

CHAPTER 1: INTRODUCTION

In a rapidly digitalizing world, website navigation is crucial to delivering an exceptional user experience (UX) and enhancing user engagement. Traditional search functionalities, though useful, often fail to grasp complex or nuanced natural language queries, resulting in inefficient browsing experiences. Navify, an AI-driven web extension, addresses this challenge by allowing users to interact with websites through natural language, creating a seamless navigation experience tailored to user intent.

Navify enables users to input queries or commands in everyday language, which are processed by an AI model trained to understand context and intent. This innovation not only reduces the time users spend searching for information but also personalizes the browsing experience. By leveraging deep learning and AI, Navify's chatbot-like interface directs users to relevant sections of a website efficiently, ensuring faster, more intuitive navigation.

Navify's architecture combines React for the front-end interface with Node.js and Express on the back end. This design ensures a robust infrastructure where user queries are quickly interpreted and responded to via Axios-based API calls and JSON data formatting. Together, these technologies create a responsive system that enhances user satisfaction and data integrity.

One of Navify's key features is its adaptive learning capability, which enables it to refine its responses over time based on user interactions. This functionality supports its versatility across a range of websites, adapting to different structures and user patterns.

Ultimately, Navify aims to set a new standard for web navigation, providing users with a streamlined, natural interaction method that aligns with modern digital expectations and enhances overall website usability.

CHAPTER 2: SYSTEM ANALYSIS

System analysis in the context of Navify, an AI-driven web extension, involves evaluating its components, functionality, and overall user experience. The goal of this analysis is to ensure that Navify can enhance website navigation efficiently and reliably by understanding user intent and directing them to relevant information within a website.

Problem Identification: Traditional website navigation relies on static menus and search bars, which are often ineffective in capturing complex user queries. Users need a way to navigate websites more intuitively, without knowing the exact structure of the site. Navify addresses this by interpreting natural language inputs and simplifying access to specific information.

Requirement Gathering: The project requires a system capable of processing natural language inputs, understanding user intent, and mapping these inputs to relevant sections of a website. The front end needs to be user-friendly, enabling seamless interaction, while the back end requires an AI model capable of parsing and responding to queries with high accuracy.

System Architecture: Navify's architecture involves a React-based front end for the user interface, a Node.js and Express backend to handle requests, and an AI model trained to understand natural language. The system leverages Axios for API requests and JSON for data interchange, ensuring fast and smooth communication between the user and the server.

Functionality Testing: System testing will ensure the AI model's accuracy in interpreting diverse user queries and the reliability of the back end in handling requests. Additionally, the interface must be tested for usability to confirm an intuitive user experience.

Security and Privacy: As Navify handles user data, data security protocols are implemented to ensure user privacy and data integrity.

2.1 General

The general section of system analysis provides an overview of the project's purpose, structure, and expected outcomes. In the context of Navify, an AI-driven web extension, system analysis focuses on developing a solution that allows users to navigate websites more intuitively and effectively. Traditional website navigation often relies on static menus or search bars, which may not be user-friendly or effective for complex, specific queries. Navify aims to enhance user experience by utilising natural language processing (NLP) to understand users' intent and guide them directly to the most relevant parts of a website.

To achieve this, the system is designed with a modular architecture, which includes a front end built in React, a backend API with Node.js and Express, and a machine learning model capable of interpreting user queries. This modular setup enables flexibility in adding new features or improving individual components without affecting the entire system. Additionally, Navify incorporates essential security protocols to protect user data and ensure compliance with data privacy standards. This general overview of Navify's system analysis sets the foundation for more detailed discussions on preliminary investigation, feasibility study, and design considerations, aligning the project's technical aspects with its ultimate goals: providing efficient, user-friendly website navigation and improved user engagement.

2.2 Preliminary Investigation

The preliminary investigation stage focuses on identifying the core problem, understanding project requirements, and evaluating potential solutions. For Navify, this phase addresses the need for a more sophisticated method of website navigation. In conventional website navigation systems, users are often limited by rigid search functions and menus, making it challenging to locate specific information or perform detailed searches. The preliminary investigation identified this gap as a significant barrier to efficient website usability and customer satisfaction.

To solve this, the Navify project team conducted research into natural language processing (NLP) as a potential solution to allow users to make more complex, nuanced queries. The investigation included user interviews, competitor analysis, and surveys to assess the effectiveness of current website navigation methods. This feedback helped refine the project requirements for Navify, particularly the need for accurate NLP interpretation, integration with existing website structures, and a user-friendly interface.

Moreover, initial investigations explored the technological requirements, such as AI models, API structures, and front-end integration, to achieve a seamless experience. The outcomes from this stage provided critical insights, informing the development and guiding system design to ensure that Navify effectively addresses real user needs by bridging the gap between user intent and website navigation capabilities.

2.3 Feasibility Study

The feasibility study evaluates the technical, economic, and operational viability of implementing Navify as a practical solution for enhanced website navigation.

2.3.1 Technical Feasibility

Technically, the project requires an AI model with robust natural language processing (NLP) capabilities to interpret diverse user queries. Given advancements in machine learning, especially in NLP, implementing this functionality is feasible. Open-source NLP libraries, such as SpaCy and Hugging Face, can be used to reduce development costs and speed up implementation, allowing for cost-effective technical feasibility.

2.3.2 Economical Feasibility

The feasibility study assessed Navify's potential costs, including development, deployment, and maintenance. The project would initially require investment in data processing infrastructure and machine learning training. However, the use of open-source tools and cloud-based deployment options offers cost-effective solutions, making it feasible for small to medium enterprises to adopt Navify as a low-cost solution to improve user engagement on their websites.

2.3.3 Operational Feasibility

Operationally, the feasibility study confirmed the ease of integration with existing websites. Navify can be deployed as a web extension, making it adaptable across various platforms without significant changes to backend systems. Additionally, user training requirements are

minimal due to its intuitive interface. Thus, from a technical, economic, and operational perspective, Navify is a feasible project with strong potential to meet user needs and deliver measurable benefits in website navigation and user satisfaction.

