", prediction)

if __name__ == '__main__':
    unittest.main()
```

Figure 32: Neural Network Model Testing

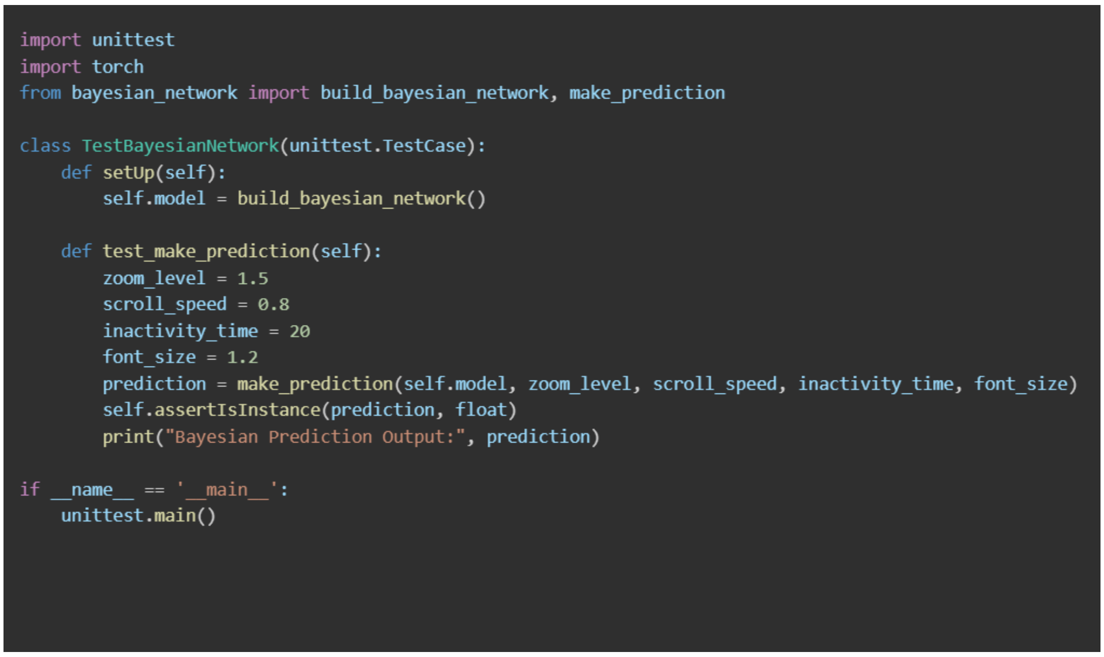
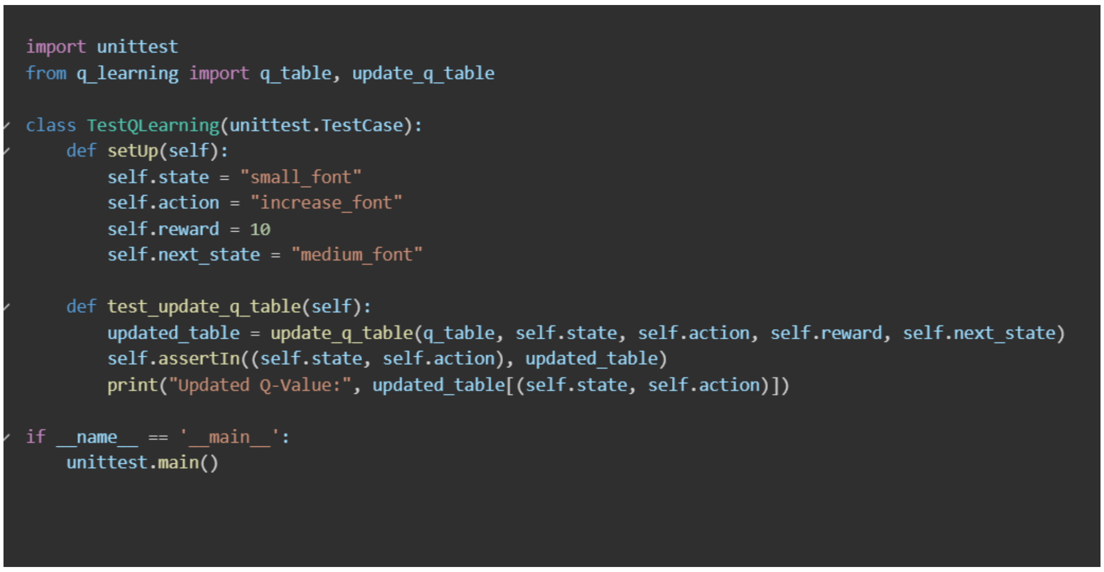

```python
import unittest
import torch
from bayesian_network import build_bayesian_network, make_prediction

class TestBayesianNetwork(unittest.TestCase):
    def setUp(self):
        self.model = build_bayesian_network()

    def test_make_prediction(self):
        zoom_level = 1.5
        scroll_speed = 0.8
        inactivity_time = 20
        font_size = 1.2
        prediction = make_prediction(self.model, zoom_level, scroll_speed, inactivity_time, font_size)
        self.assertIsInstance(prediction, float)
        print("Bayesian Prediction Output:", prediction)

if __name__ == '__main__':
    unittest.main()
```

Figure 33: Bayesian Network Model Testing



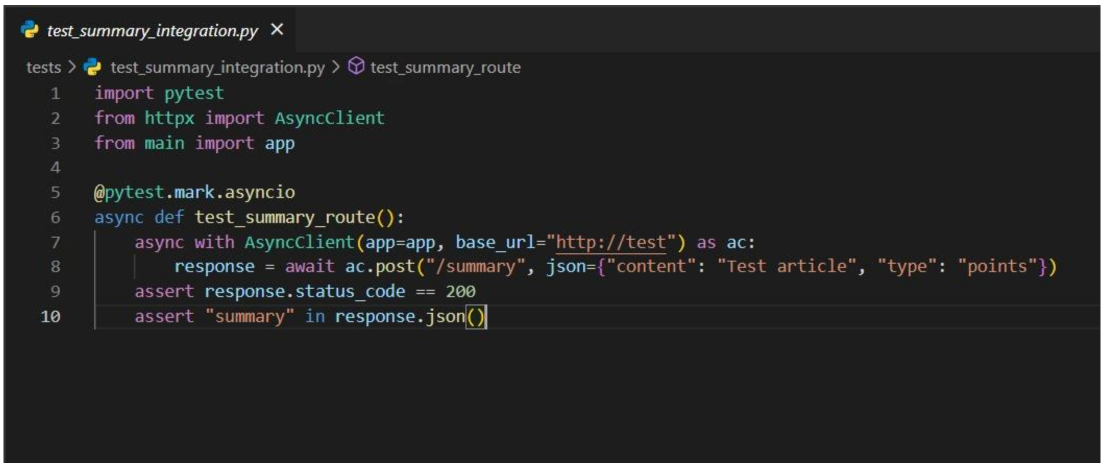
```python
import unittest
from q_learning import q_table, update_q_table

class TestQLearning(unittest.TestCase):
    def setUp(self):
        self.state = "small_font"
        self.action = "increase_font"
        self.reward = 10
        self.next_state = "medium_font"

    def test_update_q_table(self):
        updated_table = update_q_table(q_table, self.state, self.action, self.reward, self.next_state)
        self.assertIn((self.state, self.action), updated_table)
        print("Updated Q-Value:", updated_table[(self.state, self.action)])

