Adaptive Speed Reading

Matis Bodaghi, Evangelos Georgiadis, Jack Hau,
Kyoya Higashino, Konstatinos Mitsides, Fadi Zahar
{mrb23, eg923, jhh23, kh123, km2120, fz221}@imperial.ac.uk

Supervisor: Dr Konstantinos Gkoutzis

COMP70079
Department of Computing
Imperial College London

April 29th, 2024

1 Introduction

In today's digital information age, the ability to assimilate vast amounts of textual information rapidly and effectively is not just an advantage—it is a necessity. As professionals across various fields are constantly inundated with textual data, enhancing reading skills could dramatically increase their efficiency. While existing techniques primarily focus on increasing reading speed, they often compromise the depth of understanding, leading to superficial knowledge [1]. Moreover, while current speed-reading platforms may provide adaptive tools that help users read more quickly, they utilise text-centric adaptivity and create a dependency without necessarily training users to apply these skills in real-life situations. In this project, we focus on building a speed-reading trainer based on user-centric adaptivity, adapting to the reader instead of the text to train users to effectively read faster under any conditions. By consistently monitoring both the reading speeds and comprehension levels of users, we aim to bridge the gap between reading speed and depth of understanding.

1.1 Background and Related Work

Speed reading is a technique that allows people to read and process written material at a much faster pace than normal reading rates, with one of the most well-known techniques being Rapid Serial Visual Presentation (RSVP) [2]. RSVP is a method that displays words sequentially at a fixed location on the screen, eliminating the need for eye movements and the common habit of regression, which refers to the unnecessary re-reading of previous lines or words. This technique aims to boost reading speeds by reducing visual adjustments and allowing for a more focused absorption of content by presenting text at controlled speeds. While earlier research indicated that RSVP could increase reading speed by 33% for short texts without affecting comprehension [3], more recent studies, such as one examining the RSVP app Spritz [4], have revealed drawbacks. Although conventional RSVP effectively minimises eye movements by displaying words sequentially, it has been found to impair literal comprehension and increase visual fatigue due to the constant flashing of words discouraging users from blinking.

Chunking, on the other hand, is a technique that prioritises enhancing comprehension by grouping words into larger, meaningful units. This method improves information processing by efficiently utilising working memory capacity—the limited amount of information a person can hold in active memory at one time. By helping the brain recognise patterns and connect concepts more rapidly, chunking effectively boosts reading understanding [5, 6].

Turning our focus to practical applications of these techniques, platforms like Spreeder [7] and SwiftRead [8] have gained popularity. They both employ the RSVP technique to help users read faster while offering tools to customise the reading experience, such as adjusting the word-per-minute rate and the chunk size, which can be tailored to suit individual preferences and reading abilities. Although these platforms provide various pre-reading customisation options and can adapt to the characteristics of the text, they do not adapt to users themselves. This is because they lack mechanisms to continuously monitor and adjust to users' reading patterns in real-time. Moreover, these platforms operate in ways that make users reliant on their specific tools to achieve faster reading speeds, focusing on assisting users rather than training them.

1.2 Project Overview

Through our work, we aim to address the limitations identified in current speed-reading research and related platforms by focusing on two main objectives: to enable user-centric adaptivity of speed-reading tools and to serve as a training platform. The key innovation in our approach stems from integrating WebGazer's eye-tracking technology [9], which tracks users' eye movements to dynamically adjust the reading speed based on real-time observation of individual reading patterns. To mitigate RSVP's comprehension limitations while simultaneously adhering to our objective of acting as a trainer, we combine traditional chunking techniques with an adapted RSVP format. This modified method presents the text in manageable chunks—typically around 8 words—facilitated through an RSVP delivery, helping manage cognitive load more effectively. Moreover, this format encourages natural left-to-right reading, mimicking real-life reading and reducing the visual fatigue often associ-

ated with conventional RSVP systems. To ensure that these innovations accelerate reading without sacrificing comprehension, we continually monitor users' reading speeds and comprehension through quizzes linked to the text.

2 Project Design and Architecture

This section outlines the project administration, the technical direction that addresses the problems, and the solutions that achieve the objectives.

2.1 Functionality Specification

The developed platform, called Kiraka (`https://srp.doc.ic.ac.uk/`), has various functionalities that pinpoint each element of the project objectives. These are:

- **Mode 1 (DocMode):** DocMode provides a more familiar and traditional approach to speed reading. Shown in the DocMode page of Figure 1, this mode serves as a baseline/standard trainer and is useful for users new to speed reading, allowing users to read the full text at their own pace with optional supplementary features. The first feature is a pointer guide that highlights words at a rate which can be manually adjusted, pushing users to follow a sustained reading pace. The second feature is HyperBold reading style, which bolds the first parts of the words to make the text easier to process. While HyperBold may not always be available in real-life scenarios, it makes the platform more accessible, especially to those who have ADD [10]. Furthermore, the reading process is timed (can be restarted and paused) and is used to infer a user-specific average number of words per minute (WPM). Additional styling features such as dark mode add to the personalised experience.
- **Mode 2 (FlashMode):** Shown in the FlashMode section of Figure 1, this is a trainer and it is Kiraka's new approach to speed reading using innovative AI-powered features, building upon popular methods of traditional speed reading such as chunking and modified RSVP techniques. The text is divided into chunks, appearing at a rate that matches the WPM.
  - **Mode 2.1 (FlashMode − Static):** In this mode, users manually adjust the pace at which text chunks are presented. This control allows users to train their speed reading skills effectively, providing a hands-on approach to learning without automated adjustments. This also accounts for users who do not have camera access or do not wish to show their faces, which is necessary for Mode 2.2 discussed below. While static mode does not incorporate AI features, it differs from DocMode as it does not allow for re-reading, eliminating regression.
  - **Mode 2.2 (FlashMode − Adaptive):** This mode addresses the project objective to be adaptive. This mode utilises eye-tracking technology through the WebGazer library to adapt the reading speed based on the user's eye movement patterns—their gaze position over time. This adaptive speed control aims to dynamically determine the optimal reading speed based on user gaze data and adjusts the WPM to appropriately push the user. Furthermore, each chunk is analysed with AI to adjust the speed based on its lexical complexity, slowing when the text is difficult to read, and vice-versa.
- **AI Quiz Generation:** After reading, the platform utilises AI to generate quizzes, designed to test comprehension immediately and ensure that speed enhancement does not come at the expense of retention or understanding. This feature is crucial as it addresses the speed-comprehension trade-off, a common issue reported when using traditional speed reading approaches. The use of AI to generate quizzes is instrumental as it permits users to upload their preferred texts, reinforcing the platform's user-centric focus.
- **Users Analytics:** An analytics page is provided to inform users of their reading and comprehension metrics. Presented through a series of graphs, users can reflect on their progress and motivate themselves to keep practising on the Kiraka platform.

2.2 Software Engineering Design

The project team was organised into specialised roles with the project leader in charge of the overall project trajectory and alignment with the goals. The Back End Team managed server-side functionalities and database operations, while the Front End Team focused on user interface and user experience design including the reading modes. The Artificial Intelligence (AI) Team was responsible for integrating Large Language Models (LLMs) to generate dynamic content. To enhance flexibility and responsiveness within the project framework, Agile methodologies, particularly Scrum [11], were adopted. This approach included regular sprint planning sessions, daily stand-ups, and sprint reviews, which facilitated dynamic adjustment and incremental development. Notion [12] served as our central tool for note-taking, sprint planning (Appendix C), and task management, while Microsoft Teams [13] was indispensable for communication and meeting coordination among team members.

2.2.1 Front End Development

The front end of modern web applications plays a crucial role in delivering a convenient experience for users. With this in mind, the front-end development of Kiraka is built using Next.js 14 [14], a popular React framework, which offers a robust set of features for building efficient and scalable web applications. The Next.js front end is responsible for rendering user interfaces and handling client-side interactions. One of the core reasons why Next.js is chosen over its counterparts, like React and Vue, is its built-in routing system, which simplifies page navigation within the Kiraka web app. With the Next.js app router system, the routes are defined as a file structure inside the `\app` folder that maps to specific pages or components.

Thorough design planning has gone into Kiraka's front-end architecture to enhance user experience. The layout is organised to ensure content is presented logically and aesthetically, facilitating easy navigation through consistent menu placements and user interface elements such as the sidebar. Elements like quiz and calibration buttons are conditionally displayed based on the user's interactions and context to help keep the interface focused and uncluttered. For example, a quiz is only accessible after reading the full text and calibration can only be attempted after WebGazer is activated. To assist the users, pop-up boxes and tooltips are shown. These elements are crucial in enhancing the learning experience by providing timely assistance. The user interface is highly adaptive, responding to various user actions and preferences with real-time feedback. This includes customisable settings such as text size and colour, improving reading conditions and overall accessibility. An appropriate colour scheme is selected to provide the users with comfortable visualisation. Green, a low wavelength colour that promotes restfulness and concentration [15], is chosen as the dominant colour in Kiraka, improving efficiency and focus. To minimise eye strain, Kiraka employs an off-white/yellow background for all reading interfaces. This easy-on-the-eye colour has shown improved reading performance [16] and is also considered dyslexia-friendly, further supporting our commitment to diversity [17].

To optimise performance and enhance user experience, various strategies were employed. These include separating server-side and client-side components, server-side rendering for content-heavy pages, client-side rendering for interactive elements, utilising React's Context API for efficient state management, and leveraging client-side storage APIs (Local Storage and Session Storage) for caching data, persisting user preferences, and maintaining state across page transitions. This comprehensive approach aims to reduce load times, improve rendering performance, and optimise bundle sizes, ultimately providing a smooth and responsive user experience. Before releasing the app to users, an optimised version is generated for production by creating HTML, CSS, and JavaScript files based on the pages. Using Next.js compiler, JavaScript is compiled and browser bundles are minified to help achieve the best overall performance. The production-ready files are hosted in the Imperial College Department of Computing's virtual machine (VM). The Next.js application is running locally on the VM's port 3000. To expose the application to the internet, Nginx, a web server and reverse proxy server, is configured to serve over HTTPS with SSL encryption as an additional layer of security.

The overview of the user journey can be visualised in Figure 1. It begins at the landing page with a clear hero section and buttons on the navbar. When users sign in, Kiraka leverages Clerk [18], a third-party authentication, platform, to enable social connection with their Google, Apple,

Facebook, or Microsoft accounts. These OAuth providers allow secure authorisation to access users' resources without sharing their credentials and going through complex registration flows. Users can also use normal email addresses with passwords to create their accounts. Clerk's authentication flow simplifies the sign-up and sign-in process, resulting in a higher conversion rate and straightforward experience. Kiraka thus offloads complex authentication logic, which reduces the risk of potential security vulnerabilities that could arise from implementing custom authentication solutions. After authenticated users are presented with platform instructions, they are then directed to our main functionality, FlashMode Adaptive. Users can also navigate to other reading modes, where users can then select the pre-existing texts or upload their own. WebGazer for FlashMode (Adaptive) is activated by navigating via the calibration page. After a text is read, a quiz option pops up to prompt users to validate their comprehension. At any time, users can go to the analytics page to monitor their previous reading speed and quiz scores.