2.4 Software and Hardware Specification

Software Requirements:

- Operating System: Windows 10, macOS, or Linux
- Programming Languages: JavaScript (React.js for frontend), Node.js for backend
- Frontend: React.js (for creating the interactive UI)
- Backend: Express.js (for handling API requests)
- API Calls: Axios (for making API calls)
- Database: MongoDB (for storing user interactions and query data)
- LLM (Large Language Model): Llama (for keyword extraction and sentiment analysis)
- Code Editor: Visual Studio Code
- Version Control: Git, GitHub (for project version control and collaboration)
- Testing Tools: Jest (for unit testing), Postman (for API testing)
- Browser: Chrome, Firefox, or any modern browser (for testing and user experience validation)

Hardware Requirements:

- Processor: Intel i5 or equivalent, minimum (recommended i7 or higher for optimal performance)
- RAM: 8 GB minimum (recommended 16 GB or higher for smooth performance with multitasking)
- Storage: 500 GB (SSD recommended for faster processing, especially during development)

- Internet Connection: Stable high-speed connection (required for model training and API interactions)
- Graphics Card: Optional, but a dedicated GPU may help in processing large datasets for model training

2.5 Data Flow Diagram

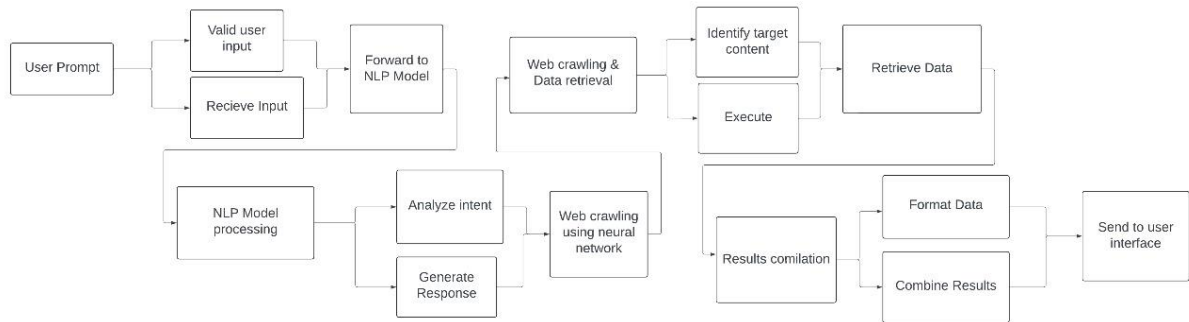


Figure 1. Data Flow Diagram of Navify

CHAPTER 3: SYSTEM DESIGN

3.1 Design Methodology

The design methodology for this monolithic project focuses on creating a unified, streamlined system where all components of the application are integrated within a single codebase. This architecture provides simplicity in deployment and maintenance, reducing the complexity often associated with distributed systems. The monolithic design is suitable here because it allows for easier handling of interdependent processes and efficient data management.

To ensure the project's functionality and scalability, the architecture uses a layered structure, separating the business logic, user interface, and data access layers. This approach improves code reusability, maintainability, and testability, allowing updates or changes within one layer without disrupting others. The backend processes, including Llama-based keyword extraction, website crawling, and sentiment analysis, are encapsulated within the application to minimize latency and improve performance. Data flow is managed through well-defined APIs that handle communication between components seamlessly.

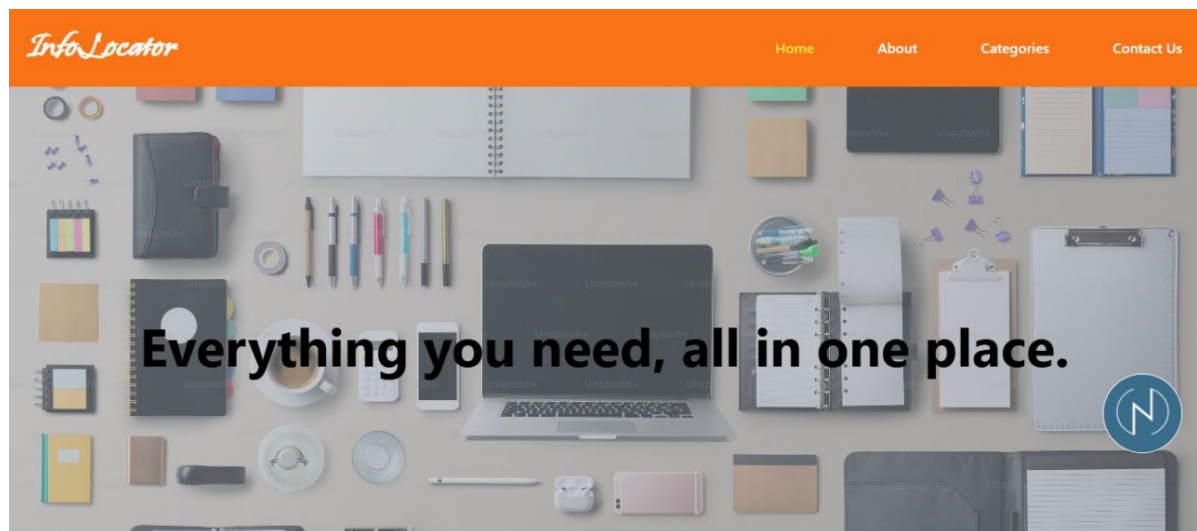
Using best practices like modularization, the design ensures the system is flexible, allowing for future integrations and upgrades. The monolithic approach was chosen to avoid the complexity of managing multiple services, providing a cohesive experience for both development and deployment while meeting the project requirements efficiently.