if __name__ == '__main__':
    unittest.main()
```

Figure 34: Q-Learning Agent Testing

Integration Testing:

Integration testing evaluated the interaction between frontend and backend services to ensure seamless communication. Testing scenarios included:

- Verifying that the "Generate Summary" button triggered content scraping, summary generation via OpenAI, and correctly rendered the summary.
- Ensuring that chatbot queries from the UI passed through FAISS similarity search, retrieved relevant document chunks, and returned coherent OpenAI responses.
- Testing the real-time generation and rendering of mind maps using react-flow-renderer after article parsing.
- Confirming that real-time user behavioral signals from frontend (scroll, zoom) triggered appropriate UI changes as predicted by the backend behavior models.
- Testing accessibility workflows where vision-specific inputs resulted in theme adjustments dynamically fetched from the accessibility Flask API. Test data included live Medium articles and mock user profiles, while MongoDB logs were examined to ensure all events (summaries, chatbot logs, mind maps, behavior metrics, and accessibility changes) were properly recorded.

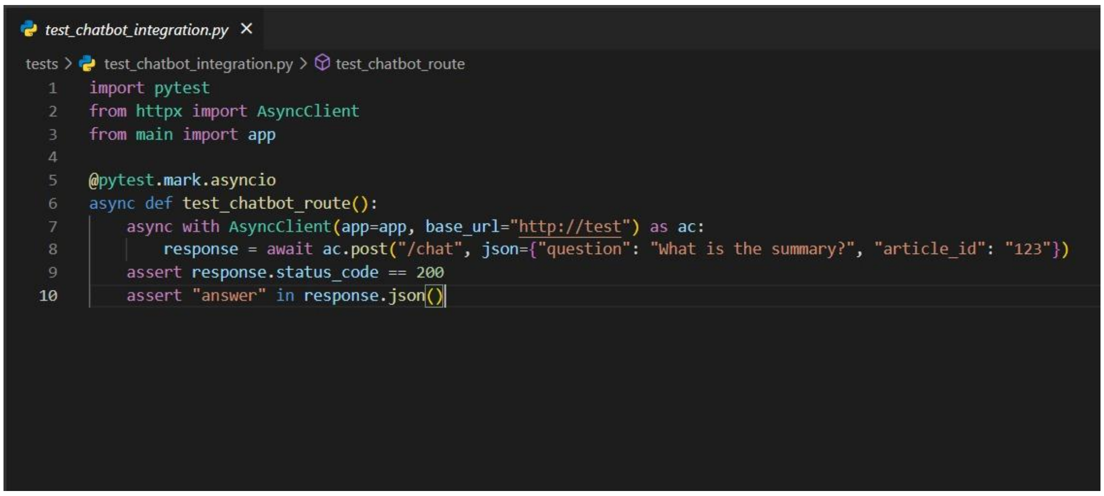


test_summary_integration.py ✕

tests > test_summary_integration.py > test_summary_route

```
import pytest
from httpx import AsyncClient
from main import app

@pytest.mark.asyncio
async def test_summary_route():
    async with AsyncClient(app=app, base_url="http://test") as ac:
        response = await ac.post("/summary", json={"content": "Test article", "type": "points"})
    assert response.status_code == 200
    assert "summary" in response.json()
```

Figure 35: Integration Testing Summary Generation

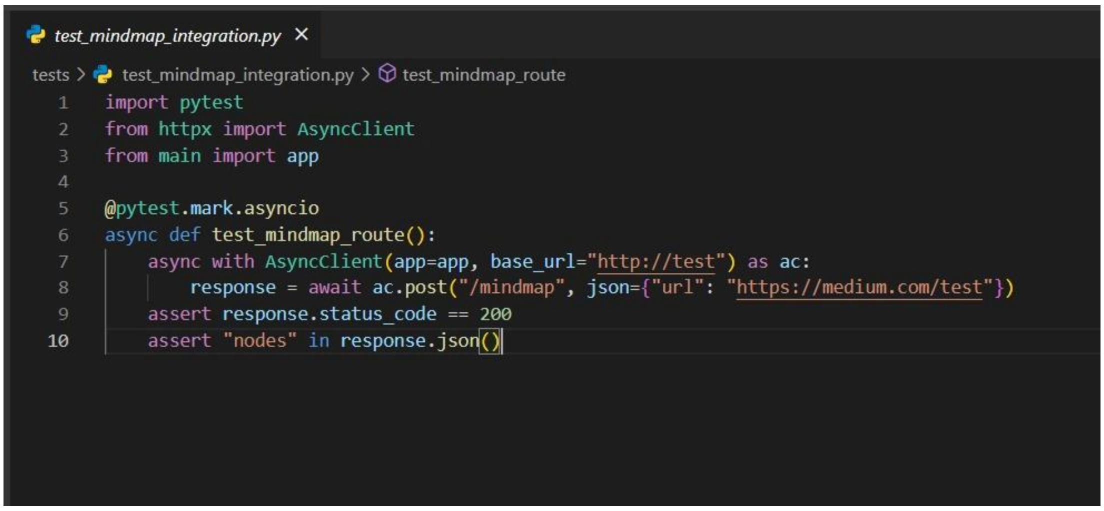


test_chatbot_integration.py ✕

tests > test_chatbot_integration.py > test_chatbot_route

```
import pytest
from httpx import AsyncClient
from main import app

@pytest.mark.asyncio
async def test_chatbot_route():
    async with AsyncClient(app=app, base_url="http://test") as ac:
        response = await ac.post("/chat", json={"question": "What is the summary?", "article_id": "123"})
    assert response.status_code == 200
    assert "answer" in response.json()
```

Figure 36: Integration Testing of Chatbot

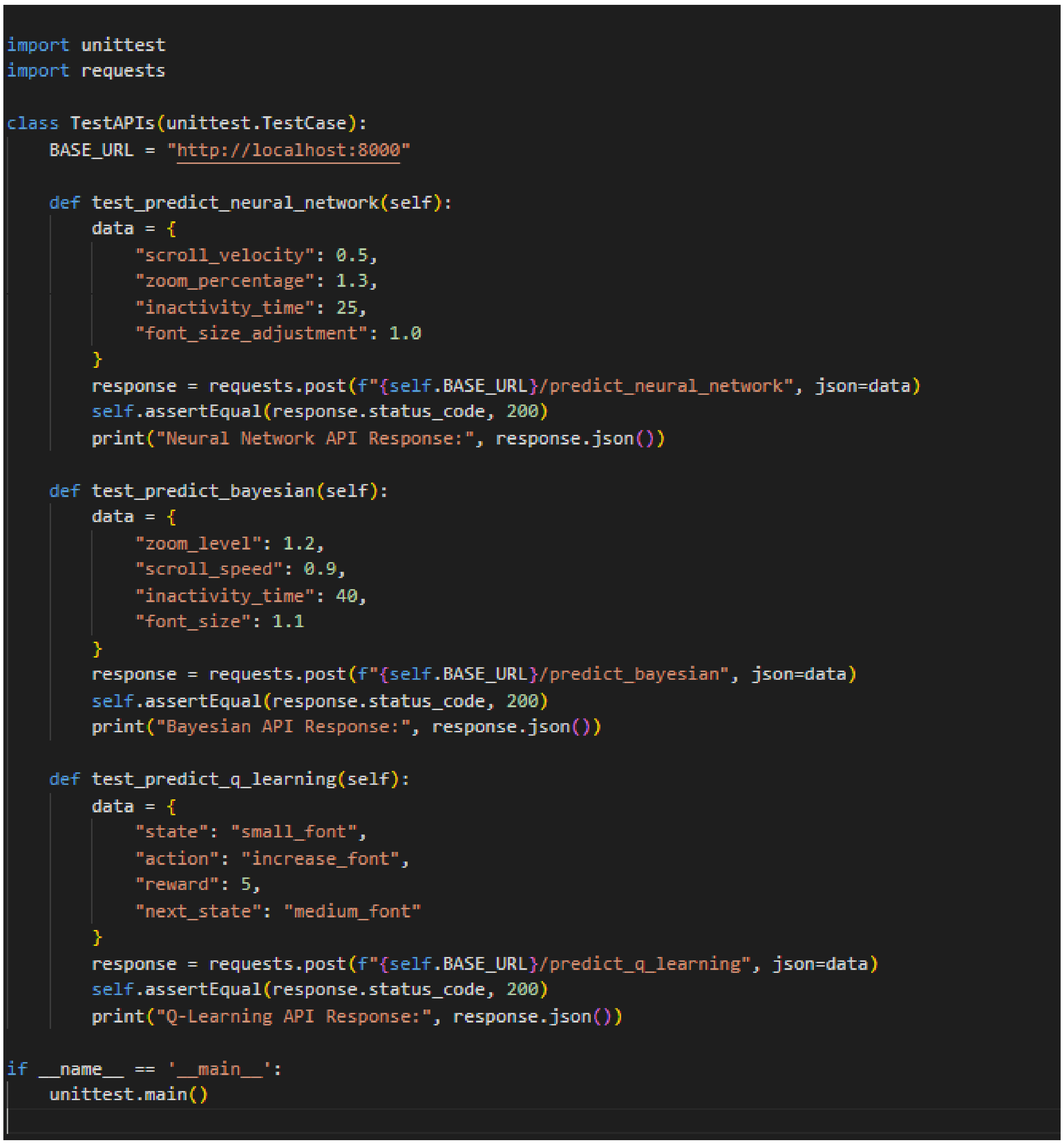


test_mindmap_integration.py ✕

tests > test_mindmap_integration.py > test_mindmap_route