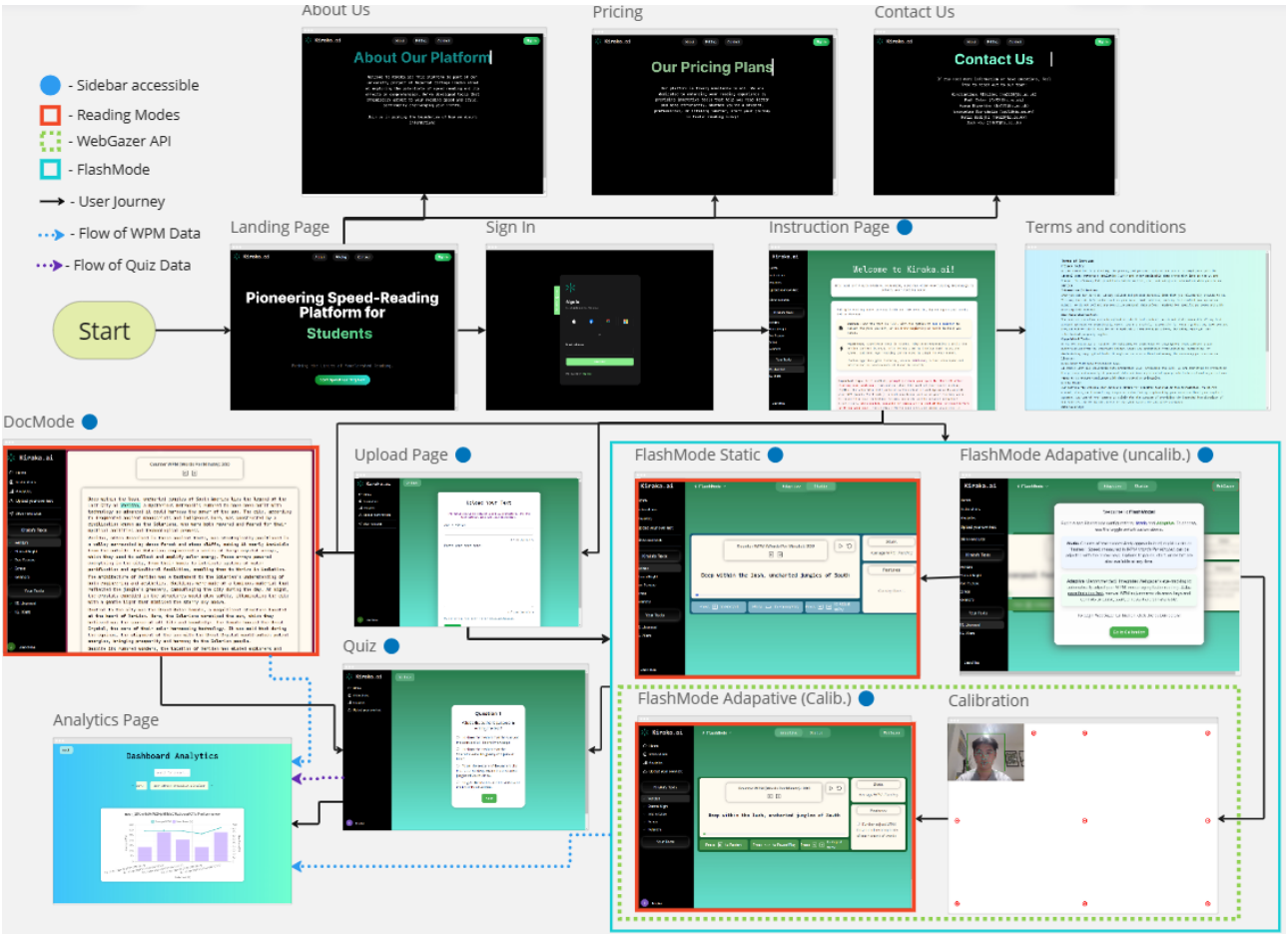


Figure 1: Snapshot of User Journey and Website Architecture

2.2.2 Back End and Database

The back end consists of two layers: a Flask application and a MariaDB database. Flask is a lightweight Python framework used to handle the back end of web applications. It acts as the middleman between the user and the database, handling HTTP requests and routing URLs to functions that perform queries or modifications to the database and perform inference on the LLMs. The app deployed by Flask runs a Python script in which API routes, as well as their functionality, are programmed.

Flask is highly configurable and supports extensions, Flask-CORS and SQLAlchemy, that can achieve a robust level of security, safeguarding against various web application vulnerabilities effectively. Flask-CORS is an extension for handling Cross-Origin Resource Sharing, which allows or restricts resource sharing across different origins. Proper configuration of CORS is essential for preventing unwanted cross-domain interactions and protecting against Cross-Site Request Forgery attacks

and data theft. SQLAlchemy allows the abstraction of SQL commands through Python classes, automatically protecting against SQL injection, which is one of the most exploited security vulnerabilities.

MariaDB is a Relational Database Management System designed for efficient storage and querying of data. All user data are stored in a single database, organised to the second normal form with constraints ensuring each piece of data is uniquely identifiable. It consists of many tables to store the users and their relevant data (see Figure 5 in Appendix B). The database does not store any of the user's sensitive information related to login activity. Instead, the back end uses only the unique identifier, created by Clerk for each user, to be stored in the Users table as a primary key. An admin column stores a boolean value which determines whether the user is an administrator of the website. Admin can view all the users' test scores and reading speed via the unique identifier. Texts are stored in a second table, and their corresponding quiz questions in a third. Upon upload, the text is broken into chunks and the complexity of each chunk is calculated (see Section 3.2.2). This design choice was made so that the chunks and complexity—used to further adjust reading speed—are not calculated on the go when a user starts reading a text in FlashMode. Finally, data from the users' practice sessions are stored. Each practice session row contains two foreign keys: the user's primary key in Users and the text's primary key in Texts. The practice session's key is referenced as a foreign key in several tables which store the more granular data about a user's practice session, like their quiz results, their eye movements (gaze data over time) and their reading speed throughout the text when using WebGazer.

The Flask app runs inside Waitress, a Web Server Gateway Interface (WSGI), which improves its scalability, reliability and robustness, making it suitable for production environments. The API functions allow the front end to fetch user information, retrieve texts and their relevant content, send texts for storage in the database, and store the results of users' practice sessions.

2.3 DevOps (Development & Operations)

To ensure efficient collaboration and code integrity, Git is employed as the version control system, and the project codebase (`https://gitlab.doc.ic.ac.uk/g237007906/kiraka`) is hosted on GitLab. The team followed a Git-Flow branching strategy to maintain work across the teams. This defines specific branch responsibilities, such as "master" for production, "dev" for active development, and feature branches (like "FlashMode") for new features. Continuous Integration and Continuous Deployment (CI/CD) are implemented to streamline the development-to-production process and to introduce automated testing and deployment. GitLab-Runner is configured on the VM to execute the CI/CD pipeline to automate the test, build, and deployment workflows. When a feature branch is merged to the dev branch, only the testing and building are triggered, whereas the entire pipeline is activated when the dev is merged to the master branch. The CI/CD pipeline begins with code linting using ESLint to enforce strict code quality standards and catch potential issues early. Then, comprehensive unit tests are implemented. The testing pipeline is exclusively composed of unit tests of the API routes, to check if the expected and actual HTTP responses match. In addition to standard requests, the test suite also checks the correct handling of requests attempting to tamper with another user's data, and of some edge cases where data is missing in the database or where the request format is incorrect. The application is built and packaged for deployment upon successful linting and testing. The packaged application is then deployed to the production environment hosted in the VM.

3 Technological and AI Integration

3.1 Webgazer Application and Optimisation

3.1.1 WPM Calculation and Display Mechanics in FlashMode

As highlighted in Section 2.2, FlashMode functions as a speed-reading trainer successively displaying chunks of text in short, rapid "flashes". These chunks have been standardised to a maximum of 50 characters, roughly translating to 10 words assuming an average of 5 characters per word. In instances of exceptionally long words, these are hyphenated and continued in the subsequent chunk(s). By fitting

these chunks over most of the available screen width, FlashMode effectively displays proper sentences that mirror the length of actual text lines, with an increased font to help enhance readability by reducing eye strain and improving visibility. Each user will encounter identical text chunks for any given text, precisely centred on the screen and scaled to their display through font size adjustments. This format maintains the designated space for the sidebar and command sections and aligns with the webcam for FlashMode Adaptive (refer to Figure 2). We utilise the monospaced 'JetBrains Mono' sans-serif font in displaying the text chunks. 'Sans-serif' fonts are characterised by cleaner, simpler lines without decorative strokes at the ends, making them less visually complex and slightly more readable. 'Monospaced' refers to the font having a fixed character width, which in this case is 60% of the font size. This allows for precise calculation of the necessary text size to ensure that the chunks fit within the display window, whose width varies with different screen dimensions. This size is determined "client-side", where font adjustments are made dynamically using the equation: $\text{fontsize} = \left(\frac{\text{width}}{\text{max\_char\_per\_chunk}}\right) \times 0.6$.

At the core of FlashMode lies the timing of text chunk displays, which are directly related to the set WPM rate. Assuming an average of five characters per word [19], the display time for each chunk, in seconds, can be calculated by $\frac{\text{chunk\_length} \times 60}{\text{WPM}} \times 5$, where chunk_length is the character length of the chunk.

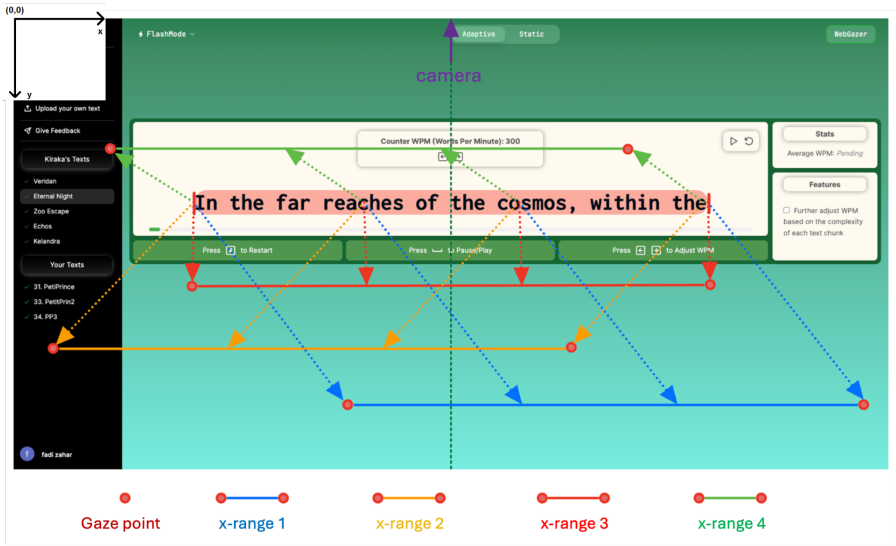


Figure 2: Snapshot of FlashMode Adaptive illustrating the variability in x-ranges, highlighting that these values are not absolute.

3.1.2 WebGazer: Usage and Challenges

Before utilising WebGazer's eye-tracking capabilities, we begin with a calibration phase to activate the library. Calibration is essential to ensure the accuracy of gaze data. Here, we store and monitor the activity status of WebGazer using a React context (useContext) to check if it is active across different components. After the initial calibration, our application begins to listen for gaze data at a sampling rate near 30Hz (updates every 33 milliseconds), a process continuously running in the background. We achieve this through a React useEffect hook that listens and responds when the eye-tracking feature is active, as long as the user is not pausing their session, and as they navigate through different parts of the text. These are the 'dependencies' the hook monitors. Whenever one of these dependencies changes, the useEffect adjusts, ensuring the application always responds with the most current gaze data to adapt the reading experience in real-time. This gaze data, which consists of the gaze's x and y coordinates (in pixels, relative to the viewport) and elapsed time since WebGazer was initiated, enables real-time analysis of the user's reading patterns. It informs adjustments to the WPM which are applied only after the end of the current chunk from which the data is collected during FlashMode, effectively assigning a stable WPM to each chunk.

Building on the WebGazer setup, the main goal is to have the user's pacing with the text at predetermined WPM rates actively monitored. Adjustments to the reading speed for subsequent chunks are accurately determined, either increased or decreased, depending on whether it is found that users are lagging or advancing. However, early on in our experiments, we observed that WebGazer's gaze tracker exhibited considerable inaccuracy and jitteriness. This was particularly noticeable in the y-direction, likely because the iris has less vertical movement, limiting the precision of vertical displacement tracking. The accuracy was somewhat better in the x-direction, especially after updating the gaze-tracking parameters, switching the regression model from "ridge" to "weightedRidge" and maintaining the Kalman filter enabled for smoother predictions. This horizontal precision was advantageous in our FlashMode, where large text chunks cover much of the screen width and the vertical direction is less critical.

3.1.3 Algorithm Development and Milestones

In our first attempt for adaptive speed reading based on eye-tracking, we divided the user's screen into five sections: three main areas within the text display window and two 'red' zones flanking either side. The total display time of T seconds for each chunk was equally distributed among the three main regions, each lasting T/3 seconds. These regions served as references, indicating where the user's gaze should ideally be if their reading pace aligns with the set WPM. This tripartite division was strategically chosen to minimise the impact of gaze data inaccuracies. Using this 'moving reference' system, we started with the leftmost region as the reference for the first T/3 seconds. During this period, gaze data was collected to determine the user's reading pace: if the gaze was left of the reference (indicating lagging), five points were subtracted; if to the right (indicating leading), five points were added. The red zones operated on a higher scale, adding or subtracting ten points to rapidly indicate significant pacing discrepancies. The process repeated for each region, and by the end, a positive total average score suggested the user was ahead, while a negative implied lagging. Adjustments to the WPM for the next chunk were based on these scores, halting point collection if the user reached the far-right zone prematurely, indicating completion of the chunk (refer to Figure 7 in Appendix D). Following this initial approach, we found that the results did not satisfactorily reflect the user's actual reading pace and proved overly restrictive. The fundamental issue stemmed from the reliance on absolute gaze positions, which are inherently inaccurate due to calibration inconsistencies, variations in lighting conditions, user-specific factors like eyewear, head movements, and the jittery nature of WebGazer itself. We observed significant discrepancies in where the x-value starts and ends for a chunk— "range shifts" (refer to Figure 2 in Appendix D), with start values between 364px to 649px and end values between 1424px to 1982px (refer to Figure 11 in Appendix D)—that did not consistently align with the actual boundaries of the text display window. These variations led to inconsistencies in gaze data both across different sessions and within individual sessions, rendering this method ineffective for accurately tracking and adjusting reading speed.