3.2 User Interface Design

The User Interface (UI) design for this project prioritizes user engagement and intuitiveness, addressing issues related to poor user retention often observed in websites with complex navigation or inadequate search functionality. The UI centers around a prompt-based chatbox that allows users to enter natural language queries for their search, making it easy and accessible. The chatbox is prominently placed to attract users' attention, while the design ensures clear visibility and ease of use.

Key design elements include the use of vibrant colors, tooltips, and a responsive layout that works across devices. The chatbox provides instant feedback, displaying top results and relevant content based on Llama-driven keyword extraction and sentiment analysis. Each result includes a clickable link, enabling users to navigate directly to relevant sections of the website effortlessly.

UI elements such as icons, buttons, and links are styled for consistency, aligning with best practices to enhance the overall aesthetic and usability. By simplifying navigation and highlighting the most relevant content, the UI design aims to reduce friction in the user journey, making it more likely for users to return, thus addressing the primary project goal of improving user retention.



About

StockSeeker is your ultimate destination for a vast collection of stock resources and data. Our goal is to provide you with everything you need to make informed decisions and stay ahead in the fast-paced world. Whether you're a student, professional, hobbyist, or simply seeking to expand your knowledge, our platform offers a wealth of resources to explore. From academic subjects and technical skills to personal interests and hobbies, we have something for everyone.

Categories

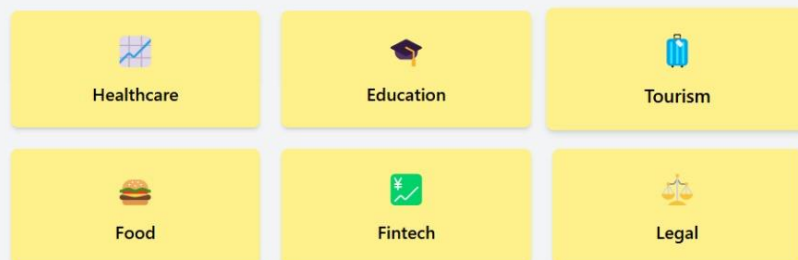


Figure 2. UI Design of Website

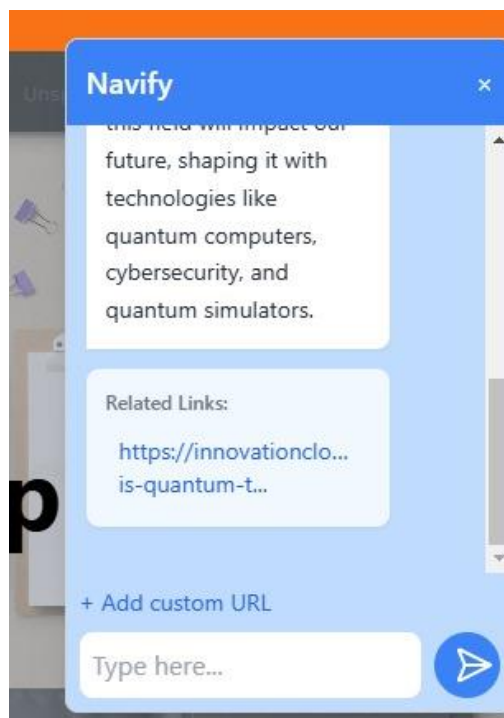
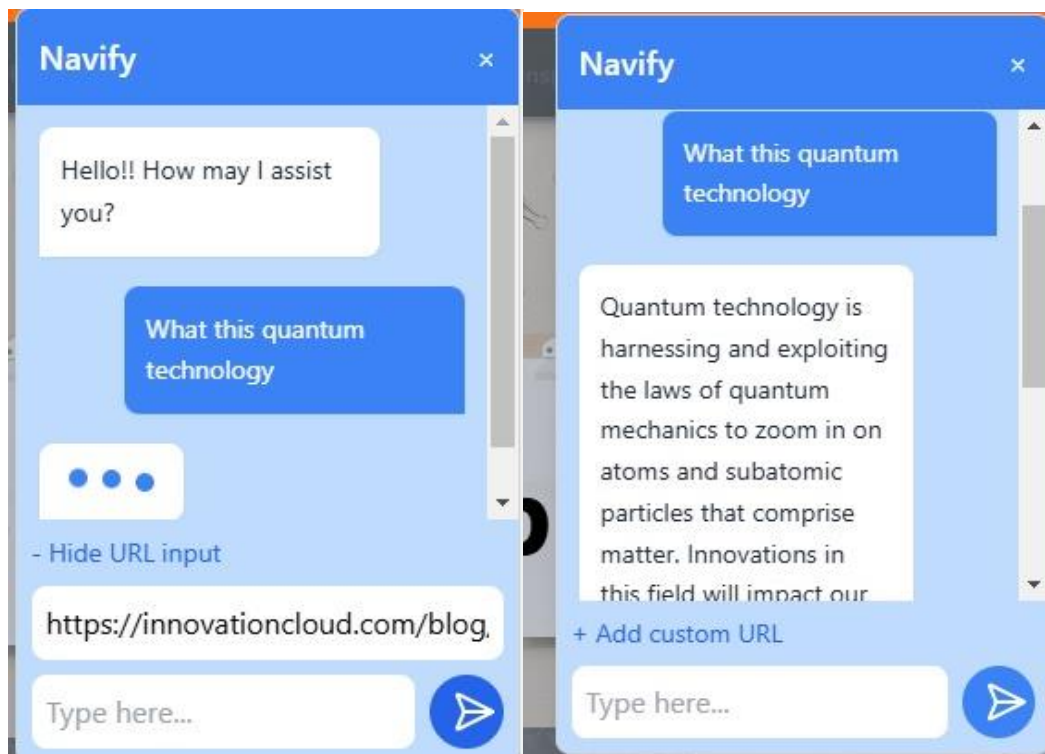


Figure 3. UI Design of Navify

CHAPTER 4: TESTING

4.1 Testing Techniques & Strategies

To ensure the reliability and performance of this monolithic application, a comprehensive set of testing techniques and strategies are employed. Given the complexity of functionalities, such as natural language processing, keyword extraction, and sentiment analysis, multiple testing approaches are necessary. The primary techniques used include:

Unit Testing: Each individual function or module is tested separately to verify its correctness. This includes testing the Llama-based keyword extraction, sentiment analysis algorithms, and backend API endpoints.