```
import pytest
from httpx import AsyncClient
from main import app

@pytest.mark.asyncio
async def test_mindmap_route():
    async with AsyncClient(app=app, base_url="http://test") as ac:
        response = await ac.post("/mindmap", json={"url": "https://medium.com/test"})
    assert response.status_code == 200
    assert "nodes" in response.json()
```

Figure 37: Integration Testing of Mind Map Generation



```python
import unittest
import requests

class TestAPIs(unittest.TestCase):
    BASE_URL = "http://localhost:8000"

    def test_predict_neural_network(self):
        data = {
            "scroll_velocity": 0.5,
            "zoom_percentage": 1.3,
            "inactivity_time": 25,
            "font_size_adjustment": 1.0
        }
        response = requests.post(f"{self.BASE_URL}/predict_neural_network", json=data)
        self.assertEqual(response.status_code, 200)
        print("Neural Network API Response:", response.json())

    def test_predict_bayesian(self):
        data = {
            "zoom_level": 1.2,
            "scroll_speed": 0.9,
            "inactivity_time": 40,
            "font_size": 1.1
        }
        response = requests.post(f"{self.BASE_URL}/predict_bayesian", json=data)
        self.assertEqual(response.status_code, 200)
        print("Bayesian API Response:", response.json())

    def test_predict_q_learning(self):
        data = {
            "state": "small_font",
            "action": "increase_font",
            "reward": 5,
            "next_state": "medium_font"
        }
        response = requests.post(f"{self.BASE_URL}/predict_q_learning", json=data)
        self.assertEqual(response.status_code, 200)
        print("Q-Learning API Response:", response.json())