Second attempt: After recognising the limitations of relying on absolute gaze positions, we shifted our focus to using relative gaze positions, extending to gaze velocities, which offered a more reliable measure, especially considering the 'range shifts' previously discussed. To refine this method, we considered linear regression to model the relationship between gaze position (x) over time, focusing primarily on the slope of this relationship, which represents the velocity of the gaze movement. This regression analysis was designed to filter out the noise and provide a stable measure of gaze velocity by fitting a line to the gaze position versus the time plot. The slope (a) represents the gaze velocity, calculated as the distance covered (d)—the chunk display width in pixels—divided by the time taken in seconds (t'), thus $a = \frac{d}{t'}$. Since display time (t) is inversely proportional to WPM $= \frac{12 \times \text{chunk\_length}}{t}$, we can derive the effective WPM from the slope: WPM $= \frac{12 \times \text{chunk\_length} \times a}{d}$. It should be noted that this approach assumes that the span of x-values is relatively stable, which is not always the case. Although user movements, such as returning their gaze to the start of a new chunk, resulted in misleading negative slopes at the data's edges, and issues like blinking introduced spikes in x-values that WebGazer struggles to handle accurately, leading to low performance, this second attempt still provided crucial insights into the velocity profiles of gaze position over time. Through experimental observations, we established a critical threshold for gaze velocity. Specifically, on a laptop with a

screen width of 1728px and a word display width of 1201px, we identified that gaze speeds slower than $-\frac{2\,\text{px}}{\text{ms}}$ consistently indicated a user's gaze returning to the left. This threshold, normalised for different screen sizes, is calculated as $-\frac{2}{1201} \times 100$ (where 100 is a scaling factor). This normalised velocity threshold has proven to be a robust indicator of the initiation of the return gaze movement to the left, allowing us to adjust our system dynamically based on user interaction and screen dimensions.

3.1.4 Adaptive WPM Adjustment: Custom Integration and Functionality

Building on the analysis of our second attempt, we further refined our approach using a "cat-and-mouse" strategy that leverages user behaviour to dynamically adjust reading speeds. By specifically monitoring the last 35% of the display time for leftward gaze movements—indicative of a user finishing reading—we avoid early negative slopes caused by transitions from previous chunks and mitigate effects such as blinking that can falsely signal a return gaze—refer to Figure 12 in Appendix D for detailed visualisation. This focused observation ensures actions are based on reliable data, with negative x velocities marking the user's completion of the reading. Detected leftward movements immediately trigger the advancement to the next chunk, with the WPM adjusted based on the actual time taken, t, according to the formula $\text{WPM} = \frac{12 \times \text{chunk\_length}}{t}$. To manage potential disruptions from blinking in the crucial last 35% of the display time, we implemented a safeguard against excessive WPM increases. Any increase between two consecutive chunks is capped at 60, and if cumulative increases exceed 90, a dampened increment of 5 WPM is applied. This adjustment mechanism ensures that the WPM remains responsive yet controlled, avoiding unintended spikes due to artifacts like blinking. Continuing from the safeguard measures, we also adjust the reading speed based on the complexity of each text chunk mentioned in Section 3.2.2. For more complex chunks (identified by a complexity score above 0.77), the final WPM adjustment is decreased, and conversely, it is increased for simpler chunks (with a complexity score below 0.7). This dynamic adjustment helps tailor the reading challenge to the content's difficulty, ensuring that the pacing is always suited to the material's demands. Additionally, the starting WPM for each reading session is informed by the average of the last ten WPM values recorded for the user in this mode, enabling a more personalised and responsive reading experience that adapts to both user performance and text complexity.

3.1.5 Limitations

While our refined approach holds promise, several inherent limitations could impact its effectiveness. User compliance is critical, as the system's accuracy depends on users returning their gaze to the left after reading—a key indicator for determining reading pace. Additionally, despite measures to reduce its impact, blinking can still compromise accuracy, especially if it occurs during the critical last segment of display time we analyse. Variabilities in calibration quality and user behaviour, such as inconsistent adherence to instructions or confusion, can also introduce erratic data that may mistakenly trigger increases in WPM. These factors underscore the complexity of reliably integrating eye-tracking in reading applications and highlight the importance of robust user training and calibration processes.

3.2 LLM-Based Implementation

3.2.1 Quiz Generation

To generate multiple-choice questions (MCQs), several open-sourced LLMs were explored, considering the base architecture, abstractive vs. extractive generation, and the use of multiple models. Ultimately, two unique strategies were developed for comparison: a single extractive model that uses cosine similarity to generate false options, and an abstractive model that uses another LLM to generate false options. While both methods utilise an LLM to generate question-answer (QA) pairs, differences stem from the nature of the questions themselves, as well as how false options are generated.

Extractive Pipeline: For extractive quiz generation, two fine-tuned models were analysed: T5-large [20] and BART-large [21]. Both models are trained on the Stanford Question Answering Dataset (SQuAD) [22], a crowdsourced set of QA pairs derived from Wikipedia articles. Through qualitative

testing on hand-crafted texts of 500-1000 words, the T5 model was observed to be more reliable, making fewer mistakes while producing more varied questions. This is likely due to its larger size, having 770 million parameters versus 406 million. Despite its size, inference speeds for both models were comparable and thus T5 was chosen.

However, several issues had to be addressed, including the repeated generation of similar questions and model's tokeniser having a maximum limit of 512 tokens. To solve this, texts are split into five sections, ensuring question diversity with each section generating a QA pair. Furthermore, using a single LLM meant only QA pairs could be extracted. Hence, the cosine similarity between word embeddings was used to find potential false options for an MCQ format. While the WordNet vocabulary is used, embeddings are derived from the BERT-based-uncased model [23]. Following this procedure, numerous exceptions needed to be handled to generate false options properly up to the trigram level, including dates, numbers, determiners, repeated root words, and more. A full description of our rules to handle these exceptions can be found in Appendix E Table 3.

Abstractive Pipeline: The abstractive approach utilised two LLMs: one for QA generation and another for false options. Trained on the ReAding Comprehension dataset from Examinations (RACE) [24] for English language learners, these T5-large-based models were developed as a set specifically for MCQ generation. While QA pairs are generated similarly, the false options are found by considering each QA pair as well as the entire text. In this way, the nature of questions is more representative of general understanding, testing overarching concepts instead of extracted facts. Ultimately, the abstractive pipeline was chosen for its higher accuracy and quality, developing QA pairs at a reduced error rate while testing overall comprehension. A full analysis can be found in Appendix E Table 4.

While capable of producing high-quality MCQs, the chosen pipeline still has a high error rate of approximately 50%, having issues such as irrelevant questions and multiple correct answers. Although the quality of the AI-generated quizzes may not as high as human-generated ones, this approach offers flexibility by allowing users to upload a text of their choice while effectively providing an indicator of the user's level of comprehension. Furthermore, platform-provided texts can easily utilise this pipeline, only requiring human-guided approval as the final step. This is the strategy used for defining platform-provided reading samples in the final user trials.

3.2.2 Chunk Complexity

To improve FlashMode Adaptive, the complexity of chunks is considered, reducing reading speed if complexity is high and vice versa. Complexity scores are calculated at the sum of two sub-scores: an LLM-derived term C that considers contextual meaning, and a deterministic term that considers each chunk's longest word LW. While both sub-scores are imperfect, their combined score provides a holistic and balanced evaluation while reducing the impact of random error. Although both are bounded within the range of 0 to 1, empirical analysis shows that the total score distribution is heavily centred in the range of 0.60 to 0.75. As such, FlashMode Adaptive focuses on scores within this range, as mentioned in Section 3.1.4. The final equation for Chunk Complexity is defined as:

$$Chunk\,Complexity = C + 0.02(LW) \tag{1}$$

Contextual Complexity: Our fine-tuned complexity LLM is derived from public submissions to Task 1 of the SemEval-2021 competition [25], predicting the lexical complexity of target phrases that are encompassed within a context sentence. While multiple submissions were analysed, this project's training pipeline builds upon a specific repository from Team CS60075-Team-2-Task-1 [26] for its replicable results within our resource constraints. After fine-tuning on the competition dataset, an optimal BERT-based model was developed, capable of scoring complexity from 0 to 1. On the pre-segregated test set (in-domain testing), the model performed well, having an MSE error of 0.00927.

To adapt the model to evaluating specific chunks, different sections of the text were considered as the context and target phrases. These methods are shown in Table 1. Note that Method 2 predicts complexity scores for both halves of the current chunk before using an average to get a final prediction.

Method	Context	Target
1	Current chunk	Current chunk
2	Current chunk	Half of current chunk
3	Previous, current and subsequent chunks	Current chunk

Table 1: Methods for contextual complexity inference

As this adaptation is specific to our downstream task, evaluation was done qualitatively by testing each method on handcrafted texts, ensuring more complex chunks were assigned higher scores and vice versa. Ultimately, all three methods produced similar results, having moderate accuracy with some mistakes. As such, Method 1 was chosen for its simplicity and faster inference speeds.

Longest-Word Complexity: However, this sub-optimal performance led to the incorporation of the second sub-score, based on the longest word (LW) in each chunk, to act as a simple and deterministic way to calculate complexity. This is based on the reasoning that longer words are more complicated and therefore require more time to read. While the average word length was initially considered, it was qualitatively observed that it was usually the case of a single long word that caused complexity. Therefore, the longest word was considered directly.

Initially, a modified sigmoid function was explored to bind the range between 0 and 1. However, its non-linear properties resulted in minor differences between very long words and large differences in medium-length words of lesser importance. Hence, a linear function through the origin was proposed to treat all marginal differences in word length equally. Ultimately, a linear coefficient of 0.02 was chosen to give a maximum score of 1 for 50-character words (maximum size of the chunk window).

3.2.3 Summary Generation

Before fine-tuning an LLM, several pre-trained encoder-decoder models were evaluated whose base architectures included Longformer Encoder-Decoder (LED) [27], BigBirdPegasus [28], and Long T5 [29]. As reading fiction is usually done for leisure, it is unlikely that users would like to learn how to speed read using these texts. Instead, it is apt to assume that non-fiction and information-heavy texts are more relevant for speed reading. As such, the fine-tuning process included scientific papers from arXiv as the training corpus, representative of the most challenging and complex texts that users could upload. However, the Long T5 model did not have a publicly available arVix-trained variant, so an eLife-trained variant was used.

Each model was tested on articles that were less than 1000 words, with model variants having a maximum context window of 16384 tokens, which was easily achievable. Using metrics like BLEU and ROGUE, it was determined that the LED model is superior, outperforming the other models in all metrics (Appendix E Figure 13). However, it was also observed that generated summaries often contained irrelevant information, being far from an ideal summarisation model for our platform.

Given that the LED model uses only 41 million parameters, the Llama-7b base model [30] was chosen for fine-tuning experimentation due to its significantly larger number of parameters (7 billion), and its popularity in summarisation tasks. For fine-tuning, an Nvidia Tesla A30 GPU was used, being the only GPU consistently provided by Imperial's GPU clusters. While it holds 24 GB of RAM, it is divided into two 12 GB instances, meaning that only 12 GB could be utilised for fine-tuning.

Without using any Parameter-Efficient Fine-Tuning (PEFT) methods, and by using 32-bit floating-point precision, this setup would require 28 GB for inference. Additionally, during the training stage, the memory requirements for model parameters, estimating gradients, the optimiser, and activations for backward propagation—assuming a batch size of one—would amount to approximately 28 + 28 + 56 + 18 = 130 GB. To reduce these size requirements and proceed with the process, we utilised the Quantised Low-Rank Adaptation (QLoRA) technique. Specifically, both 4-bit quantisation and double quantisation were employed, thereby reducing the model weights from the typical 32-bit floating-point precision to 4-bit. Consequently, the memory required for storing the model's parameters was reduced by a factor of 8, leading to a required size of roughly 4 GB for inference. Moreover, for LoRA, we

used a matrix rank of 2 for the updated matrices. In our case, this approach entails updating only 5 million parameters (0.14%) during training, a significant reduction from the initially targeted 3.5 billion parameters, thereby reducing the required memory size during fine-tuning and allowing the process to run smoothly.