Integration Testing: Since this is a monolithic application, integration testing is crucial to ensure that various components work cohesively. It focuses on the interaction between the UI, backend, and database layers to detect any faults in data flow and dependencies.

End-to-End (E2E) Testing: This approach tests the entire workflow from start to finish. For instance, testing involves entering a query in the prompt, processing it through the backend, and finally validating the results and links returned on the frontend. This helps ensure that the application performs as expected from a user's perspective.

Performance Testing: Given that the project needs to quickly deliver relevant search results, performance testing assesses response times, system stability under load, and latency in data processing.

User Acceptance Testing (UAT): In this phase, potential users or stakeholders test the system to verify if it meets the desired requirements and is user-friendly.

4.2 Debugging & Code Improvement

Debugging and code improvement are essential components of the development cycle to maintain code quality and system stability. Throughout the development process, tools such as integrated debugging environments (IDEs), logging, and error-tracking systems like Sentry are utilized to identify and resolve bugs. Debugging begins with identifying potential problem areas within the code, often revealed through failed unit or integration tests. Techniques such as breakpoints, step-through debugging, and logging statements are used extensively to track down errors in the execution flow.

To improve code quality, best practices such as code reviews and refactoring are implemented. Code reviews ensure that multiple developers examine the code, catching potential issues or inefficiencies early. Refactoring is done to optimize and simplify code without altering its functionality, making it more readable, maintainable, and efficient. For example, refactoring can involve removing redundant code, optimizing database queries, or improving the modularity of the codebase.

Additionally, performance monitoring is used to detect bottlenecks, especially in resource-intensive processes like crawling and sentiment analysis. Improvements are then made by optimizing algorithms, caching frequently accessed data, or enhancing data structures. Debugging and code improvement contribute to a smoother, more efficient application, ultimately enhancing user satisfaction and retention.

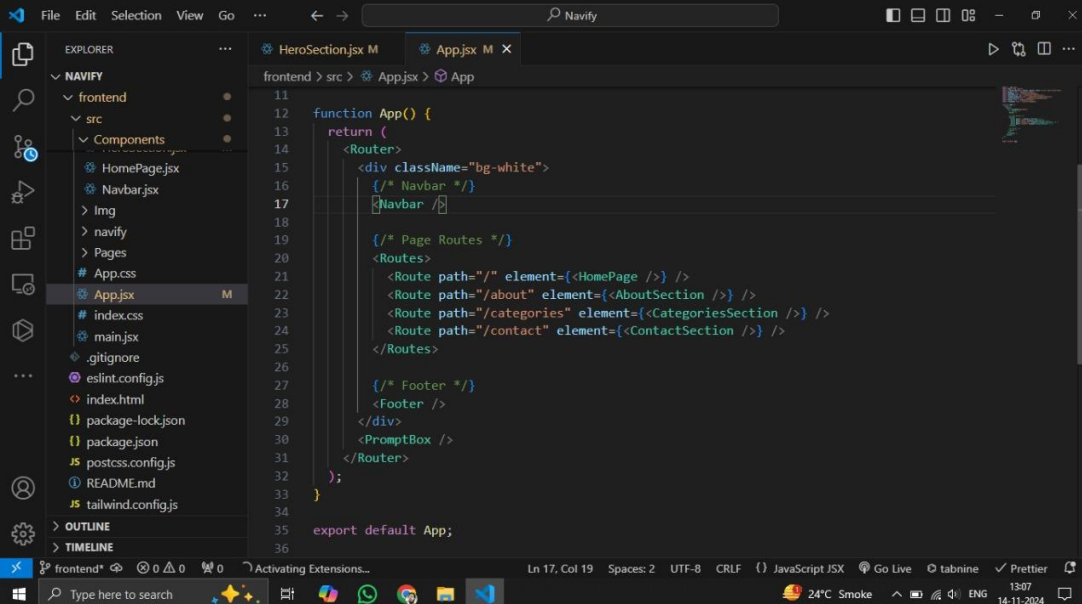
CHAPTER 5: IMPLEMENTATION

5.1 System Implementation

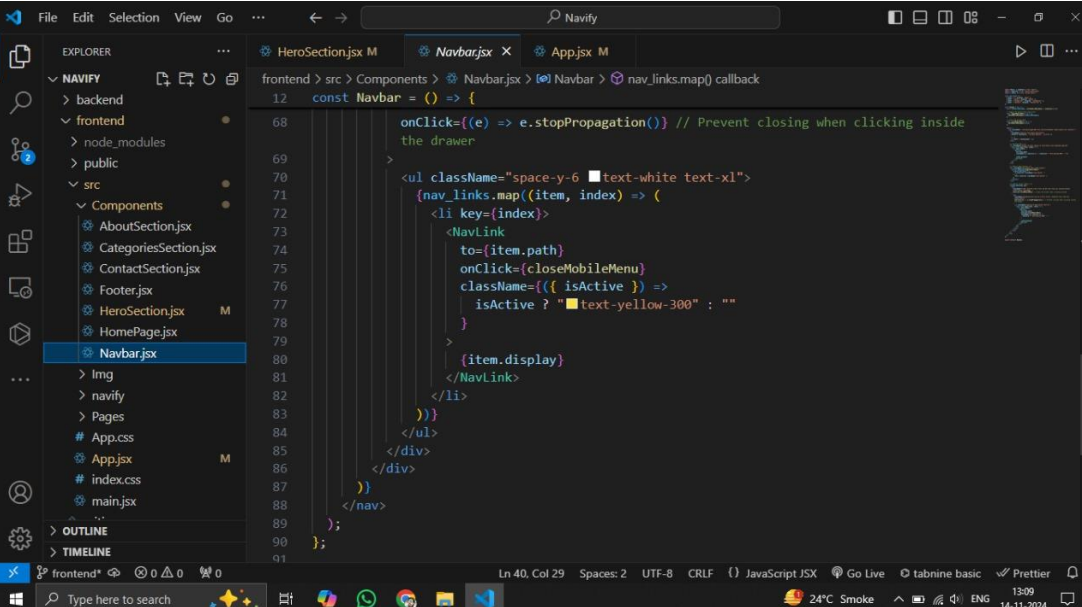
The implementation phase involves integrating various components of the monolithic application to deliver a seamless user experience. In this project, system implementation includes the setup and configuration of the backend server, the frontend user interface, and the database. The backend is powered by a natural language model (Llama) for keyword extraction and sentiment analysis, which provides users with relevant search results in real time. The backend also incorporates web crawling to analyze website content for relevance and ranks it based on sentiment scoring. The system is designed to handle user queries efficiently by leveraging these integrated components, thus providing accurate results to retain users. The monolithic structure simplifies deployment and management by consolidating all application layers into a single codebase.