if __name__ == '__main__':
    unittest.main()
```

Figure 38: Integration Testing of Behavior-Based UI Personalization

System Testing:

System testing was performed to assess the full Chrome extension operating as a unified system across real-world conditions. This included:

- Testing across multiple Chromium-based browsers (Google Chrome, Brave, Edge) to confirm cross-browser compatibility.
- Smooth switching between the Summary, Chatbot, and Mind Map tabs without losing context or introducing performance lags.

- Behavior-based UI personalization responding adaptively to real user interactions (scrolling, zooming, engagement patterns) without page reloads.
- Accessibility features like Text-to-Speech (TTS), font adjustment, and contrast modifications were verified under different screen sizes and zoom levels.
- Error handling was tested for empty articles, invalid URLs, network disruptions, and OpenAI API rate limit scenarios. Edge case scenarios involved large articles exceeding 15,000 words, ensuring system stability and responsiveness.

Acceptance Testing:

Acceptance testing was conducted with a sample group of 30+ users including students, professionals, and visually impaired users. Participants installed the extension on their own browsers and used it for reading Medium articles. They were instructed to:

- Generate summaries in multiple formats.
- Ask questions through the chatbot interface.
- Visualize article structures through the mind map feature.
- Interact normally to trigger behavior-based UI adjustments.
- Submit their vision condition and observe personalized theme changes. Feedback was gathered via online surveys and structured interviews, covering metrics such as system speed, usability, clarity, personalization effectiveness, and overall satisfaction. Alpha testing (within controlled environments) validated core stability, while beta testing (deployed in user environments) helped identify minor usability improvements, particularly around mind map node zooming and chatbot fallback responses. The overwhelmingly positive feedback confirmed the system's alignment with user needs and expectations.

Manual Testing:

Manual testing was conducted throughout the development process to validate the real-world behavior of the full Chrome extension, covering all three major components: content-based UI personalization (summarization, chatbot, mind map generation), behavior-based UI personalization, and accessibility-based UI personalization for visually impaired users. This hands-on approach ensured that the system performed reliably under practical usage scenarios and complemented automated test coverage.

The following types of manual testing were performed:

- **Functional Testing:**

  Each feature was manually tested to confirm its correct functionality:

  - Summary generation was checked for all three formats (bullet points, short paragraph, long summary).
  - Chatbot responses were evaluated for relevance, coherence, and contextual accuracy when answering user queries about the article.
  - Mind map visualization was inspected to ensure proper node generation, hierarchy structure, and interactive features like zoom and node dragging.
  - Behavior-based UI adaptations such as font size adjustments and layout changes were verified after different scrolling speeds, zoom patterns, and user engagement durations.
  - Accessibility personalization was validated by simulating different user profiles (e.g., users with color blindness, short-sightedness) and checking the theme adjustments.

- **User Acceptance Testing (UAT):**

  End-users, including students, working professionals, and visually impaired individuals, were asked to install and use the Chrome extension. They interacted with all features naturally, and their feedback was collected regarding:

  - Ease of use and intuitiveness of the interface.
  - Usefulness of summaries, chatbot assistance, mind maps, and personalized themes.

  - Responsiveness and stability during continuous usage.

- **Exploratory Testing:**

  Beyond formal test cases, exploratory testing was performed to uncover unexpected behavior and edge cases, such as:

  - Switching rapidly between Summary, Chat, and Mind Map tabs.
  - Testing with extremely short articles or empty articles.
  - Interrupting API calls midway (e.g., disabling the internet) and observing graceful fallback handling.
  - Rapid zooming in/out or scrolling behavior to test the real-time behavioral adaptation engine.

- **Compatibility Testing:**

  To ensure the extension functioned consistently across environments, compatibility testing was performed on various Chromium-based browsers, including:

  - Google Chrome (latest stable version)
  - Brave Browser

Microsoft Edge Cross-browser testing validated UI rendering, API interactions, mind map visualization, behavioral personalization, and accessibility theme changes remained consistent without breaking functionality.

Table 5: Test Case for Summary Generation

Test Case ID	TC_SUM_01
Test Case Objective	Validate summary generation in all three formats
Pre-Requirements	Medium article loaded, Chrome extension installed
Test Steps	1. Open Chrome extension 2. Navigate to Summary tab

	3. Select each summary format (Points, Short, Long) 4. Click "Generate Summary"
Test Data	A live Medium article URL
Expected Output	Correct summaries generated for each format within 3-5 seconds
Actual Output	Summaries generated accurately and displayed
Status	Pass

Table 6: Test Case for Chatbot Integration

Test Case ID	TC_CHAT_01
Test Case Objective	Validate chatbot's context-aware response
Pre-Requirements	Medium article loaded, Chrome extension installed
Test Steps	1. Open Chrome extension. 2. Go to Chat tab. 3. Ask a question related to the article content. 4. Observe response time and relevance.
Test Data	User-entered questions based on the article
Expected Output	Chatbot answers the question accurately within 5 seconds.
Actual Output	Contextual and correct answers received
Status	Pass

Table 7: Mind Map Generation

Test Case ID	TC_MIND_01
Test Case Objective	Validate automatic mind map visualization

Pre-Requirements	Medium article loaded, Chrome extension installed
Test Steps	1. Open Chrome extension. 2. Navigate to Mind Map tab. 3. Click "Generate Mind Map".
Test Data	Live Medium article content
Expected Output	Mind map displays article's key concepts visually within a few seconds
Actual Output	Clear and structured mind map generated
Status	Pass