Despite making the fine-tuning process more time- and memory-efficient, the performance improvements for our summarisation task were minimal. Although the model was trained for 7 hours on a dataset of 4038 1000-word texts for 2 epochs, the performance was nearly identical to those of the pre-trained model. This modest progress was anticipated since we deliberately used low parameters, including a matrix rank of 2 and a batch size of 1, to meet our computational limits, which limited the effectiveness of the fine-tuning.

After three weeks of experimenting with smaller-sized pre-trained LLMs like T5-3b to balance efficiency and effectiveness in fine-tuning, we decided to exclude text summarisation from our website due to underwhelming results. Although a desirable feature, further investment in this area seemed unjustifiable given its limited potential value. Users can achieve high-quality summaries using other LLM-based platforms and then import them to our site, maintaining the site's primary functionality without compromising its objectives.

4 User Trials

During the development process, user trials were conducted to gather real feedback and iteratively improve the platform. Each trial tested a progressively more advanced version of the website. A brief description of the website version and the trial's specific objectives can be found in Table 2.

Trial	Platform Features	Specific Objectives
1	- DocMode (Pointer Highlighting, HyperBold) - Static FlashMode - No Quizzes	- Act as a non-AI version baseline - Gather user insights on Doc/FlashMode - DocMode vs FlashMode reading speed
2	- FlashMode (Adaptive) - Human-Generated Quizzes - User/Admin Analytics Page	- Gather user insights to FlashMode - Analyse if FlashMode is challenging users - Observe speed vs comprehension trade-off
3	- DocMode (Toggle Options) - Dynamic FlashMode (Chunk Complexity) - AI-Generated Quizzes (Human Approved) - Self-Upload Text (beta)	- Gather insights for Chunk Complexity - Observe new reading speed variance - Gather insights on AI-generated quizzes - Gather insights on Self-Upload Texts

Table 2: Detail and objectives of all user trials

4.1 Ethics and Privacy

To ensure the privacy and security of user data following the seven principles of the General Data Protection Regulation (GDPR) and Data Protection Act 2018 (DPA18), multiple measures are implemented. First, a "Terms and Conditions" agreement is shown to users within the log-in and self-upload pages, detailing how their personal data will be used under GDPR's principles of lawfulness, fairness, and transparency as well as their rights to access, correct, and delete their information. Second, before the WebGazer calibration stage, a pop-up message is displayed indicating how Kiraka uses video data, assuring them that no video content is saved, only the coordinates of their gaze and timestamps.

4.2 Trial Set-Up

All trials lasted two days each, with participants reading up to five provided texts in any order, and at any time. Users were also asked to give feedback via a Google form, shown in Figure 14 in Appendix F, that covered specific features and general user experience. Participant demographics primarily include university students from various academic backgrounds as well as professionals from various

industries. However, all participants are fluent in English so the Chunk Complexity and automatic quiz generation features could be tested, being features that utilise LLMs that were fine-tuned on English datasets.

As fiction texts are normally read for leisure and therefore do not require speed reading, the provided texts were more information-focused. While public domain texts were explored, several issues were encountered. Generally, copyright law states that texts that are 70 years past the author's death are free to use, meaning available texts are mostly old novels. Searching for excerpts that aligned with modern English and an information focus proved difficult, making for sub-optimal reading examples. Instead, texts were generated via ChatGPT4 to ensure no copyright issues while allowing for modifications through careful prompt engineering. The topics of provided texts were obscure enough to provide challenging quizzes, but also simple enough to understand on a first read.

As detailed in Table 2, the method for generating MCQ quizzes changed from trial to trial, improving until a reliable pipeline was developed. In the first trial, no quizzes were provided. In the second trial, ChatGPT4-generated quizzes were manually edited to ensure accuracy and adequate complexity, also acting as the gold standard for quiz generation. In the final trial, Kiraka's quiz generation pipeline with human approval was used, testing whether it is capable of generating quizzes of comparable quality.

4.3 Results and Outcomes

4.3.1 User Trials 1

User feedback regarding DocMode and FlashMode was as expected, with most users saying that while DocMode is more familiar, FlashMode can sharpen reading focus by removing the ability to re-read. However, recorded metrics also showed that users rarely adjusted the reading pace between texts, showing that they were not challenging themselves to improve their reading speed. For DocMode, there were mixed reviews for the karaoke-style pointer feature, with some saying it was useful and others saying it was unnecessary and unhelpful.

4.3.2 User Trials 2

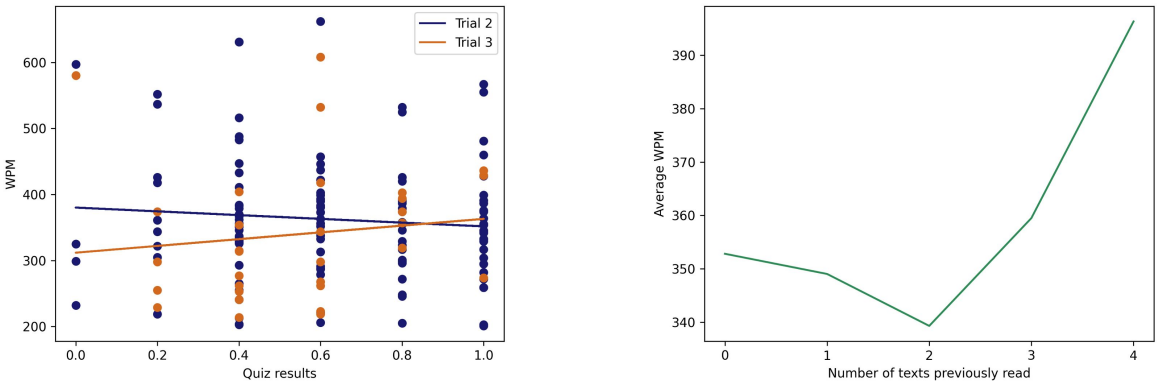
In general, user experience was reportedly high as 74% of participants approved of the website design and 82% said that they would recommend this platform to a friend. Specific feedback regarding key platform features is discussed below.

FlashMode Adaptive received mostly positive reviews, with 83% of users saying that their eyes were properly tracked, and the adjusted reading speeds challenged them to read faster. This initial feedback is supported by a macro-analysis of recorded WPMs and quiz scores, showing no significant decrease in comprehension when reading speed increases (Figure 3). However, common issues included the tendency to re-read chunks and falsely signal that users were finished reading, inaccuracy during blinking, and difficulties for users with glasses.

Feedback on the quizzes was also positive, with users saying that seeing their performance analytics motivated them to improve and 74% of participants rated quiz effectiveness at least a 4 out of 5. However, 65% of users also claimed that speed reading negatively impacted their understanding of the text, primarily due to reading speeds sometimes becoming too high. This is what Chunk Complexity aims to solve, reducing reading speed when necessary to ensure important information is properly digested. Users also suggested including more obscure topics as some questions could be answered using prior knowledge.

4.3.3 User Trials 3

Overall, 89% of users rated their experience positively, marking a 20% improvement. The platform also reported user approval ratings of 78% and 80% for website layout and website design respectively. For improved reading modes, reviews were mixed. For FlashMode, 54% of participants said that Chunk Complexity was an improvement and all participants agreed that it challenges them to read



(a) Quiz results vs WPM, a quantitative measure of the speed-comprehension trade-off

(b) Quantified learning curve using the average WPM of user as they continue to train on the Kiraka platform

Figure 3: Quantitative analysis of user trials results

faster. However, critiques regarding eye-tracking accuracy and false signalling persisted, likely due to deeper issues involving the WebGazer API.

In this trial, DocMode became much less popular than FlashMode, having only 5 recorded sessions in comparison to FlashMode's 48, suggesting that DocMode is an inherently less attractive feature. While 77% of participants agreed it is more useful than normal reading, there are repeated critiques that its non-adaptive pointer system made for a "robotic" reading experience with an over-emphasis on individual words.

As in the previous trial, recorded reading speeds and quiz results showed no correlation (Figure 3), signalling that users have been challenged to read faster without sacrificing comprehension. Analysis of participants' reading speeds over multiple sessions shows a steady increase in WPM after an initial decrease, as seen in Figure 3. This learning curve is likely due to users becoming more confident with the Kiraka platform after overcoming the initial unfamiliarity of Flashmode.

Human-approved quizzes received positive feedback, with 67% of users stating they found them sufficiently challenging. Furthermore, 80% found these quizzes to be more effective than the hand-crafted ones, showing a higher affinity for abstractive-based questions. However, this abstractive style also led to repeated comments about answer subjectivity, listing multiple options that are technically valid. These criticisms are more prevalent in the AI-generated quizzes from self-uploaded text, not being able to benefit from a human filter.

5 Evaluation

5.1 Software Engineering

To evaluate the Kiraka platform's performance and software engineering practices, Google Lighthouse [31] was used, a popular benchmarking tool for websites. As shown in Figure 15 in Appendix G, results were excellent, with a performance score of 99/100, an accessibility score of 94/100 and a perfect score of 100 for Search Engine Optimisation (SEO). However, one area of improvement was "best practices," having a score of 78/100, due to the website's use of third-party cookies from Clerk as shown in Figure 16 in Appendix G, but we chose to keep using Clerk since it offers many security benefits. As browsers like Google Chrome are moving towards not supporting third-party cookies, this presents a challenge, requiring us to balance Clerk's security benefits with evolving web privacy standards.

WRK2 [32] was also used to simulate user load on the website under worst-case scenarios, specifically when a large number of users access the site simultaneously. This stress test is designed to evaluate our website's scalability by varying the number of connected users from 5 to 50. In stress testing, we assumed a conservative five-second interval per user request. Given that WebGazer tracking and reading modes operate client-side, this actually provides an overestimate of server demand.

As illustrated in Figure 17 in Appendix G, latency becomes significant when the user count exceeds 20 (approx. four requests per second), with mean latency remaining under 200ms (and 99.99% percentile latency only being slightly above 200) for up to 10 users but rising sharply beyond that point. This indicates acceptable performance for a limited number of users but potential scalability issues. During tests, uneven CPU load distribution across four cores were discovered during peak demand, as evidenced in Figure 18 in Appendix G. To address this, we plan to implement multiple server instances and a load balancer to more evenly distribute the workload and increase our capacity for simultaneous connections.

The back end can perform reliably and securely. The VM is configured to automatically launch the website when booted, increasing the reliability of the software, and requests always require the Clerk ID of the user, adding a layer of security. On the other hand, the long execution time of the quiz generation can cause a socket connection timeout in the request when uploading a text. While the request is still handled properly, a response is never sent to the front end. Additionally, although the database follows most normalisation principles, it does not conform to the third normal-form, making it vulnerable to logical errors.

5.2 DocMode

Throughout user trials, the functional performance of DocMode remained high. Meeting the technical specification, DocMode successfully provides users with an accurate pointer that highlights words at the WPM rate and converts texts into hyperbold style. In all trials, there were no reported operational issues, such as lag in the transition of the pointer or unresponsiveness in manual control buttons like, start/pause. This underlines the appropriate technical implementation of DocMode, particularly the interface in the front end and text fetching in the back end.

The purpose of DocMode is to train users to read faster. Although this is a relatively standard feature, user trials showed positive feedback on its ability to train users for real-life reading situations, iteratively improving with each trial. From User Trial 1, only slightly above 50% of participants found the DocMode to be useful for increasing reading pace. Specifically, the negative reviews mentioned the pointer and hyperbold were sometimes distracting, making speed reading unproductive as users were not able to turn on/off features. As such, these features became optional. Additionally, the pointer size (the number of words that can be highlighted) and text font size also became adjustable. The effects of the improved features can be seen in User Trial 3, having a 22% increase in participants who found DocMode to be useful (77% total). Despite the promising comments, DocMode's HyperBold feature can be improved to bold syllables of the word rather than the first few characters.

5.3 FlashMode

FlashMode, the project's centerpiece, was refined through multiple iterations, with each update enhancing its technical performance and user utility. Test-driven development addressed many issues inherent to eye-tracking, such as poor calibration and pixel offsets (referred to in Section 3.1.5). When working as intended, the feature is intuitive, only requiring users to return their gaze left when read is finished. As mentioned in Section 4.3, users who tried FlashMode generally reported a positive experience, appreciating the forced focus and enhanced reading speed.