5.2 Software Implementation

Frontend



```
11
12 function App() {
13   return (
14     <Router>
15       <div className="bg-white">
16         <Navbar />
17       </div>
18
19       </* Page Routes */>
20       <Routes>
21         <Route path="/" element={<HomePage />} />
22         <Route path="/about" element={<AboutSection />} />
23         <Route path="/categories" element={<CategoriesSection />} />
24         <Route path="/contact" element={<ContactSection />} />
25       </Routes>
26
27       </* Footer */>
28       <Footer />
29     </div>
30     <PromptBox />
31   </Router>
32 );
33
34 export default App;
```



```
12 const Nav = () => {
68
69   onClick={(e) => e.stopPropagation()} // Prevent closing when clicking inside
70   the drawer
71
72   <ul className="space-y-6 text-white text-xl">
73     {nav_links.map((item, index) => (
74       <li key={index}>
75         <NavLink
76           to={item.path}
77           onClick={closeMobileMenu}
78           className={({ isActive }) =>
79             isActive ? "text-yellow-300" : ""
80           }
81           {item.display}
82         </NavLink>
83       </li>
84     ))}
85   </ul>
86 </div>
87
88 </nav>
89
90 );
91
```

This screenshot shows the Visual Studio Code editor with the 'Navify' project open. The Explorer panel on the left shows the file structure, with 'PromptBox.jsx' selected under the 'navify' directory. The main editor displays the code for 'PromptBox', which is a React component that handles sending messages to an API. The code includes a stateful input field, a 'handleSendMessage' function that uses 'axios' to post data to 'http://localhost:8000/api/v1/navify', and logic to update the state with the response data. The status bar at the bottom indicates the file is at line 83, column 31, using UTF-8 encoding and CRLF line endings.

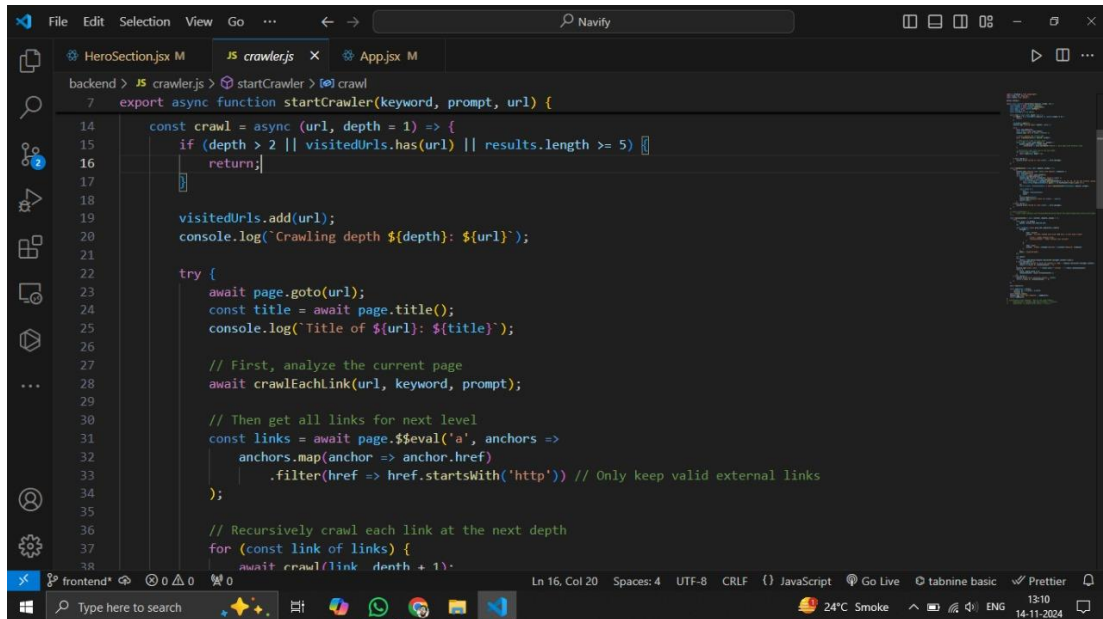
```
6 const PromptBox = () => {
21   const handleSendMessage = async () => {
32
33     const urlToUse = customUrl || window.location.href;
34
35     try {
36       const response = await axios.post("http://localhost:8000/api/v1/navify", {
37         prompt: currentInput,
38         url: urlToUse,
39       });
40
41       if (response.status === 200 && response.data) {
42         const content = response.data.results[0].content;
43
44         const uniqueUrls = [
45           ...new Set(response.data.results.map((result) => result.url)),
46         ];
47         const links = uniqueUrls.map((url) => ({
48           url: url,
49           label: url,
50         }));
51
52         setMessages((prevMessages) => [
53           ...prevMessages,
54           {
55             type: "bot",
```

This screenshot shows the Visual Studio Code editor with the 'Navify' project open. The Explorer panel on the left shows the file structure, with 'Footer.jsx' selected under the 'Components' directory. The main editor displays the code for 'Footer', which is a React component that renders the footer of the application. The code uses Tailwind CSS classes for styling and includes links to the project's GitHub, YouTube, and Instagram profiles. The status bar at the bottom indicates the file is at line 9, column 108, using UTF-8 encoding and CRLF line endings.