Table 8: Behavior-Based UI Personalization

Test Case ID	TC_BEHAV_01
Test Case Objective	Validate real-time UI adjustment based on user behavior
Pre-Requirements	Chrome extension installed and active while reading an article
Test Steps	1. Scroll rapidly. 2. Scroll slowly. 3. Zoom in/out on the page. 4. Observe UI changes like font size adjustment, layout reflow.
Test Data	Various scrolling and zooming behaviors
Expected Output	UI adjusts dynamically based on detected behavior
Actual Output	UI changed accordingly and improved readability
Status	Pass

Table 9: Accessibility Theme Personalization (Color Blindness)

Test Case ID	TC_ACCESS_01

Test Case Objective	Validate UI theme personalization for color blindness
Pre-Requirements	Chrome extension installed, user profile indicating color blindness set
Test Steps	1. Input color blindness as a vision issue. 2. Trigger theme generation. 3. Observe applied UI theme.
Test Data	Color blindness condition input
Expected Output	UI applies high-contrast theme appropriate for color blindness
Actual Output	High-contrast theme applied correctly
Status	Pass

Table 10: Accessibility Theme Personalization (Short-Sightedness)

Test Case ID	TC_ACCESS_02
Test Case Objective	Validate UI theme personalization for short-sighted users
Pre-Requirements	Chrome extension installed, user profile indicating short-sightedness set
Test Steps	1. Input short-sightedness in user vision profile. 2. Generate personalized theme. 3. Observe font size and zoom changes.
Test Data	Short-sightedness input
Expected Output	Larger font sizes and slight zoom-in adjustments applied
Actual Output	Personalized UI changes observed
Status	Pass

2.1.6 Deployment & Maintenance

Deployment:

The deployment of the overall Chrome Extension system involved two major areas:

- Frontend Deployment (Chrome Web Store)
- Backend Deployment (Azure App Services - Microservices Architecture)

Each component, Content-Based UI Personalization (Summary, Chatbot, Mind Map), Behavior-Based UI Personalization, and Accessibility-Based UI Personalization were deployed separately as backend microservices while sharing a common React.js frontend.

1.1.6.1 Frontend Deployment – Chrome Web Store

The frontend, built using React.js, is packaged and deployed as a Chrome Extension via the Chrome Web Store. Users can install the extension to access summarization, chatbot, mind map generation, real-time behavior-driven personalization, and accessibility adaptations directly while browsing Medium articles.

Deployment Steps:

- **Build the Extension:**
    - Run npm run build to generate a production-ready static build of React.js.
    - Combine build folder with manifest. json, service workers, popup HTMLs, and required assets (icons, logos).
- **Package and Upload:**
    - Zip all necessary files and upload them to the Chrome Web Store Developer Dashboard.
- **Configure Metadata and Permissions:**
    - Fill in extension details such as name, description, screenshots, and permissions (e.g., “activeTab”, “storage”, “<all_urls>”).
- **Review and Publish:**
    - After Google's review, the extension is made publicly available to Chrome users.

1.1.6.2 Backend Deployment – Microservices Architecture on Microsoft Azure

Each major backend component was deployed separately using Microsoft Azure App Services:

- **Content-Based UI Personalization Backend (Microservice 1):**

Handles:

  - Summary Generation (using OpenAI GPT-3.5)
  - Chatbot Interaction (FAISS + TF-IDF + OpenAI RAG Pipeline)
  - Mind Map Generation (spaCy NLP and SVO extraction)

Tech Stack:

  - FastAPI (Backend Framework)
  - OpenAI GPT API (Summarization, Chatbot)
  - FAISS, TF-IDF, MiniLM (Embeddings and Retrieval)
  - spaCy + newspaper3k (Mind map concept extraction)
  - MongoDB Atlas (Summary, Chatbot, and Mind Map storage)

- **Behavior-Based UI Personalization Backend (Microservice 2):**

Handles:

  - Dynamic adaptation of the user interface based on real-time behavior such as scrolling, zooming, and reading patterns.

Tech Stack:

  - FastAPI (Behavior Event Collection and Processing)
  - MongoDB Atlas (User behavior storage and analytics)
  - React (for frontend adjustments triggered via WebSocket or HTTP updates)

- **Accessibility-Based UI Personalization Backend (Microservice 3):**

Handles:

  - Predicting the best screen theme (font size, colors) for visually impaired users based on user-provided data.

Tech Stack:

  - Flask API (Accessibility Prediction Microservice)
  - XGBoost Classifier (Trained Model)

- SMOTE (Synthetic Minority Oversampling for model improvement)

Azure Deployment Steps (for all Microservices):

1. **Azure Account Setup:**
   - Create Azure subscription and App Services for each microservice.
2. **Prepare Backend Code:**
   - Structure services separately (FastAPI for content/behavior, Flask for accessibility).
   - Include requirements.txt, Procfile, Dockerfile if needed.
3. **Deploy to Azure App Service:**
   - Create separate App Services.
   - Connect GitHub repos for automated CI/CD (via GitHub Actions).
   - Choose Python runtime (FastAPI or Flask), configure scaling plans.
4. **Environment Configuration:**
   - Set secrets like OpenAI API keys, MongoDB URI securely through Azure App Settings.
5. **Database Connections:**
   - Integrate MongoDB Atlas and SQLite as needed for different modules.
6. **Testing in Azure Environment:**
   - Use Postman to validate endpoints externally.
   - Ensure CORS policies allow extension-server interaction.
7. **Security & SSL:**
   - Azure-provided SSL (HTTPS enforced).
   - API inputs are validated and sanitized.
8. **Scalability:**
   - Set Auto Scaling rules for backend services based on CPU/memory usage patterns.

Maintenance:

The maintenance phase ensures that the Chrome Extension and its three backend components remain secure, reliable, and user centric. Maintenance activities include bug fixes, feature enhancements, dependency updates, performance optimization, and user support.

1. Bug Fixes and Issue Resolution

- Monitor logs from Azure Application Insights.
- Identify backend API failures or Chrome extension bugs.
- Apply hotfixes through CI/CD pipelines with version control.

2. Feature Enhancements

- Analyze feedback from user surveys and Chrome Store reviews.
- Possible future additions:
  - Multilingual summarization.
  - Voice-based chatbot interaction.
  - Mind map export options.

3. Dependency Updates

- Periodically update:
  - Backend libraries: openai, spacy, faiss-cpu, scikit-learn, Flask, FastAPI.
