Project Planning - This section captures all activities completed during the Project Planning phase. It should include:

1. A summary of the project plan outlining the sequence of user stories and development milestones.
    - Project Planning: Mobile Application Development
        - In the planning phase, our focus was on answering the fundamental question: "How do we build a mobile application that can easily be developed, tested, maintained, and distributed?" Since our target users are visually impaired individuals, we prioritized accessibility, real-time object recognition, and intuitive audio feedback. We decided to develop the app using React Native with Expo, allowing deployment on both Android and iOS while ensuring cross-platform consistency and ease of maintenance. The Expo framework was chosen for its streamlined development tools, robust camera API, and built-in support for media playback.
    - Mapping User Stories to a Product Plan
        - Object Identification (High Priority)
            1. Implemented first since it forms the foundation for navigation and awareness.
            2. Required integrating Expo Camera and object recognition models.
        - Traffic Awareness (Dependency on Object Identification)
            1. Built upon object detection to analyze moving objects (vehicles, pedestrians).
            2. Required real-time processing for timely feedback.
        - Location-Based Queries (Dependency on Object Detection & User Input)
            1. Needed integration with GPS and location APIs for contextual responses.
            2. Allowed users to query nearby landmarks like bus stops.
        - Customizable Audio Feedback (Enhances User Experience)
            1. Enabled speech rate adjustments and voice selection via Expo AV.
            2. Stored preferences for a personalized experience.
        - Real-Time Guidance (Requires Continuous Camera Processing)
            1. Developed once object recognition was stable.
            2. Provided continuous feedback on obstacles and navigation routes.
        - Shopping Assistant (Extension of Object Recognition)

1. Implementing OCR (Optical Character Recognition) for reading aisle signs and product labels.
2. Allowed users to compare items based on nutritional data.

2. The final system model (e.g., class diagram, sequence diagram, component diagram) that represents the project's architecture. You can refer to these examples and slides 8 and 30 discussed in class lectures.

Model diagram (general flow overview)

https://lucid.app/lucidchart/799fae2e-a424-4b69-b8aa-3b926df7ab11/edit?viewport_loc=43%2C4%2C2217%2C1087%2C0_0&invitationId=inv_6e532b3e-d16f-4b76-a63d-afdb91fb1000

3. The division of work into three sub-teams, explaining the reasoning behind the distribution.

Mobile Development – Focuses on building the mobile front-end using Next.js + React, ensuring a smooth user experience across devices.

Web Infrastructure & Media Processing – Manages image/video encoding and hosting, ensuring efficient storage, retrieval, and display of media assets as well as enabling the ML team and mobile devs can take advantage of integration into one cohesive platform.

AI & Audio Processing – Handles text-to-speech and vision processing using OpenAI and GPT-4o, integrating advanced AI capabilities into the platform.

4. Summary of the technology stack chosen for the project (e.g., programming languages, frameworks, databases, cloud services, etc.) and why.

The project utilizes Next.js + React for the front-end with TypeScript as the primary coding language and TailwindCSS for styling, ensuring a typesafe, scalable, and modern UI architecture. The back-end is built with JavaScript/TypeScript, leveraging Next.js (preferred) or Express.js for API handling. Next.js is a simple and scalable solution with a lot of features (routing, package handling) out of the box, and a couple of our team members have prior experience working with it. PostgreSQL or SQLite (for lightweight initial development) is used for data storage, managed via Prisma ORM for efficient queries and to avoid having to use the SQL language. GPT-4o will be used for vision processing and we will use OpenAI's text-to-speech API. From our testing, these

are the best performing models available to us. Cordova/Capacitor enables cross-platform mobile development. The stack is chosen for its flexibility, scalability, and strong ecosystem support. Next.js + React enables both static and dynamic rendering, while TypeScript ensures maintainability by ensuring type-safety. PostgreSQL with Prisma provides efficient database management, and GPT-4o enhances vision processing and text-to-speech. With Capacitor/Cordova enabling cross-platform support, the stack balances performance, ease of development, and future-proofing.

Sub-Teams - Each sub-team adds a section that clearly documents the work they completed during the Implementation phase. It should include:

5. Team members in this sub-team and their roles.
6. A description of the specific features, modules, or components built by the sub-team.
7. Lessons learned from this phase.
    - What worked well?
    - What didn't work well? What can you do better for future deliverables?

Mobile Development Team

- **Andrew & Amaan** → Focus on building the mobile front-end.