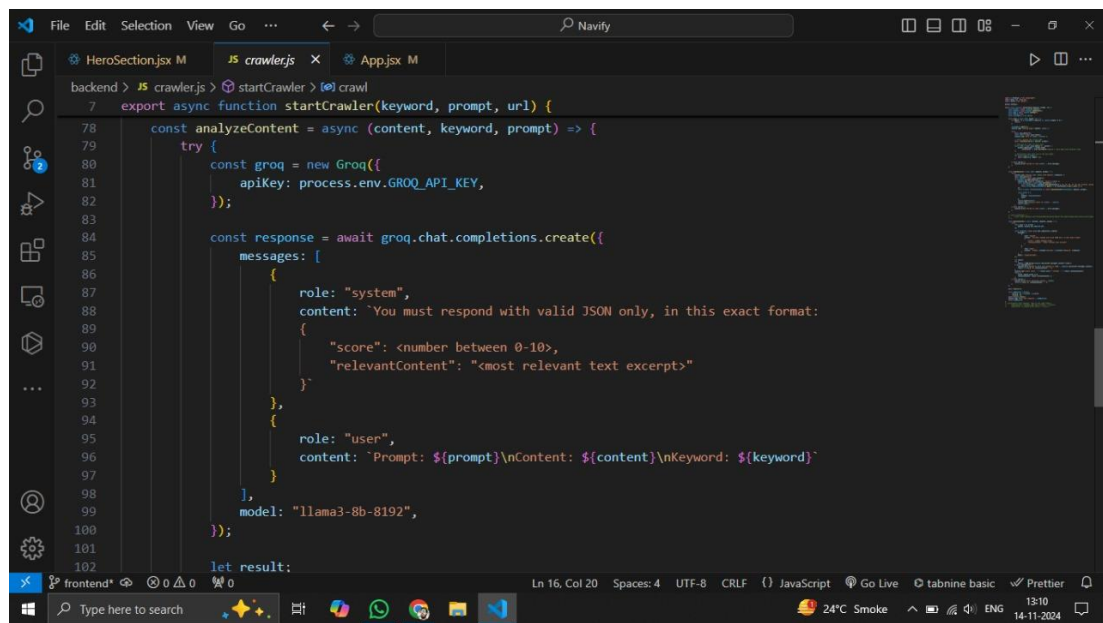
```
3 const Footer = () => {
4   return (
5     <footer className="bg-orange-500 py-10 px-8 md:px-16 text-white text-center">
6       <div className="flex flex-col md:flex-row justify-between items-center space-y-8 md:space-y-0 md:space-x-4">
7         {/* logo section */}
8         <div className="text-center">
9           <img alt="Logo" data-bbox="380 525 420 555"/>
10          <h3 className="font-semibold">Connect On</h3>
11        </div>
12        <div className="flex space-x-4 justify-center mt-2">
13          <a href="#" className="text-xl"><i className="ri-youtube-line"></i></a>
14          <a href="#" className="text-xl"><i className="ri-github-fill"></i></a>
15          <a href="#" className="text-xl"><i className="ri-instagram-line"></i></a>
16        </div>
17      </div>
18
19      {/* links section */}
20      <div className="text-white text-center">
21        <h5 className="Footer_link-title text-2xl font-semibold mb-4">Quick Links</h5>
22        <div className="font-semibold mb-0 flex flex-col items-center gap-3">
23          <p>Home</p>
24          <p>About</p>
25          <p>Categories</p>
26        </div>
27      </div>
28    </footer>
29  );
30}
```