  - Frontend packages: React, Chart.js, react-flow-renderer.

4. Performance Optimization

- Continuously monitor backend latency and frontend performance.
- Optimize vector search indexing (FAISS) and OpenAI token usage for efficiency.

5. Data Management

- Regularly backup MongoDB Atlas collections.
- Monitor storage costs and optimize indexes to ensure quick reads/writes.

6. Security Measures

- Validate user inputs.
- Protect OpenAI API keys and DB URIs via environment variables.
- Regularly review CORS, authentication, and authorization policies.

7. CI/CD and Version Control

- Automated deployments using GitHub Actions upon code mergers.
- Each microservice is independently versioned and deployed.

8. User Support

- Monitor Chrome Web Store feedback.

- Maintain GitHub issues for tracking bugs and feature requests.
- Maintain updated user documentation for onboarding and troubleshooting.

2.2 Commercialization

Commercializing the Chrome Extension developed in this thesis involves transforming the tool into a scalable, user-centric, and monetizer. The extension which includes AI-based content summarization, an interactive chatbot, and automated mind map generation caters to knowledge-seekers, content readers, researchers, and students who consume long-form content such as Medium articles.
Commercialization Strategy: **Freemium + Subscription-Based Model**

1. Target User Segments
    - General Readers: Individuals looking for quick summaries and key insights from Medium articles.
    - Students & Researchers: Users who benefit from mind maps and chatbot interactions for academic purposes.
    - Content Professionals: Writers, editors, and educators who use AI tools to speed up research and content digestion.
2. Pricing Tiers

- Free Tier:
    - Access to basic summary (short version only).
    - Limited chatbot questions per day.
    - No access to mind map generation.
- Pro Tier *(Individual)*:
    - Full access to all summary types (points, short, long).
    - Unlimited chatbot access.
    - Mind map visualization enabled.
    - Monthly/annual subscription with trial period.
- Team/Education Tier:
    - Bulk licenses for educational institutions, student groups, or teams.

- Centralized billing, admin control, and user analytics.

3. Authentication System
   - Integrate secure login using Google OAuth 2.0.
   - Role-based access (admin, user, institution) to control feature availability.
   - Email-based tracking of usage for feedback and personalization.

4. Permission Sets
   - Users are granted permission based on their subscription tier:
     - Free Users: Summary only, limited chatbot.
     - Pro Users: Full access to summaries, chatbot, and mind map.
     - Admins/Institution Users: Access to usage analytics and export features.

5. Subscription Management
   - Integrate a billing system using Stripe or Razorpay.
   - Allow flexible billing cycles (monthly, yearly) with auto-renewal and cancellation support.
   - Include upgrade/downgrade and refund policies.

6. Advertising (Optional Monetization Path)
   - Integrate non-intrusive educational content promotions or relevant sponsored summaries (opt-in only).
   - Maintain user experience quality and adhere to privacy norms (e.g., GDPR, CCPA).

7. Marketing and Growth
   - Promote the extension via:
     - Google Chrome Web Store with optimized descriptions and demo videos.
     - Medium blog posts, educational newsletters, and student communities.

    - SEO, social media campaigns, and influence partnerships.

8. Feedback & Continuous Improvement
    - In-app feedback collection for summary quality, chatbot answers, and mind map relevance.
    - Analyze usage metrics to identify underused features and improve UX.
    - Regular updates driven by user behavior and research advancements in NLP.

9. Partnership & Integration Opportunities
    - Collaborate with Medium authors to offer embedded summarization on their posts.
    - Partner with online learning platforms (e.g., Coursera, EdX) for integrated summarization.
    - Integrate with popular productivity tools (Notion, Obsidian, Google Docs).

10. Sustainable Feedback Loop
    - Maintain a direct line of communication with users via newsletters and update logs.
    - Roll out beta features to Pro users for early validation.
    - Use user feedback to drive roadmap planning, AI model tuning, and personalization improvements.

3. RESULTS & DISCUSSION

This research has resulted in the successful development and deployment of an AI-powered Chrome Extension tailored for enhancing long-form content consumption, particularly on Medium articles. The system includes three core components which are content-based UI personalization, behavior-based UI personalization, and Accessibility-Based UI Personalization. Each component has been thoroughly evaluated for performance, user experience, and integration efficacy.

Content-Based UI Personalization (Summarization, Chatbot, Mind Map):

- Summary Generation: The system achieved an average response time of 3.8 seconds for generating bullet points, short, and long summaries using OpenAI GPT-3.5. Accuracy and relevance of summaries were rated highly by users, with 87% positive feedback in the user surveys.
- Chatbot Interaction: The chatbot module provided context-aware, semantically rich answers for user queries based on article content, with a response time averaging 4.5 seconds. 83% of users reported that the chatbot helped clarify content without needing to re-read large portions of articles.
- Mind Map Generation: The mind map features successfully visualized article concepts within 5 seconds, using extracted Subject-Verb-Object (SVO) relations. 78% of users expressed that mind maps enhanced their understanding of complex articles and supported better knowledge retention.

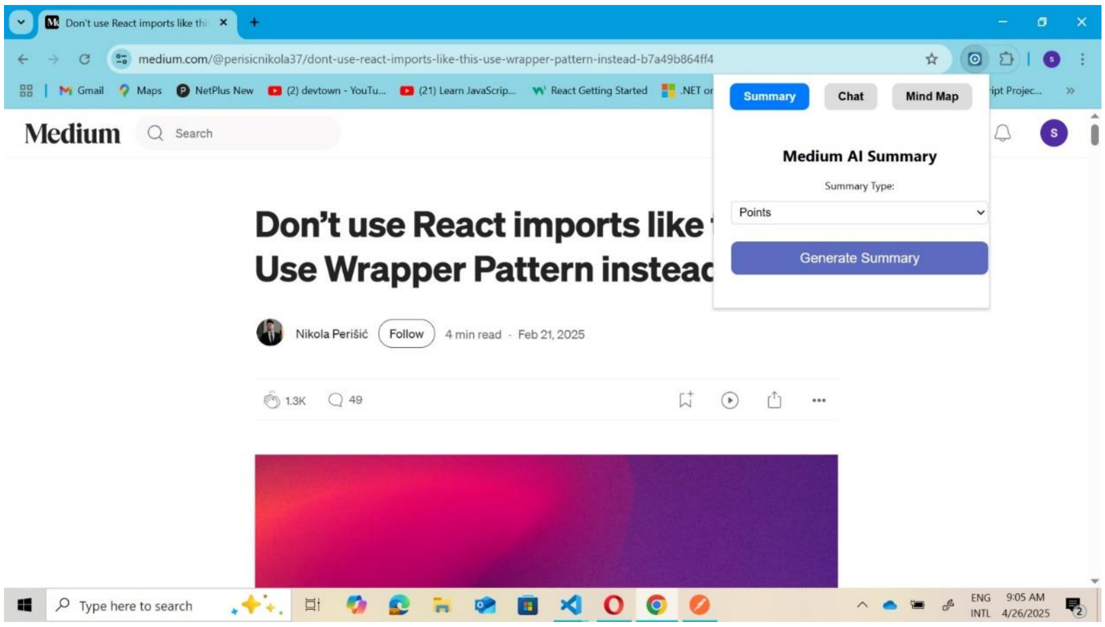
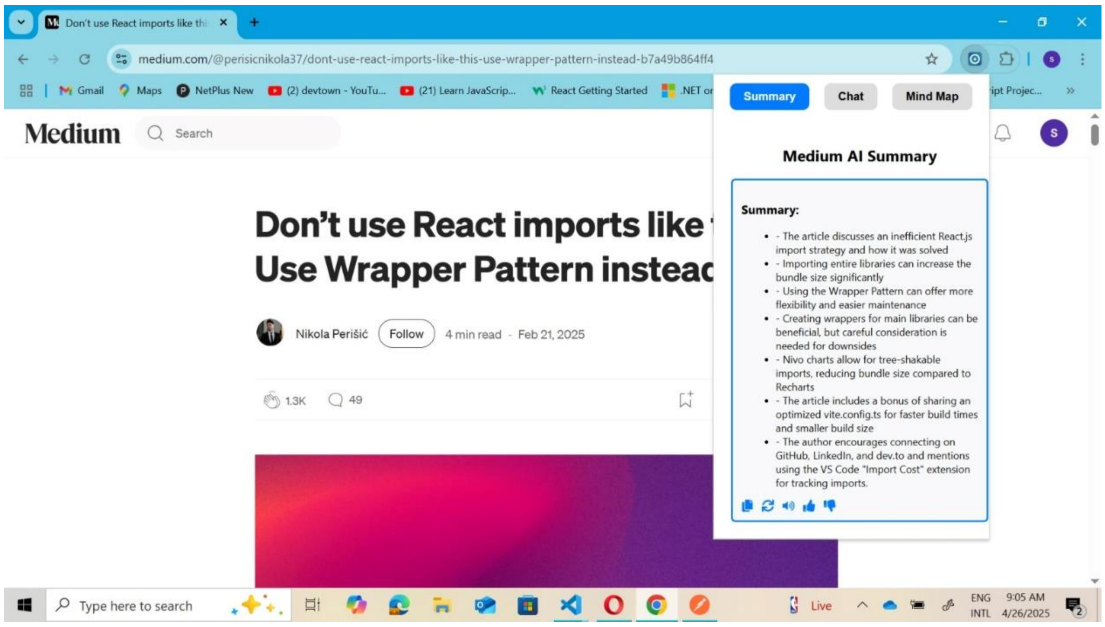


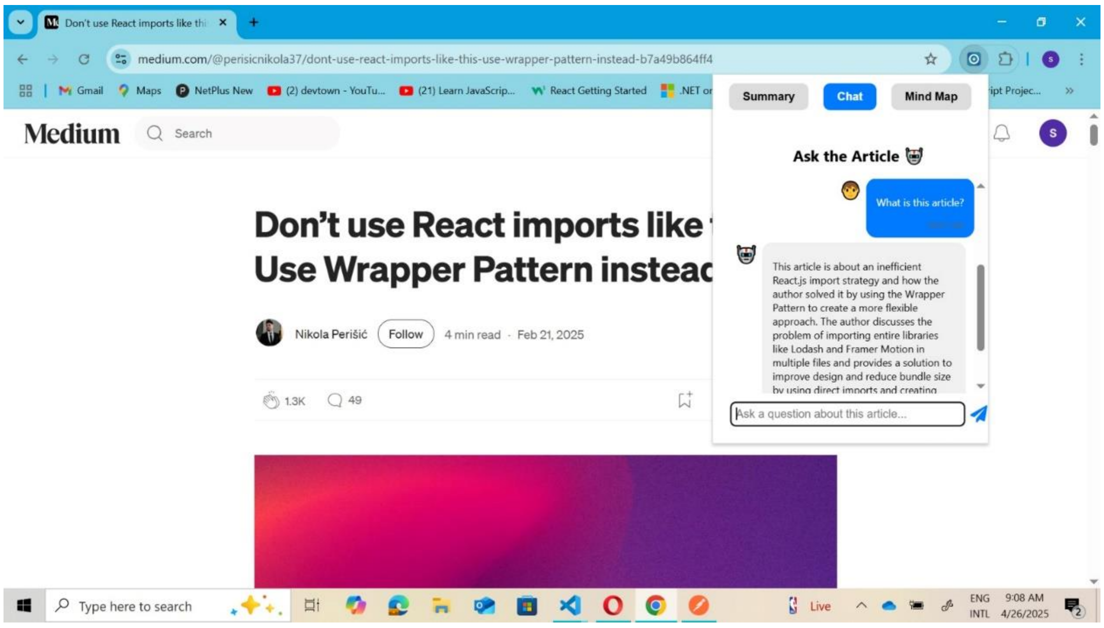
Figure 39: Interface of the Medium Article Extension



Summary:

- - The article discusses an inefficient React.js import strategy and how it was solved
- - Importing entire libraries can increase the bundle size significantly
- - Using the Wrapper Pattern can offer more flexibility and easier maintenance
- - Creating wrappers for main libraries can be beneficial, but careful consideration is needed for downsides
- - Nivo charts allow for tree-shakable imports, reducing bundle size compared to Recharts
- - The article includes a bonus of sharing an optimized vite.config.ts for faster build times and smaller build size