  Implementation Work and Lessons Learned:

  On the mobile side, we used Expo Camera's new CameraView to capture images from a device's camera, returning the image data in base64 format, converted into a blob to allow for processing. We then posted this blob data to our Next.js API endpoint, which handled the crucial steps of image interpretation (via OpenAI) and converting the resulting text into synthesized speech. Finally, the endpoint responded with an audio stream, which was played on the device using Expo's expo-av library by converting the binary MP3 data into a buffer and saving it in the cache temporarily to allow playback. This approach kept the mobile code focused on user interaction (taking photos and playing audio), while delegating the text-generation and text-to-speech logic entirely to the backend, offering a clean separation of concerns and an efficient, end-to-end workflow. One of the major issues was setting up the app itself. Expo Go was used to help test the app, however because of networking issues, the Expo Go was failing to load the app itself. From reading online and researching, we were able to fix this by running the npm server with tunneling. An additional bug encountered that was not obvious was use of http, as it is blocked by ios and android by default unlike https, and from investigating the constant network errors we were able to determine the cause and fix it by adjusting the configuration for Expo. Overall, we learnt a lot on how to work with cross mobile development and how to use Expo's tools to aid us, allowing us to have a functioning mobile app that can be used on both ios and android to provide a tts description of photos.

Future Deliverables:

Currently, there are 3 tabs available, they are the default tabs for when a react native project is created, but we have kept them there as we have plans (stretch goals possibly) to add user handling (login and sign up) for future deliverables. Additionally the "video" feature is not yet implemented, as currently only manual pictures can be analyzed. We have put thought into how we can keep a live feed of our surroundings, through sending frames every x seconds (so we don't use too many api calls as it will cost money) and possibly queuing audio playback for each frame.

Web Infrastructure & Media Processing Team

- **Derek** → Handles ML infra
- **Henry** → Manages image URL generation and hosting.

Implementation Work and Lessons Learned:

What we did:

Our responsibilities consisted of:

1. Developing a pipeline to turn an image file into a publicly accessible static link (URL),
2. Creating web front-end pages so that users can interact with our application via a browser
3. To set up the hosting of the remote server.

We decided on pushing off more of the core model orchestration and user context ideas off to the next deliverable as we mainly wanted to get up and running the main individual pieces of functionality described in our user story before we get to piecing it all together in one cohesive system. We flexed to aid in making sure these deliverables were achieved so we can begin designing the next iteration of our overall system.

Firstly, the user provides us with a locally stored (on the user's device) image file. However, the OpenAI API requires a publicly available static image url. So we must host the image and generate a valid corresponding URL. The image pipeline can be summarised into the following steps:

1) We must encode the image into a base64 string so that it can be packaged in a JSON to make an http request to the server. (in generateImageLink function in imageInterface.tsx).
2) Once the image has been packed in a Json, we make a POST http request to our server's api endpoint for image uploading.
3) We then must implement the actual server side api endpoint/handler, which sanitizes the incoming http request, takes the given encoded image, decodes it back into the original image file format, and saves it to a publicly accessible static

folder. Finally, the api handler returns/responds by sending back the url to the statically hosted image (see /pages/api/image-process/image-url-generate.ts).
4) Finally, the front end interface (imageInterface.tsx) can return to the user the newly available image url - this url can now be used to make calls to OpenAI.

Moreover, a similar functionality was implemented for the Audio files to do essentially the same thing, though having this store raw audio files may be a value add in the future as it helps alleviate some overhead from the AI team as currently the audio transcription portion of the API requires first feeding in an audio file via base64 and then reconverting it back to an audio buffer file.

Secondly, we had to implement the web front-end. Here, we were able to reuse some code from an old next.js project which Henry had created in a past course. We had to reconfigure the project routes, and create new web pages for describing an image (file or url), and a web page to upload an image file. We also made adjustments to the project's web theme files as necessary (see pages under components/pages).

Finally, the most challenging aspect was deploying the project onto an external server. We ended up using a Digital Ocean Droplet, as it provides us with a virtual Ubuntu machine, which allows for significantly more customizability than other options (such as a Vercel serverless backend). To allow users to connect to our server, we had to configure Nginx to a) act as a reverse proxy (listens to ports for requests), and b) to properly serve our web pages and static image files. In this process, we had to carefully modify .env files and nginx config files, and we had to set up pm2 to properly run our next.js project in “build” mode permanently.

What we learned: For our web component and api pipeline, we learned how to make proper http requests. Overall, generating and processing http requests went well. What was more challenging was properly performing base64 encoding and the decoding. We spent a lot of time trying to debug the raw data that had been decoded, and it ended-up being resolved by using dataURL as the encoding method. While deploying the project, we learned how to properly configure Nginx and how to use pm2 to manage processes. PM2 turned out to be intuitive and easy to use. However, learning to understand Nginx was challenging. We had significant issues properly serving the public image folder, since Next.js in build mode does not refresh the /public folder, so it is not aware of any new images that are added after building. We ended up having to configure Nginx as a reverse proxy listening on a separate port, which then properly served the correct image files.

AI & Audio Processing Team

- **Daniel** → Works on text-to-speech.
- **James** → Works on speech-to-text.

Implementation Work and Lessons Learned:

During the implementation phase, our subteam developed an image processing feature that allows users to input an image URL and receive a description generated by GPT-4o. Additionally, we integrated text-to-speech (TTS), enabling users to hear the descriptions in various AI-generated voices, with a feature that lets them choose between different voice options.