Figure 4. Frontend Code

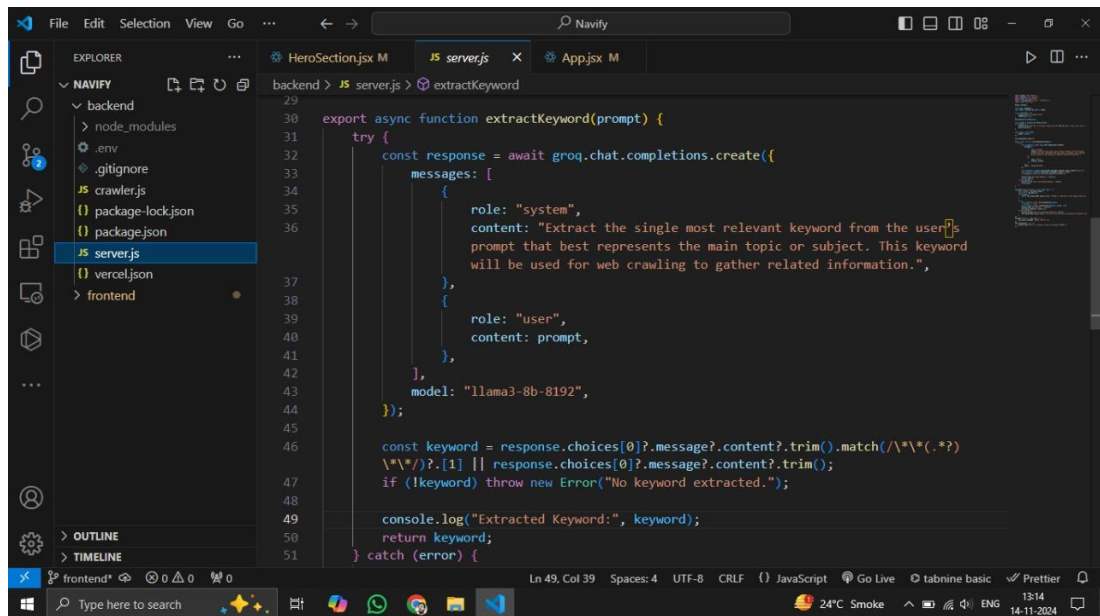
Backend



```
File Edit Selection View Go ... Navify
HeroSection.jsx M JS crawler.js X App.jsx M
backend > JS crawler.js > startCrawler > crawl
7 export async function startCrawler(keyword, prompt, url) {
14   const crawl = async (url, depth = 1) => {
15     if (depth > 2 || visitedUrls.has(url) || results.length >= 5) {
16       return;
17     }
18
19     visitedUrls.add(url);
20     console.log(`Crawling depth ${depth}: ${url}`);
21
22     try {
23       await page.goto(url);
24       const title = await page.title();
25       console.log(`Title of ${url}: ${title}`);
26
27       // First, analyze the current page
28       await crawlEachLink(url, keyword, prompt);
29
30       // Then get all links for next level
31       const links = await page.$$eval('a', anchors =>
32         anchors.map(anchor => anchor.href)
33         .filter(href => href.startsWith('http')) // Only keep valid external links
34       );
35
36       // Recursively crawl each link at the next depth
37       for (const link of links) {
38         await crawl(link, depth + 1);
39       }
40     } catch (error) {
41       console.error(`Error crawling ${url}: ${error}`);
42     }
43   };
44
45   crawl(url, 1);
46 }
47
48 export { startCrawler };
49
50 // Test the crawler
51 if (require.main === module) {
52   startCrawler('AI', 'Find the best AI tools', 'https://www.google.com/');
53 }
```



```
File Edit Selection View Go ... Navify
HeroSection.jsx M JS crawler.js X App.jsx M
backend > JS crawler.js > startCrawler > crawl
78   const analyzeContent = async (content, keyword, prompt) => {
79     try {
80       const groq = new Groq({
81         apiKey: process.env.GROQ_API_KEY,
82       });
83
84       const response = await groq.chat.completions.create({
85         messages: [
86           {
87             role: "system",
88             content: `You must respond with valid JSON only, in this exact format:
89             {
90               "score": <number between 0-10>,
91               "relevantContent": "<most relevant text excerpt>"
92             }
93           },
94           {
95             role: "user",
96             content: `Prompt: ${prompt}\nContent: ${content}\nKeyword: ${keyword}`
97           },
98         ],
99         model: "llama3-8b-8192",
100       });
101
102       let result;
```

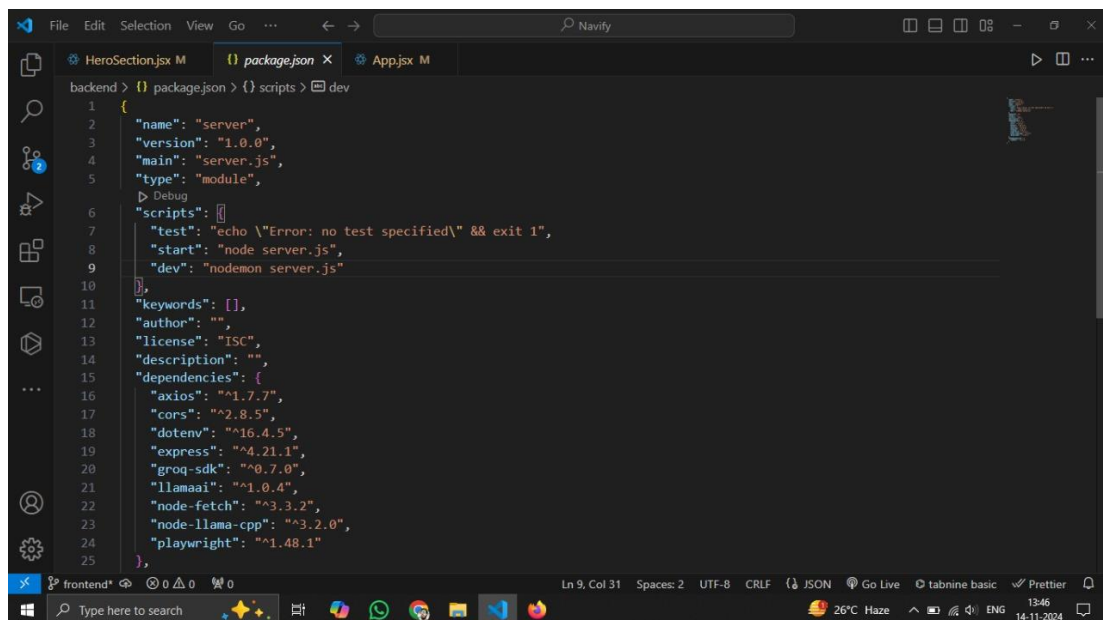


```
29
30 export async function extractKeyword(prompt) {
31   try {
32     const response = await groq.chat.completions.create({
33       messages: [
34         {
35           role: "system",
36           content: "Extract the single most relevant keyword from the user's
prompt that best represents the main topic or subject. This keyword
will be used for web crawling to gather related information.",
37         },
38         {
39           role: "user",
40           content: prompt,
41         },
42       ],
43       model: "llama3-8b-8192",
44     });
45
46     const keyword = response.choices[0]?.message?.content?.trim().match(/\"(.*?)\"/)[1] || response.choices[0]?.message?.content?.trim();
47     if (!keyword) throw new Error("No keyword extracted.");
48
49     console.log("Extracted Keyword:", keyword);
50     return keyword;
51   } catch (error) {
```