Beyond engineering quality, FlashMode's utility as a speed-reading trainer is supported through qualitative and quantitative analysis. Feedback confirms that users like FlashMode, having a 67% approval rating overall, and all users agreed that they were pushed to read faster. As seen in Figure 3a, FlashMode's auto-provided speeds also did not result in any reductions in quiz performance, showing that users were adequately challenged without surpassing the speed-comprehension trade-off. The learning curve seen in Figure 3b also supports that the platform is intuitive, with performance quickly increasing within a few reading sessions. As indicated in Section 4.3, FlashMode has proven more beneficial than DocMode, as evidenced by more favourable written reviews and greater popularity.

However, FlashMode does have some drawbacks. Users report occasional inaccuracies such as inconsistent eye-tracking exacerbated by blinking, sometimes confusing the system to increase reading speed greatly. When provided reading speeds were very high, users also often struggled to regain

control, which caused further acceleration and disruption of the reading flow. Furthermore, the reduced duration of user trials does not allow for a proper assessment of FlashMode's long-term effectiveness in improving reading speed. While the engineering quality and user experience can be analysed, a longer, scientific study is required to assess its actual utility as a speed-reading trainer.

5.4 Quiz Generation

While only tested in the final user trial, Kiraka's pipeline performed well, having 80% of users agreeing that the AI-generated quizzes were of similar or greater quality in comparison to the standard, hand-crafted quizzes. Questions are also generated with minimal latency, creating new questions approximately every two seconds. Given its abstraction-based question format, user feedback has been positive, indicating that the AI-generated quizzes test general understanding in comparison to extractive questions that test small details. The human approval system ensures a quality check, with the AI's output generally aligning with user preferences.

Despite these advantages, the current pipeline does have its limitations. Without human approval, quiz quality becomes inconsistent, with useable questions only being generated about 50% of the time. Incorrect answers and false options are often observed, thus requiring extensive fine-tuning or a sophisticated but automated validation process in the future. Additionally, the current question format might be too abstractive, occasionally resulting in simple questions, such as asking for title recommendations, without exploring deeper comprehension. The reliance on two large LLMs also contributes to a high computational load, given their considerable size (2.8 GB each). Another constraint is the LLMs' training on English-only texts, limiting their multilingual capabilities.

5.5 User Experience and Analytics Page

The overall user experience is intuitive and visually pleasant, with feedback reporting 78% user approval. This is thanks to research-informed decisions such as using an off-white background for reduced eye strain and feedback-driven improvements like the instructions page. Popup boxes were also implemented to guide users, providing helpful information about each feature or page's purpose. The user analytics page also received positive feedback, with 61% of users stating that it helped motivate them while identifying areas of improvement. However, feedback also indicated that the popup boxes were often too long and generic, with some covering the entire page. In the future, smaller, localised popups should be used, allowing users to digest information in segments without overwhelming them.

6 Conclusion and Future Work

In conclusion, our project has successfully addressed key limitations in speed-reading platforms through Kiraka's user-centric adaptivity and training focus. Through the digitisation of established techniques, DocMode provides a familiar and traditional approach to speed reading, supported by 78% of trial users. Even so, FlashMode proved to be more popular during user trials, evidencing the utility of FlashMode's innovative approach. Aligned with our original objective of maintaining reading comprehension, FlashMode successfully challenges readers without exceeding their limits, with users showing no decrease in quiz performance at higher reading speeds. The quality AI-generated quizzes themselves was also rated highly, ensuring that reading comprehension was accurately measured. However, the current pipeline still requires human approval, keeping the self-upload feature in beta development until its consistency can be improved.

In future work, we aim to streamline the quiz generation pipeline by eliminating the need for human approval. This could be achieved either by replacing human oversight with an LLM or by fine-tuning a larger pre-trained model, which may potentially demonstrate enhanced performance in our downstream task. Additionally, inspired by [33], we plan to improve the chunking techniques used in FlashMode through the integration of a fine-tuned LLM. This model will identify optimal moments in the text for inserting pauses, thereby guiding FlashMode on where to reduce reading speed.

References

[1] K. Rayner, E. R. Schotter, M. E. Masson, M. C. Potter, and R. Treiman, "So much to read, so little time: How do we read, and can speed reading help?" *Psychological Science in the Public Interest*, vol. 17, no. 1, pp. 4–34, 2016.

[2] R. Spence, "Rapid, serial and visual: A presentation technique with potential," *Information Visualization*, vol. 1, no. 1, pp. 13–19, 2002. [Online]. Available: https://doi.org/10.1057/palgrave.ivs.9500008

[3] M. Migoli, G. Öquist, and S. Björk, "Evaluating sonified rapid serial visual presentation: An immersive reading experience on a mobile device," vol. 2615, 10 2002, pp. 508–523.

[4] S. Benedetto, A. Carbone, M. Pedrotti, K. Le Fevre, L. A. Y. Bey, and T. Baccino, "Rapid serial visual presentation in reading: The case of spritz," *Computers in Human Behavior*, vol. 45, pp. 352–358, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747563214007663

[5] H. Nishida, "The influence of chunking on reading comprehension: Investigating the acquisition of chunking skill," *The Journal of Asia TEFL*, vol. 10, pp. 163–183, 2013. [Online]. Available: https://www.researchgate.net/publication/289208481

[6] B. H. I. Kana'an, S. D. A. Rab, and A. Siddiqui, "The effect of expansion of vision span on reading speed: A case study of EFL major students at King Khalid University," *English Language Teaching*, vol. 7, p. p57, 9 2014. [Online]. Available: https://ccsenet.org/journal/index.php/elt/article/view/40553

[7] Spreeder, "Spreeder - speed reading app & software." [Online]. Available: https://www.spreeder.com/

[8] SwiftRead, "Swiftread - speed reading software." [Online]. Available: https://swiftread.com/

[9] A. Papoutsaki, P. Sangkloy, J. Laskey, N. Daskalova, J. Huang, and J. Hays, "Webgazer: Scalable webcam eye tracking using user interactions," in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI, 2016, pp. 3839–3845.

[10] Doing ADHD, "Bionic reading: Game-changer for adhd readers," 2023, accessed: 2024-04-29. [Online]. Available: https://doingadhd.com/2023/technology/reading/bionic-reading-game-changer-for-adhd-readers/

[11] K. Schwaber and J. Sutherland, "The scrum guide: The definitive guide to scrum: The rules of the game," 2020, accessed: 2024-04-29. [Online]. Available: https://scrumguides.org/scrum-guide.html

[12] Notion, "Your connected workspace for wiki, docs projects — notion." [Online]. Available: https://www.notion.so/

[13] M. Teams, "Video conferencing, meetings, calling — microsoft teams." [Online]. Available: https://www.microsoft.com/en-gb/microsoft-teams/group-chat-software

[14] Vercel, "Next.js by vercel - the react framework." [Online]. Available: https://nextjs.org/

[15] M. Calligeros, "Seeing green boosts your concentration, research shows," 5 2015. [Online]. Available: https://www.smh.com.au/technology/seeing-green-boosts-your-concentration-research-shows-20150525-gh8udh.html

[16] L. Rello and J. P. Bigham, "Good background colors for readers: A study of people with and without dyslexia," in *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. ASSETS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 72–80. [Online]. Available: https://doi.org/10.1145/3132525.3132546

[17] D. Scotland, "Contrasting advice – what colours are best for accessibility? — dyslexia scotland - dyslexia scotland," 7 2023. [Online]. Available: https://dyslexiascotland.org.uk/contrasting-advice-what-colours-are-best-for-accessibility/

[18] Clerk, "Clerk — authentication and user management." [Online]. Available: https://clerk.com/

[19] Y. Hisao, "A historical study of typewriters and typing methods: from the position of planning japanese parallels," *Journal of Information Processing*, vol. 2, no. 4, pp. 175–202, 02 1980. [Online]. Available: https://cir.nii.ac.jp/crid/1050001337894287488

[20] P. Manakul, A. Liusie, and M. J. Gales, "Mqag: Multiple-choice question answering and generation for assessing information consistency in summarization," *arXiv preprint arXiv:2301.12307*, 2023.

[21] A. Ushio, F. Alva-Manchego, and J. Camacho-Collados, "Generative Language Models for Paragraph-Level Question Generation," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Abu Dhabi, U.A.E.: Association for Computational Linguistics, Dec. 2022.

[22] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, J. Su, K. Duh, and X. Carreras, Eds. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. [Online]. Available: https://aclanthology.org/D16-1264

[23] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[24] G. Lai, Q. Xie, H. Liu, Y. Yang, and E. Hovy, "RACE: Large-scale ReAding comprehension dataset from examinations," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 785–794. [Online]. Available: https://aclanthology.org/D17-1082

[25] M. Shardlow, R. Evans, G. H. Paetzold, and M. Zampieri, "Semeval-2021 task 1: Lexical complexity prediction," pp. 1–16.

[26] A. Nandy, S. Adak, T. Halder, and S. M. Pokala, "cs60075_team2 at SemEval-2021 task 1 : Lexical complexity prediction using transformer-based language models pre-trained on various text corpora," in *Proceedings of the 15th International Workshop on Semantic Evaluation (SemEval-2021)*. Online: Association for Computational Linguistics, Aug. 2021, pp. 678–682. [Online]. Available: https://aclanthology.org/2021.semeval-1.87

[27] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv:2004.05150*, 2020.

[28] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," 2021.

[29] M. Guo, J. Ainslie, D. Uthus, S. Ontanon, J. Ni, Y.-H. Sung, and Y. Yang, "Longt5: Efficient text-to-text transformer for long sequences," *arXiv preprint arXiv:2112.07916*, 2021.

[30] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M.

Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[31] Google, "Lighthouse: An open-source, automated tool for improving the quality of web pages," Software available from Google Lighthouse web page, 2024.

[32] G. Tene, "wrk2: A constant throughput, correct latency recording variant of wrk," 2023, [Online; accessed 28-April-2024]. [Online]. Available: https://github.com/giltene/wrk2

[33] J. Busler and A. Lazarte, "Reading time allocation strategies and working memory using rapid serial visual presentation," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 43, no. 9, pp. 1375–1386, 2017.