Building on this, we recently implemented a speech-to-text (STT) feature that allows users to record their voice and have it transcribed in real-time using OpenAI's Whisper model. This feature enhances accessibility and provides an alternative input method for users who prefer verbal interactions over typing. While the implementation was straightforward, we encountered challenges with handling audio permissions across different devices and ensuring that recordings were properly formatted for Whisper's API. Debugging also required extensive testing due to variations in microphone quality and background noise affecting transcription accuracy.

Here is a video demonstrating our speech-to-text's capability
https://drive.google.com/file/d/1XA7DfGcKzXYRlBA8tZRH7ONwun7Rcplt/view?usp=sharing

Moving forward, better documentation, clearer commit messages, and enforcing a stricter code review process will help streamline development and reduce debugging time. Additionally, implementing automated tests for different audio input scenarios and refining our error-handling mechanisms will improve the reliability of the STT feature.

Team - This section provides a high-level overview of the team's collective progress in implementing the project. It should contain:

8. A detailed explanation of the common foundation built at the start of the Implementation phase.
9. A summary of how the sub-teams' work contributed to the overall project.
10. The team's **overall progress** in the context of the roadmap from the Planning phase.
11. Any major technical or organizational challenges encountered and how they were handled.

At the start of the Implementation phase, the team established a common foundation by setting up the project structure, defining core functionalities, and ensuring the necessary tech stack was in place. This included configuring the Next.js framework, integrating PostgreSQL with Prisma ORM, and setting up API routes to handle image processing and user interactions. Early planning helped streamline development across sub-teams and ensured smooth integration between components.

The **AI & Audio Processing Team** enhanced user accessibility by developing the image recognition and text-to-speech (TTS) system using GPT-4o. This feature allows users to input an image URL, receive a detailed description, and choose between different AI-generated voices for audio playback. By enabling both vision processing and voice synthesis, our sub-team contributed to the project's goal of providing an interactive and inclusive user experience.

Expanding on this, we also implemented **speech-to-text (STT)** functionality, allowing users to record spoken input and have it transcribed in real-time using OpenAI's Whisper model. This addition makes the platform more accessible to users who prefer voice input or have difficulties with typing. While integrating STT, we encountered challenges with handling audio permissions across different devices and ensuring accurate transcription in various recording environments. Addressing these issues required extensive testing, refining noise-handling mechanisms, and optimizing API calls to improve responsiveness.

Overall, the team has made steady progress in alignment with the roadmap outlined in the Planning phase. While the core functionalities have been successfully implemented, debugging undocumented errors in teammate code, particularly in URL validation and audio processing, posed significant challenges. Moving forward, we aim to mitigate these issues by enforcing stricter code reviews, improving documentation, and implementing automated testing for different input scenarios. These steps will enhance development efficiency and ensure a smoother user experience.

Web Infrastructure & Media Processing Team

We made great progress since our back-end and web-application are now hosted and available online! Our images are successfully being uploaded to a static home, and our external server can successfully serve both web and mobile clients. This is a huge milestone for our project, as a major source of concern initially was whether we would be able to host the project on a small server, and so far it is working well. Moreover, after completing our components, we now have a smooth and complete pipeline, from the user taking an image to the user getting a description of the image (visually and audio).

The Mobile Development Team focused on integrating Expo Camera's CameraView to capture images and seamlessly send them to the backend for processing. Using expo-av, we enabled real-time audio playback of AI-generated speech, ensuring an intuitive user experience. Mobile was critical to our use case, as camera-based interaction was essential for capturing images that drive the AI-powered analysis and text-to-speech pipeline. Our contributions ensured smooth integration between image capture, backend processing, and audio playback, making the system accessible and interactive.

Here is a video demo of the app: RPReplay_Final1739835324.mp4

running it on your own may not be possible right now due to use of .env file holding our keys and api paths, but will be adjusted in the future to work.

In terms of overall progress, the team successfully implemented most planned features in alignment with the roadmap from the Planning phase. However, one of the major technical challenges we faced was dealing with undocumented errors in teammate code, particularly with URL validation. This led to unexpected debugging efforts, requiring close collaboration between team members to resolve the issues. Moving forward, better documentation, improved communication, and enforcing stricter code reviews will be crucial in preventing similar setbacks and ensuring a more efficient development process.

Another technical challenge we have encountered has band limits on the size of http requests. Firstly, we had to manually adjust the preset limits to allow packages of up to 4Mb. On the mobile-front-end, we have begun to pre-process images to reduce their footprint (i.e reducing resolution, cropping, scaling) to meet the requirements. We are also working on doing the same for our web app, but this remains work in progress. One concern this introduces is whether reducing image quality will reduce our description accuracy, and so far, we have not seen any significant impacts. However, given our plans to transmit images at a higher frequency, we may need to further reduce our package sizes, which could ultimately impact accuracy.

Another technical challenge which we have been working on resolving is latency. Currently, when a user sends a request to our service, it requires ~3-5 seconds before receiving a response. We continue to explore techniques to attempt to reduce this latency, by simplifying the pipeline (i.e reduce the number of intermediate api calls), potentially caching previous data, and reducing package sizes. Overall, we believe this will be the main technical challenge which we will need to work on for the rest of the project.