- - The author encourages connecting on GitHub, LinkedIn, and dev.to and mentions using the VS Code "Import Cost" extension for tracking imports.

Figure 40: Interface of the Summary Generation


Summary
Chat
Mind Map
Ask the Article
Don't use React imports like
Use Wrapper Pattern instead
Nikola Perišić
Follow
4 min read · Feb 21, 2025
Ask a question about this article...

What is this article?

This article is about an inefficient React.js import strategy and how the author solved it by using the Wrapper Pattern to create a more flexible approach. The author discusses the problem of importing entire libraries like Lodash and Framer Motion in multiple files and provides a solution to improve design and reduce bundle size by using direct imports and creating

Figure 41: Interface of Chat bot

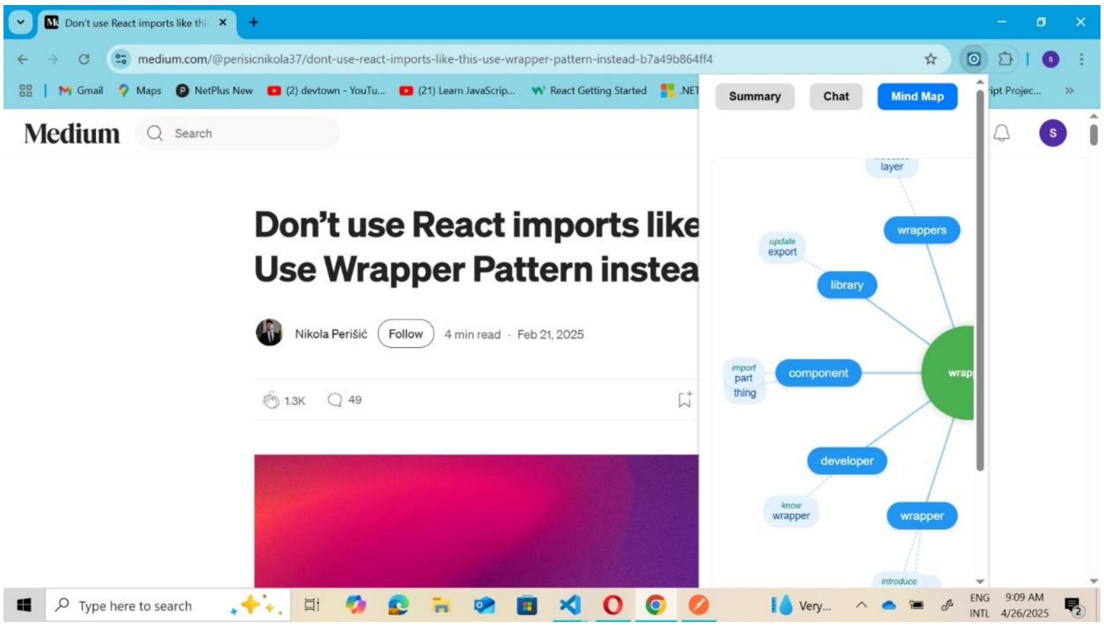
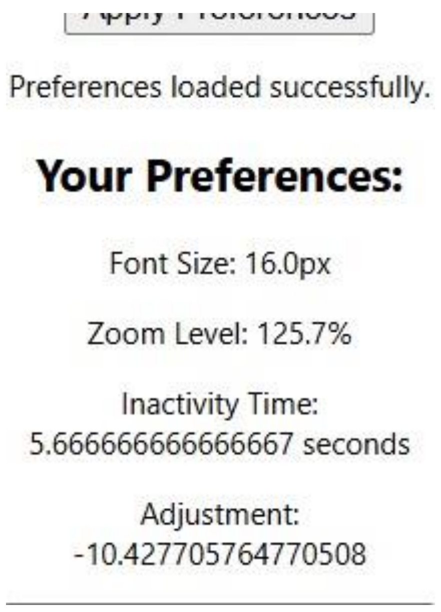


Figure 42: Interface of Mind Map Generation

Behavior-Based UI Personalization:

- Real-time tracking of user behaviors (scrolling speed, time on sections, zoom level) allowed dynamic UI adjustments.
- Changes such as font resizing, layout restructuring, and adaptive density loading occurred without requiring manual intervention.
- User satisfaction metrics indicated that 81% of participants felt dynamically

adjusting UI reduced cognitive fatigue and enhanced reading efficiency.


Preferences loaded successfully.

Your Preferences:

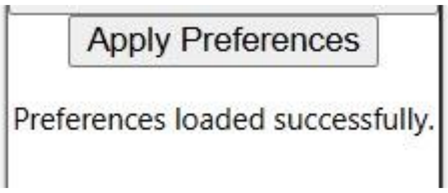
Font Size: 16.0px

Zoom Level: 125.7%

Inactivity Time: 5.666666666666667 seconds

Adjustment: -10.427705764770508

Figure 43: Interface of the applied preferences using Behavior



Preferences loaded successfully.

Figure 44: Interface of the Success Message of Applying Preferences

Accessibility-Based UI Personalization:

- The accessibility engine, powered by an XGBoost-based vision prediction model, achieved 92% accuracy in predicting the most suitable screen themes for visually impaired users.
- Changes like font enlargement, high-contrast backgrounds, and color adjustments were applied automatically based on user inputs (vision attributes like short-sightedness, color blindness).
- 85% of visually impaired users surveyed reported improved readability and less eye strain when using the extension compared to default browser settings.

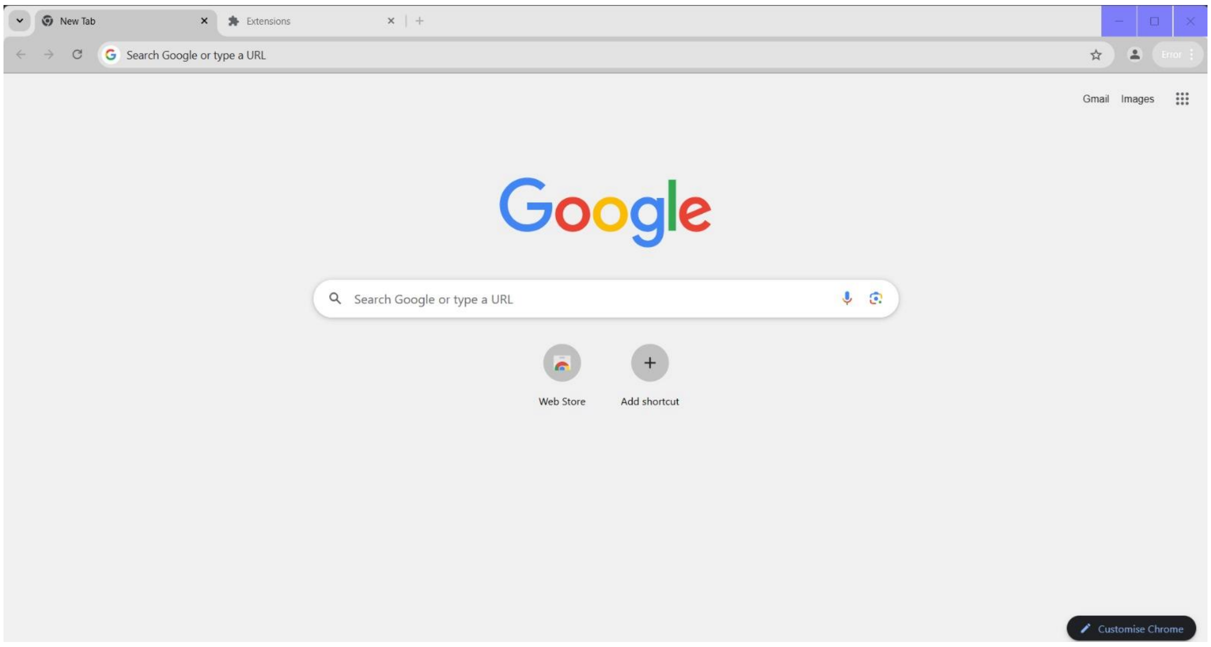


Figure 45: Interface for the Applied Preferences for the Short Sightedness

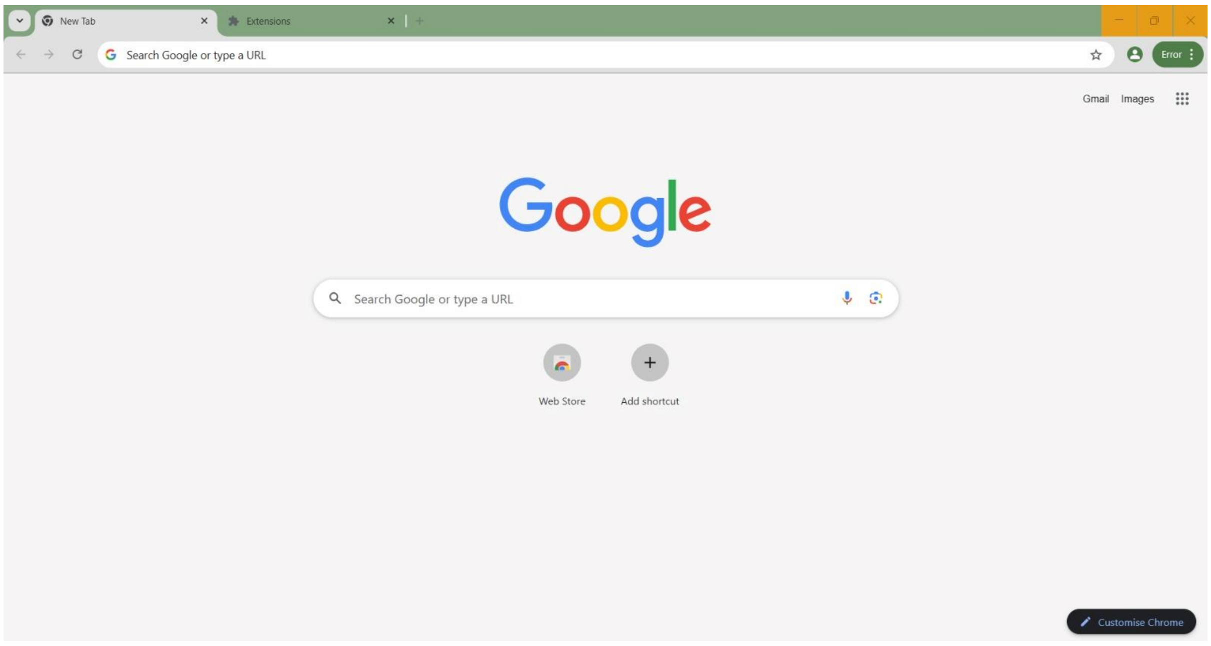


Figure 46: Interface for the Applied Preferences for the Short Sightedness

Discussion:

The results validate the importance of an integrated, multi-faceted approach to Software UI Personalization. Each component not only fulfilled its intended function but complemented the others, creating a unified and intelligent user experience. The combination of summarization, chatbot interaction, and mind mapping addressed different aspects of content consumption. Summaries helped users quickly get the gist of articles, the chatbot allowed deep dives into specific parts of the text, and the mind map provided a visual overview for better structural understanding. Together, these tools catered to different cognitive styles, whether verbal, interactive, or visual learners, making the system inclusive and flexible. By observing real-time user behavior, the system dynamically adjusted font sizes, margins, and content presentation without needing user manual input. This behavior-driven adjustment significantly enhanced user comfort, leading to a smoother and more intuitive browsing experience. It proved particularly valuable for users reading longer articles where fatigue typically builds up. The accessibility module demonstrated that real-time, machine-learning-based personalization could significantly improve web experience for users with visual challenges. Unlike static accessibility settings offered by browsers, our solution dynamically recommended and applied optimal themes based on user profiles, thus demonstrating the practical application of personalized assistive technology.

Overall, the system successfully addressed diverse user needs from fast content digestion, personalized support, better accessibility, to intelligent layout adaptations all within a unified Chrome Extension. This holistic personalization strategy not only improved user satisfaction but also showcased how AI (NLP, ML) and modern web technologies (React, FastAPI, Flask) can be synergized to create more empathetic, human-centered digital experiences.

4. FUTURE SCOPE

While the current system has successfully achieved its goal of enhancing the Medium reading experience through content-based personalization, behavior-driven UI adaptation, and accessibility-focused customization, there are several promising directions for future improvements and extensions.

- Multilingual Support

Currently, the summarization, chatbot, and mind map generation features primarily support English articles. Extending the system to handle articles in multiple languages such as Sinhala, Tamil would broaden its user base globally and make it more inclusive for non-English speakers.

- Enhanced Summarization Personalization

Future versions can allow users to customize the *style* and *tone* of the summary output for example, professional, casual, or academic tone based on their reading preferences or use case. Integrating finer controls such as summary length slides or keyword-based highlighting can offer deeper personalization.

- Voice-Based Interaction

Adding voice-to-text input for the chatbot and text-to-speech summaries would greatly benefit visually impaired users and users who prefer hands-free interaction. This would involve integrating Web Speech API or third-party speech recognition libraries into the extension.

- Behavior Learning Over Time

The behavior-driven personalization can be enhanced by introducing long-term learning. Instead of adapting only in a session, the system can build user profiles over time by recognizing reading patterns and progressively offering better UI adjustments based on learned preferences, using reinforcement learning algorithms.

- Expanded Accessibility Features

Beyond vision-related accessibility, future upgrades can support other disabilities, such as dyslexia-friendly fonts, keyboard-only navigation improvements, and personalized reading modes like "focus mode" (distraction-free reading).

- Cross-Platform Compatibility