Appendices

A Diagram overview of the Website

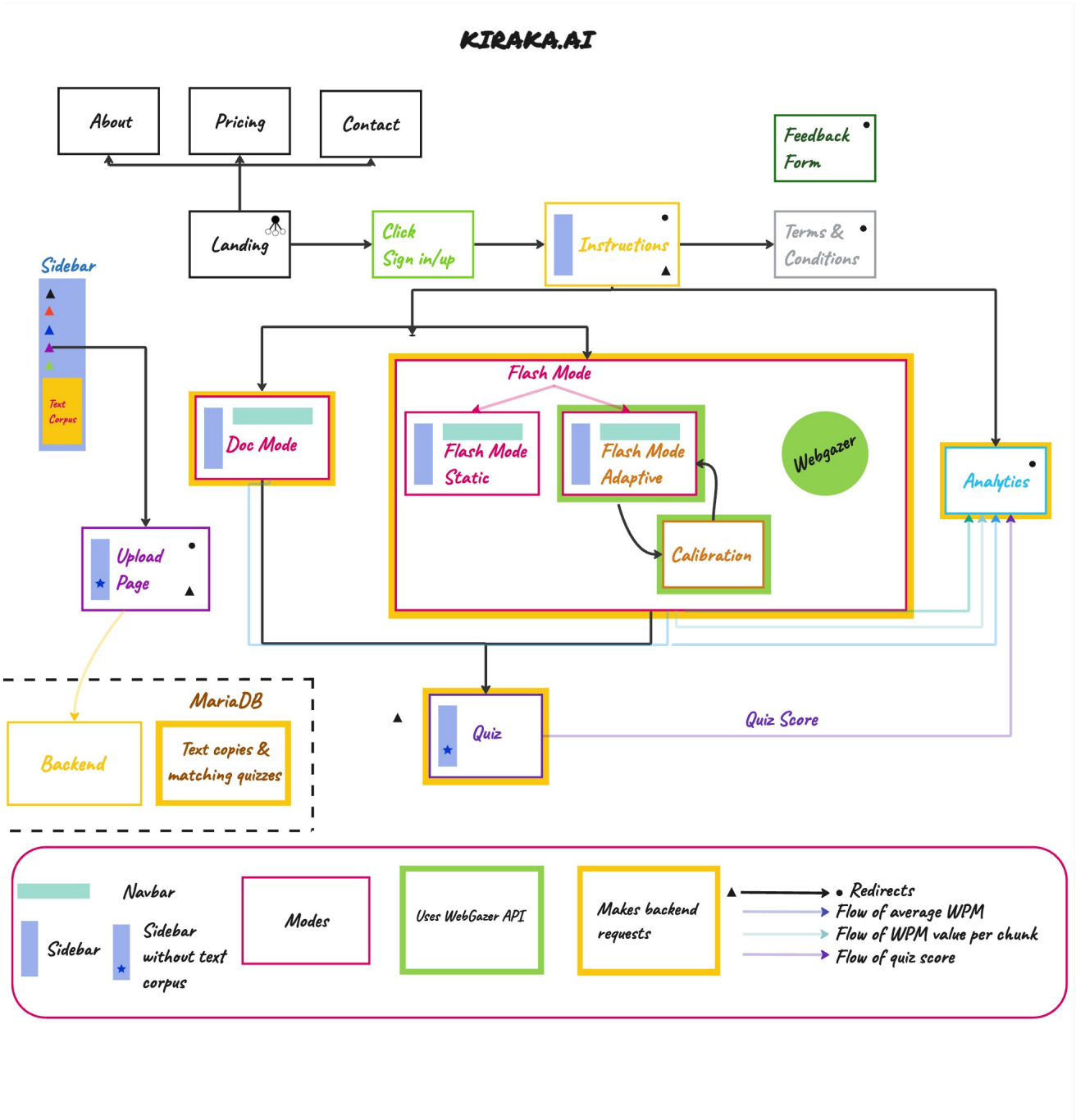


Figure 4: Diagram overview of the website, showing all the pages and connections

B Entity Relationship Diagram of the database

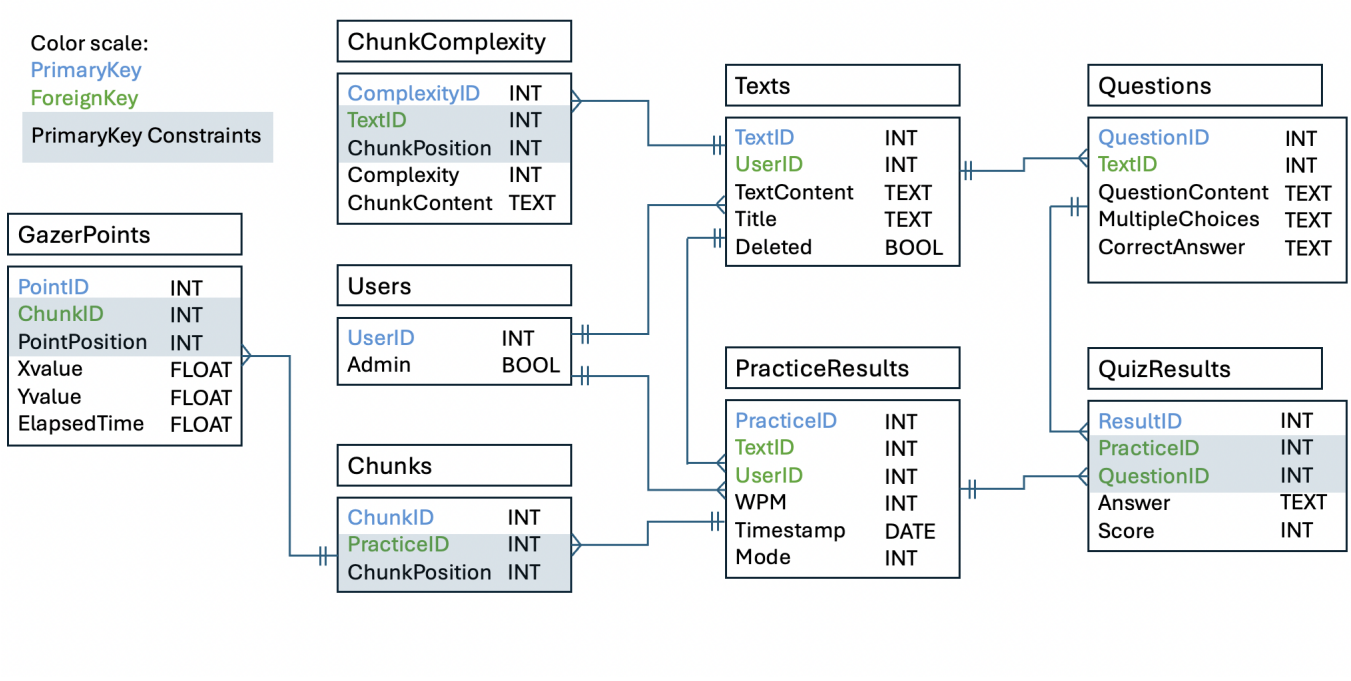


Figure 5: Entity Relationship Diagram of the database

C Project Management

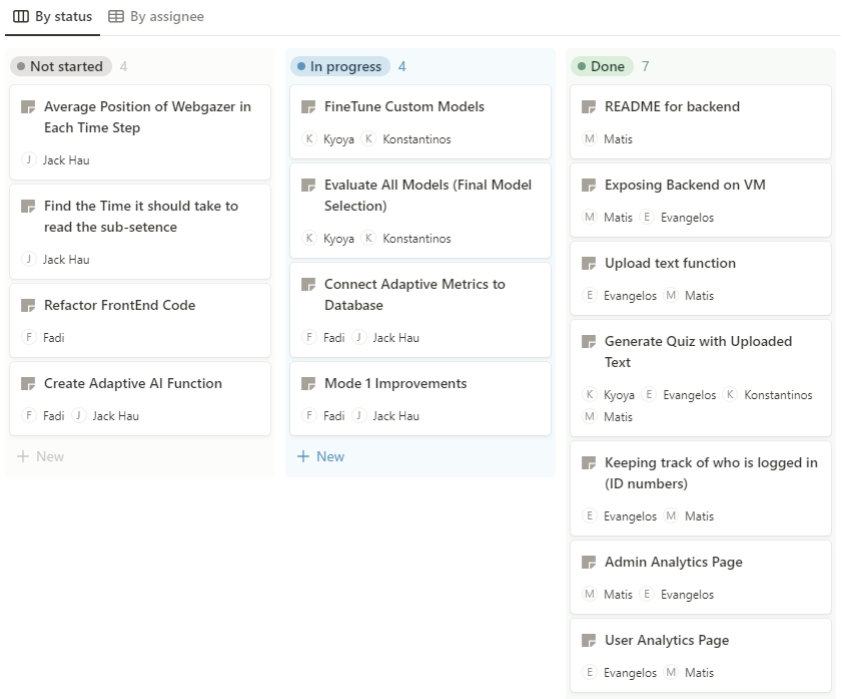


Figure 6: Example of our Notion sprint board for one particular sprint

D WebGazer Related Information

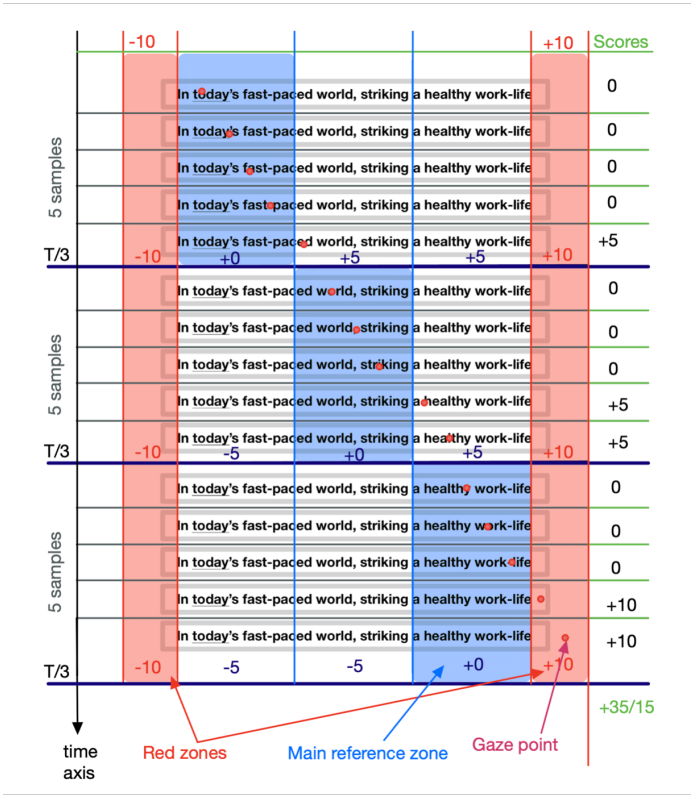
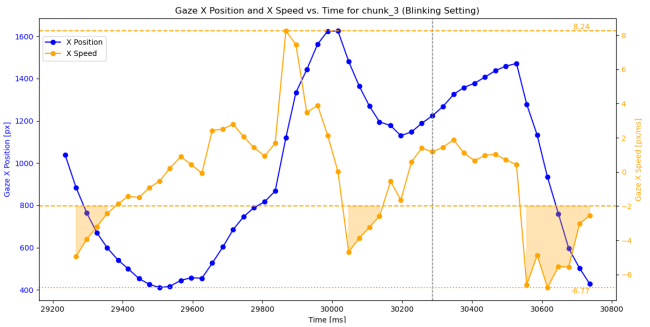
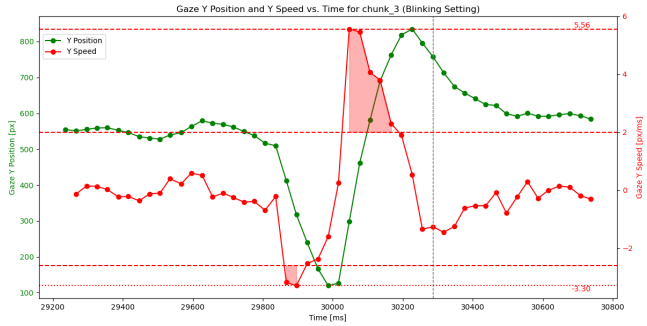


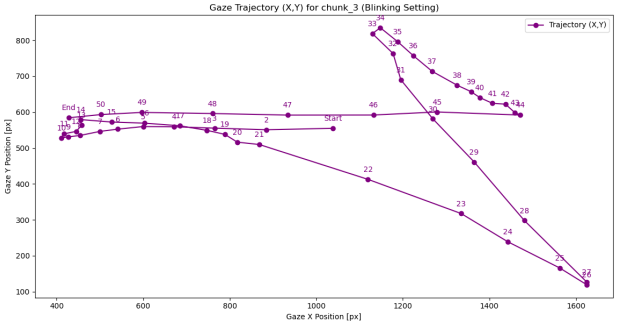
Figure 7: The scoring system for each of the 5 separated sections to determine WPM adjustment at each time step.



(a) Gaze X Position and X Speed vs Time

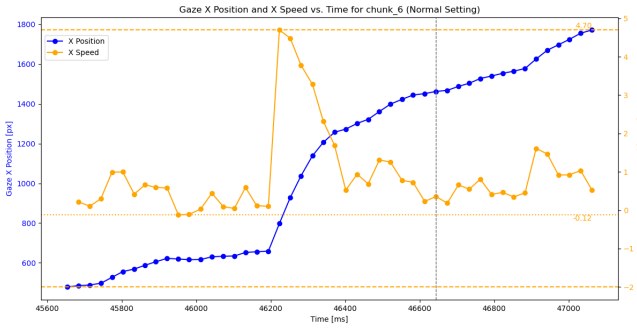


(b) Gaze Y Position and Y Speed vs Time



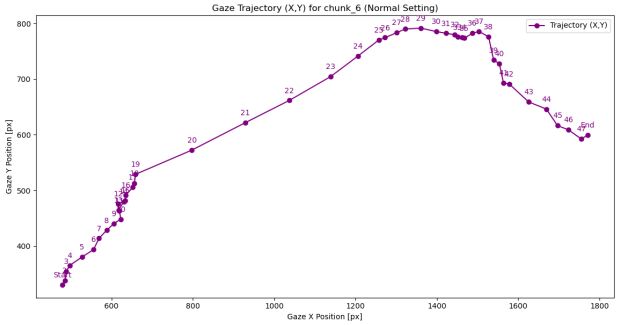
(c) Gaze Trajectory

Figure 8: Chunk 3 of a text for Blinking Setting



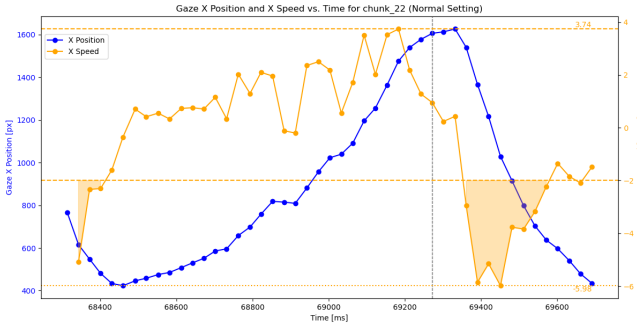
(a) Gaze X Position and X Speed vs Time

(b) Gaze Y Position and Y Speed vs Time

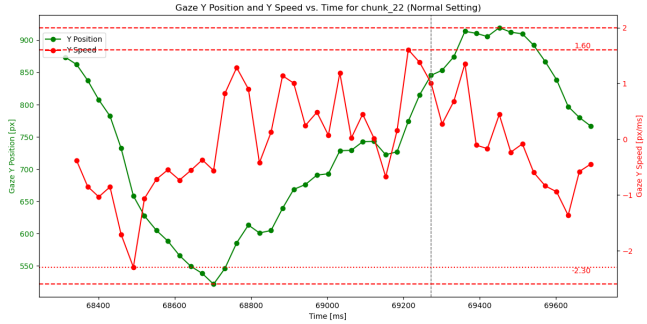


(c) Gaze Trajectory

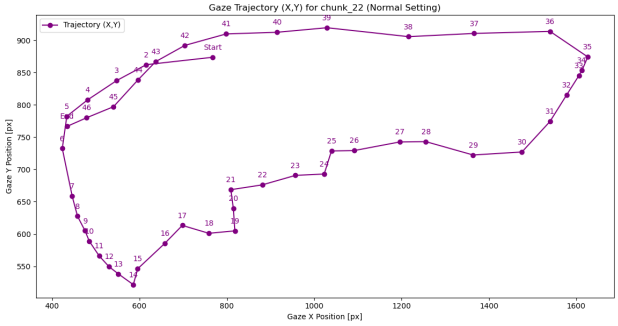
Figure 9: Chunk 6 of a text for Normal Setting