Figure 5. Backend Code

5.3 Software Installation

The installation process involves setting up the environment required to run the monolithic application effectively. First, all necessary dependencies, including Node.js, Llama, and any frontend frameworks, are installed and configured. Database configuration follows, which may involve setting up a local or remote database depending on the project's requirements. After setting up the backend, frontend, and database, the system is deployed on a server or cloud platform to ensure it is accessible to users. Documentation is provided to outline the installation process and configuration steps, helping developers replicate the environment easily. Once deployed, the application is monitored for any issues, and further adjustments are made as needed.



```
1 {
2   "name": "server",
3   "version": "1.0.0",
4   "main": "server.js",
5   "type": "module",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1",
8     "start": "node server.js",
9     "dev": "nodemon server.js"
10  },
11  "keywords": [],
12  "author": "",
13  "license": "ISC",
14  "description": "",
15  "dependencies": {
16    "axios": "^1.7.7",
17    "cors": "^2.8.5",
18    "dotenv": "^16.4.5",
19    "express": "^4.21.1",
20    "groq-sdk": "^0.7.0",
21    "llamaai": "^1.0.4",
22    "node-fetch": "^3.3.2",
23    "node-llama-cpp": "^3.2.0",
24    "playwright": "^1.48.1"
25  },
26 }
```

Figure 6. System Installation Code

CONCLUSION AND FUTURE SCOPE

Conclusion

The Navify project tackles a significant challenge in web user experience: poor navigation and ineffective information retrieval. These issues contribute to high bounce rates, with studies showing that 95% of users do not return to websites due to difficulty finding relevant information or a lack of engagement. Navify addresses this problem by introducing an intelligent chatbot powered by the Llama natural language processing (NLP) model.

The system allows users to interact with websites using plain language prompts, eliminating the need for complex navigation or extensive searches. Through its advanced features, such as keyword extraction, content crawling, and sentiment analysis, Navify retrieves the most relevant pages from a website. It ranks these results based on relevance and sentiment, presenting users with the top three results almost instantly. This streamlined process significantly reduces user frustration and enhances the overall browsing experience.

By improving accessibility, Navify fosters user retention and satisfaction, creating a seamless interaction between users and web content. Furthermore, its potential for scalability, integration with advanced NLP models, and adaptability to various platforms ensures its relevance in an evolving digital landscape. The project demonstrates how AI-driven tools can transform web navigation, offering an engaging, user-friendly, and efficient solution to a longstanding issue in online experiences.

Future Scope

As the web continues to evolve, it is becoming increasingly critical to ensure that users can easily navigate and interact with websites, regardless of their complexity. The Navify system seeks to revolutionize how users browse the web by utilizing natural language processing (NLP) and AI to deliver an intuitive, conversational interface. This innovative system allows users to interact with websites through conversational prompts, rather than relying solely on traditional search methods or navigation structures.

Transition to Microservices: The current monolithic architecture of Navify serves its purpose in handling user interactions and API calls, but as the number of users and websites grows, it will be essential to consider transitioning to a microservices architecture. This shift will improve scalability by allowing different services, such as keyword extraction, response generation, and AI training, to scale independently. This is especially important as the user base expands and traffic spikes occur.

Integrating Advanced NLP Models: The effectiveness of Navify's keyword extraction and sentiment analysis heavily relies on NLP models. To further enhance the accuracy and understanding of user queries, future iterations of Navify could incorporate advanced NLP models or integrate Llama with other cutting-edge algorithms. This would ensure better handling of complex queries and a more accurate interpretation of user intent.

Machine Learning for Personalization: Another area of future development is the integration of machine learning (ML) algorithms to personalize user experiences based on their browsing history and interaction patterns. Over time, Navify could learn from each user's preferences and

behavior, offering more tailored and relevant suggestions, thus boosting engagement and retention.

Multilingual Support and UI/UX Improvements: Expanding multilingual support will be crucial for broader global adoption. By making the system adaptable to multiple languages, Navify could cater to a more diverse audience. Additionally, UI/UX improvements could be focused on enhancing user interaction, ensuring an engaging and seamless experience across devices.

Compatibility with Other Platforms: To further broaden the scope of Navify, future versions could aim for greater compatibility with various content management systems (CMS) or other web platforms. This would make it more adaptable, allowing integration with a wider range of websites, ultimately fostering its use across different industries and domains.

APPENDIX

A. Code Structure

Frontend:

PromptBox.js: Contains the main chatbot interface and handles UI interactions, such as sending messages and displaying bot responses.

NavifyIcon.png: The icon representing Navify within the web extension.

Dependencies: Uses React for UI components, Axios for HTTP requests, and Lucide-react for iconography.

Backend:

Server Setup: Built on Node.js with Express for handling API requests and responses.

navifyAPI.js: Backend endpoint (/api/v1/navify) processes user prompts, fetching relevant pages based on keywords, site structure, and context.

NLP Module: A deep learning model for natural language processing helps interpret user queries and dynamically adapt responses.

B. Technical Specifications

Frontend: React, TailwindCSS, Axios

Backend: Node.js, Express

NLP Model: Python-based, deployed via REST API

Deployment: Localhost for testing, scalable to cloud hosting

C. Sample API Response Format

Request:

```
{ "prompt": "Show login page", "url": "https://example.com/home" }
```

Response:

```
{  
  
  "content": "Here is the login page you requested.",  
  
  "links": [  
  
    { "label": "Login Page", "url": "https://example.com/login" }  
  
  ]  
}
```

REFERENCES

Russell, S., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach. Pearson.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

React Documentation. (2024). Retrieved from <https://reactjs.org/docs/getting-started.html>

Axios Documentation. (2024). Retrieved from <https://axios-http.com/docs/intro>

Node.js Documentation. (2024). Retrieved from <https://nodejs.org/en/docs>

Playwright. <https://playwright.dev/docs/intro>