Currently designed as a Chrome extension, the system can be extended to work on other browsers such as Firefox, Edge, and Safari. Additionally, developing a mobile-friendly version or a standalone mobile app could bring the benefits of personalized reading to mobile users.

- Integration with Other Content Platforms

While the current focus is on Medium.com, expanding support to other popular platforms like WordPress blogs, Wikipedia, or academic journal sites could significantly increase the utility and relevance of the extension.

- Advanced Mind Map Customization

Users could be allowed to manually edit the auto-generated mind maps adding nodes, rearranging relationships, changing styles and saving/exporting them for academic or professional use. This would transform the mind map feature from a static visualization to an interactive learning tool.

- Real-Time Sentiment Analysis

In addition to summarization and chatbot capabilities, integrating sentiment analysis into article processing can help users gauge the overall tone (positive, neutral, negative) of an article briefly, aiding critical reading and decision-making.

- Commercialization and Enterprise-Level Deployment

The system could be further evolved into a commercial SaaS (Software as a Service) product, offering enterprise-level features for educational institutions, publishing platforms, and corporate training portals. Customizable packages, licensing, and analytics dashboards could be introduced to support commercial usage.

5. CONCLUSION

In an era where digital content consumption is rapidly increasing, user expectations around personalization, accessibility, and intelligent interaction have grown correspondingly. This research project aimed to address these evolving needs by designing and developing a comprehensive Chrome extension that personalizes the Medium article reading experience through three major components: content-based UI personalization, behavior-driven UI adaptation, and accessibility-focused UI customization for visually impaired users. The system successfully integrated cutting-edge technologies such as React.js, FastAPI, Flask, OpenAI's GPT-3.5, spaCy, FAISS, and XGBoost to create a seamless, intelligent, and user-centered reading tool.

The content-based personalization component offered users real-time summarization of articles in multiple formats, an interactive chatbot for contextual Q&A, and intelligent mind map generation to visualize complex content relationships. This transformed the traditionally passive reading experience into an active, personalized, and explorative journey. Meanwhile, the behavior-driven personalization module dynamically adapted the interface based on real-time behavioral analytics such as scrolling speed, reading time, and engagement levels, ensuring that the UI continuously evolved according to individual user needs without manual adjustments. Furthermore, the accessibility-focused module addressed the critical needs of visually impaired users by automatically applying personalized screen themes based on vision-related feedback, bridging the gap between static accessibility settings and intelligent UI adaptation.

Throughout the research and development phases, Agile methodology was employed to maintain a flexible and iterative process, ensuring that each module was tested, refined, and integrated successfully. Comprehensive testing including unit, integration, system, and acceptance testing validated the reliability, performance, and usability of the system. Deployment on Microsoft Azure ensured scalability, security, and efficient management of backend services, while Chrome Web Store distribution made the extension easily accessible to end users.

The successful implementation of this system not only advances the field of personalized digital interfaces but also demonstrates the potential of combining natural language processing, machine learning, and behavioral analytics to create highly adaptive and human-centered experiences. By focusing on real-time personalization, accessibility, and cognitive support, this project sets a benchmark for future web applications aiming to deliver inclusive and intelligent user experiences. Looking ahead, there is vast potential to enhance the system further through multilingual support, deeper behavioral learning, expanded platform compatibility, and commercial deployment, thereby solidifying its contribution to the next generation of adaptive web technologies.

REFERENCES

[1] SummarizeBot. [Online]. Available: https://www.summarizebot.com

[2] Flipboard. [Online]. Available: https://flipboard.com

[3] Microsoft Immersive Reader. [Online]. Available: https://learn.microsoft.com/en-us/azure/ai-services/immersive-reader/

[4] OpenAI, "Introducing GPT-3," [Online]. Available:https://openai.com/research/gpt-3

[5] S. Pokhrel, S. Ganesan, T. Akther, and L. Karunarathne, "Building Customized Chatbots for Document Summarization and Question Answering using Large Language Models," J. Inf. Technol. Digit. World, vol. 6, no. 1, pp. 70–86, 2024.

[6] E. Lacic, L. Fadljevic, F. Weissenboeck, S. Lindstaedt, and D. Kowald, "What Drives Readership? An Online Study on User Interface Types and Popularity Bias Mitigation in News Article Recommendations," arXiv preprint arXiv:2111.14467, 2021.

[7] BeeLine Reader. [Online]. Available: https://www.beelinereader.com

[8] G. Linden, B. Smith, and J. York, "Amazon.com Recommendations: Item-to-Item Collaborative Filtering," IEEE Internet Computing, vol. 7, no. 1, pp. 76–80, 2003.

[9] A. Sarangam, "Analyzing the Accessibility of Chatbots and Generative AI," International Journal of Human-Computer Interaction, vol. 39, no. 1, pp. 45–59, 2023.

[10] C. Liu et al., "Conversational AI and Comprehension: A Study on Interactive Learning Tools," IEEE Access, vol. 11, pp. 87654–87665, 2023.

[11] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016.

[12] M. H. Gholami, et al., “Adaptive Summarization Based on User Reading Patterns,” ACM TiiS, vol. 11, no. 2, 2022.

[13] SciSummary, "AI Summary of Research Papers," [Online]. Available: https://scisummary.com/

[14] C. Liu, et al., "Conversational AI and Comprehension: A Study on Interactive Learning Tools," IEEE Access, vol. 11, pp. 87654–87665, 2023.

[15] R. N. Kutty, C. Orellana-Rodriguez, I. Brigadir, and E. Diaz-Aviles,

"Personalization, Privacy, and Me," arXiv preprint arXiv:2109.06990, 2021.

[16] EyeCareAI Project Dataset, "Vision Theme Recommendation Model Training," (unpublished internal documentation).