(a) Gaze X Position and X Speed vs Time



(b) Gaze Y Position and Y Speed vs Time



(c) Gaze Trajectory

Figure 10: Chunk 22 of a text for Normal Setting

Gaze Data Detailed Stats Over 32 Chunks – Normal Setting

Metric	Average	Std Dev	Min	Max
X Speed Range [px/ms]	10.96831778	2.545064018	4.817928024	15.24240624
X Max Speed [px/ms]	4.017154108	1.018206697	2.747542148	7.330863512
X Min Speed [px/ms]	-6.951163671	2.355782985	-11.22441148	-0.121648108
Y Speed Range [px/ms]	3.584798872	0.724179596	2.436341452	5.171738456
Y Max Speed [px/ms]	1.597760316	0.477305798	0.806178901	2.79576275
Y Min Speed [px/ms]	-1.987038556	0.588350411	-3.384093733	-0.772515744
Metric	Average	Std Dev	Min	Max
X Position Range [px]	1260.274912	114.2973956	997.567323	1495.272231
X Max Position [px]	1711.843799	97.22191967	1424.436757	1982.419078
X Min Position [px]	451.568887	59.99585013	363.9246482	649.3276378
Y Position Range [px]	385.2763913	91.52352564	229.2371744	578.4629805
Y Max Position [px]	849.6258205	58.30883886	649.2993185	933.7809909
Y Min Position [px]	464.3494293	103.3137044	251.0081716	645.9376452

Gaze Data Detailed Stats Over 32 Chunks – Blinking Setting

Metric	Average	Std Dev	Min	Max
X Speed Range [px/ms]	12.40854005	2.489288948	6.719480014	16.69336128
X Max Speed [px/ms]	5.5674787	2.036604808	2.833862222	11.16880088
X Min Speed [px/ms]	-6.841061347	1.922214991	-10.55526475	-2.811368311
Y Speed Range [px/ms]	7.934225703	2.205711478	2.185520112	10.51461181
Y Max Speed [px/ms]	4.122029136	1.210788129	0.881307467	5.762081045
Y Min Speed [px/ms]	-3.812196567	1.253235891	-5.918752731	-1.048972334
Metric	Average	Std Dev	Min	Max
X Position Range [px]	1269.658295	160.9215175	763.4582447	1547.871375
X Max Position [px]	1595.657563	137.556454	1234.070456	1842.608588
X Min Position [px]	325.9992674	66.21074499	187.1706308	470.6122111
Y Position Range [px]	529.8317962	165.7105253	96.83656411	832.400038
Y Max Position [px]	789.0112119	77.90731412	588.2951703	921.0527633
Y Min Position [px]	259.1794157	137.0314384	50.67062495	599.1549806

Figure 11: Gaze data from WebGazer detailing the speed range, max speed, min speed, position range, max position, and min position of X and Y direction for both normal and blinking settings.

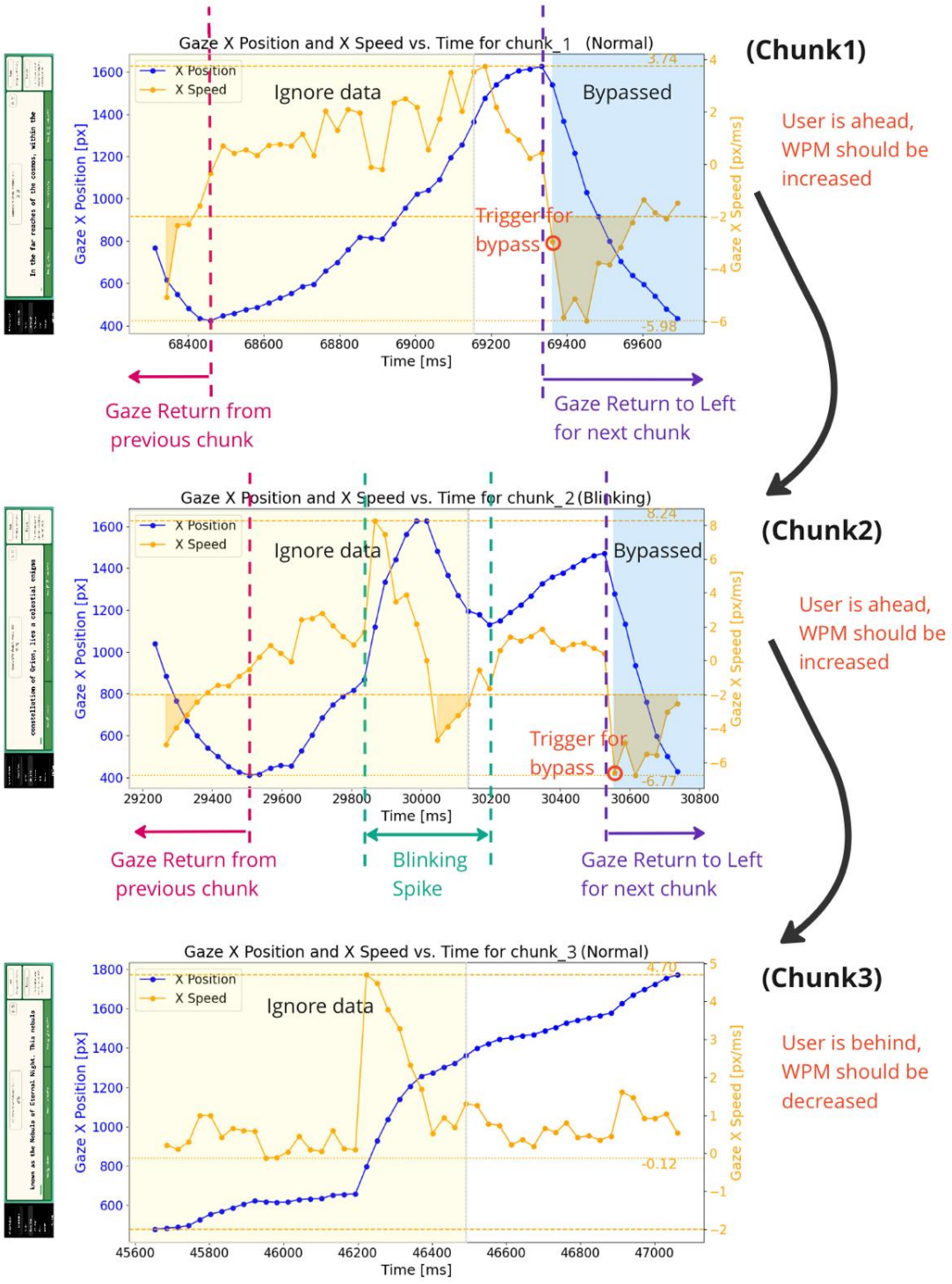


Figure 12: Cat-and-mouse WPM adjustment strategy

E LLM

Level	Default Handling	Exceptions / Conditions
Unigram	Use cosine similarity	1. **4-digit numbers:** Sample from ±10 of the number itself. 2. **Other numbers:** Use cosine similarity, ensure it's a number. 3. **Names:** Ensure returned options are capitalized.
Bigram	Use cosine similarity for the whole bigram	1. **4-digit number in bigram:** Treat as pair of unigrams; sample from ±10 for the number; use cosine similarity on other unigram; return pair in original order. 2. **Non-4-digit number in bigram:** Use cosine similarity, ensure it's a number. 3. **Names:** Ensure returned options are capitalized. 4. **Determinants, prepositions:** Do not change.
Trigram	Default: split on the determiner, then apply appropriate cosine similarity on units (e.g., unigram-det-unigram, det-bigram)	1. **det-det-unigram (e.g., at least 30000):** Do not accept; generate new pair. 2. **Unigram is a 4-digit number:** Apply resampling technique (sample from ±10). 3. **Unigram is another number:** Use cosine similarity, ensure it's a number.

Table 3: Handling Rules and Exceptions for Unigram, Bigram, and Trigram

Pros	Cons
QA pairs almost invariably align with the context, providing a higher relevance compared to extractive methods.	Abstractive questions may differ fundamentally, such as asking for the best title for a section, which requires specific attention.
The inference process for generating 20 QA pairs is relatively swift, taking approximately 23 seconds without word count restrictions.	Occasional inaccuracies can occur, producing less accurate answers for well-formed questions.
The separate LLM designated for distractors efficiently generates plausible and contextually appropriate false options.	Rarely, a nonsensical QA pair is produced, which can undermine the quality of the output.
The distractor LLM can handle answers of any length, offering versatility in processing varied input.	Distractors may be repeated exactly, although these are easily identifiable through direct string comparison, suggesting a need for refinement in generation rules.
The capability to generate exactly three options consistently, with the option to vary these through sampling, enhances the flexibility of the system.	The additional LLM for distractors increases both inference time and storage requirements, potentially impacting system performance.

Table 4: Advantages and Disadvantages of Quiz Generation Pipeline

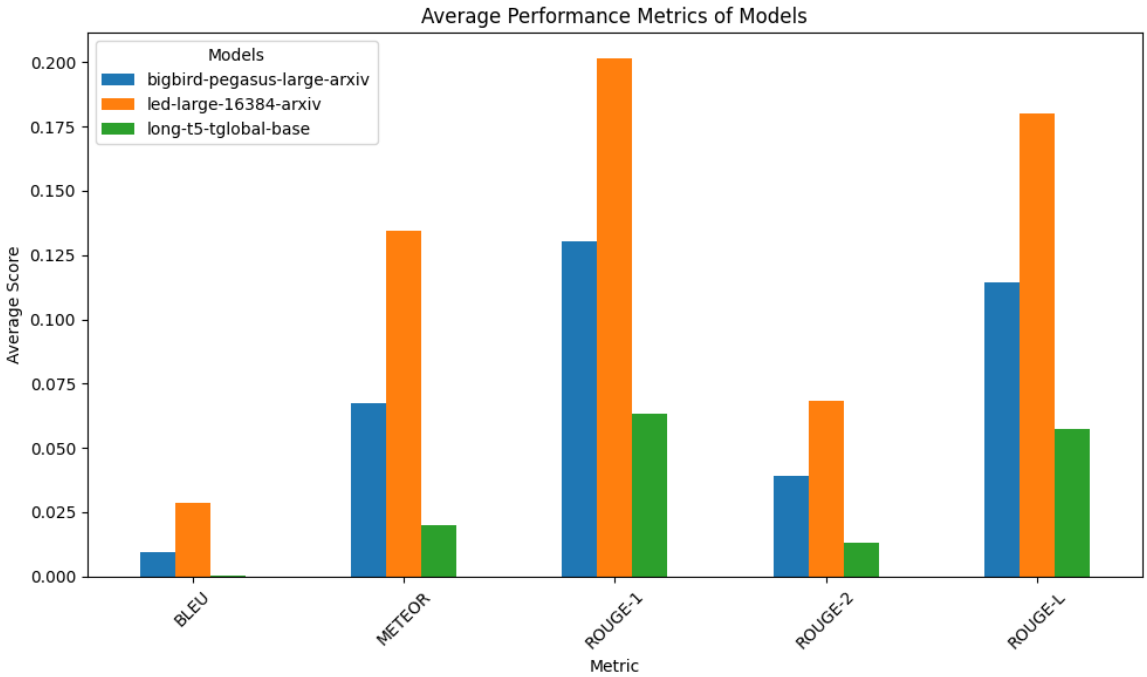


Figure 13: Performance of all open-sourced fine-tuned models regarding summary generation of scientific articles.

F User Trial Feedback

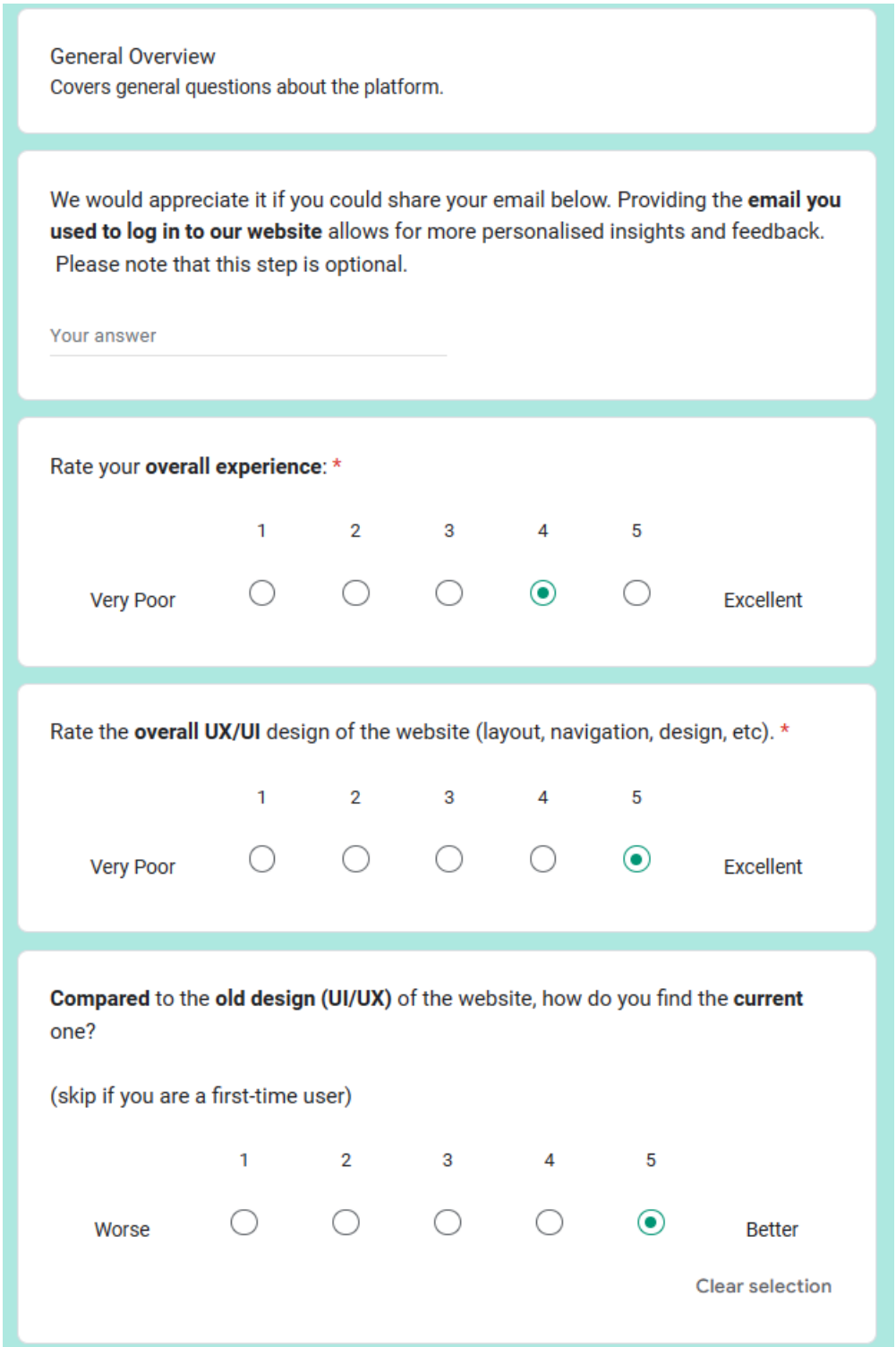


Figure 14: Example of the User Trial Feedback form

G Software Engineering Evaluation scores

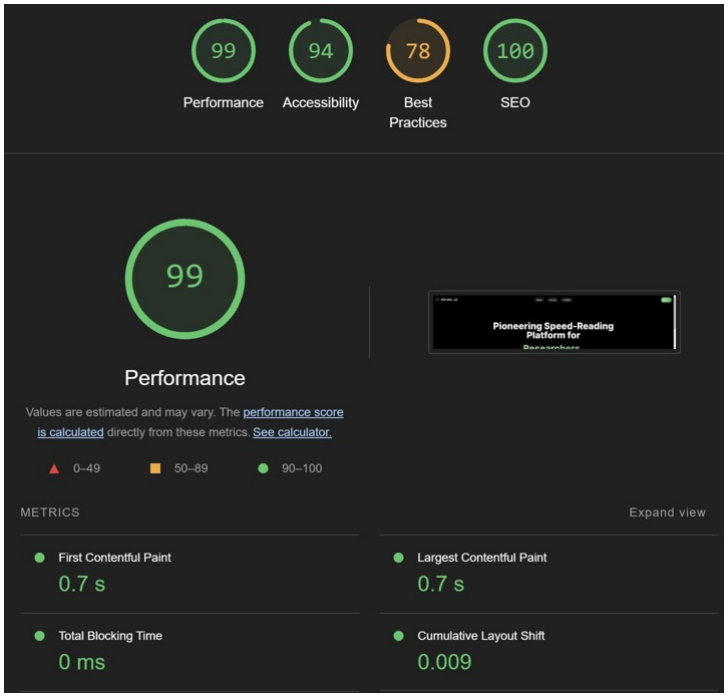


Figure 15: Google Lighthouse Scores

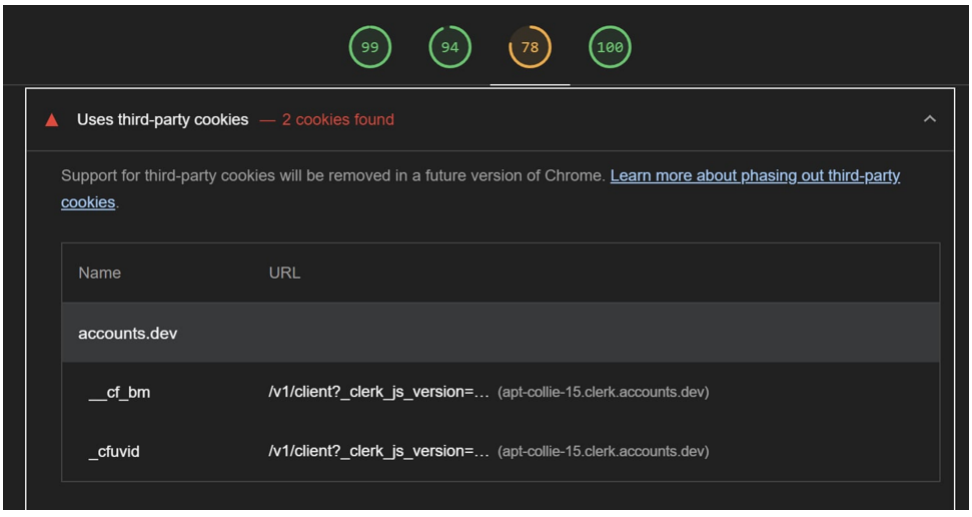


Figure 16: Google Lighthouse "Good Practices" Score Breakdown

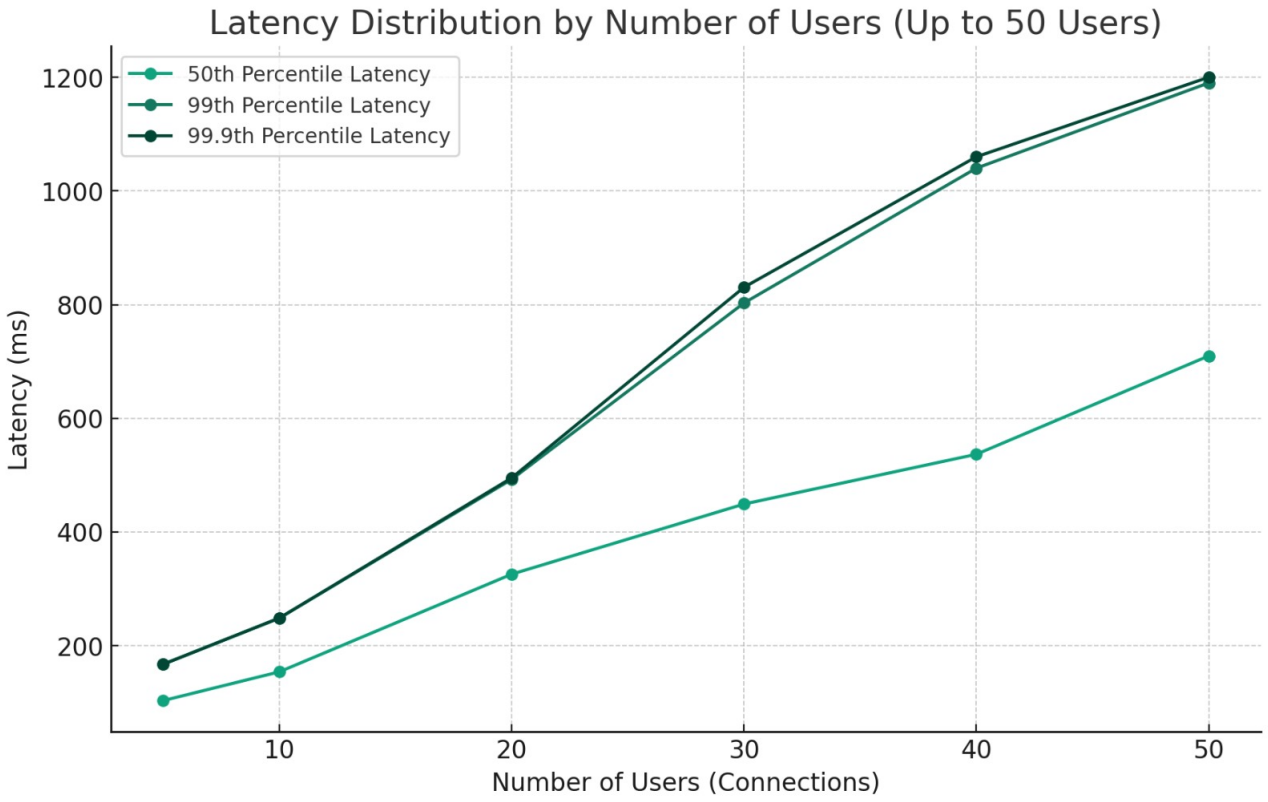


Figure 17: Latency plot for various number of simulates users

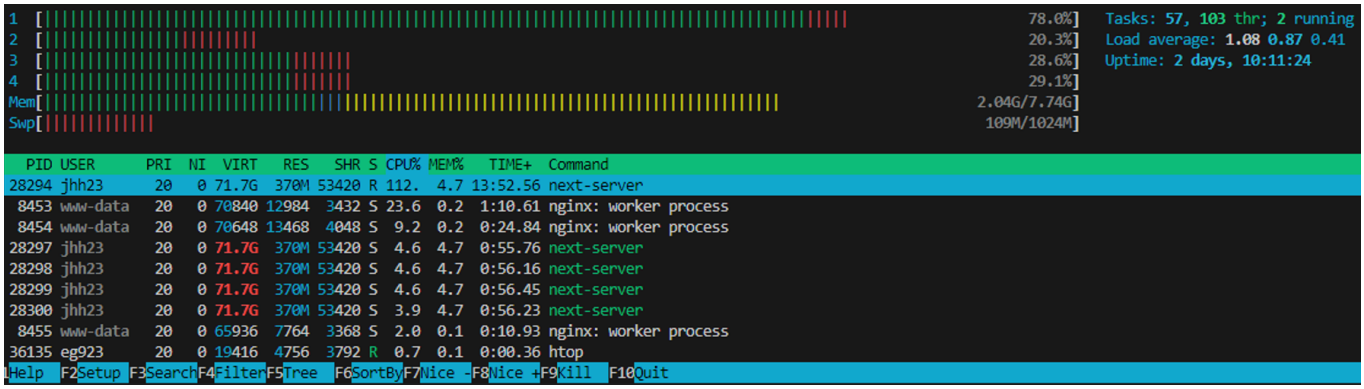


Figure 18: 'htop' output displaying real-time system resource utilisation during peak server load conditions.