Behavioural 2.0

[Frontend Tasks:](#_nv0eqdel1s9j)

[● Download feature:](#_s04ejtrmy7dz)

[● Subscription based UI:](#_a98xxhpno7e)

[● Integrated Google Drive and One Drive](#_6dx3bluip43i)

[● Fixed token refresh issue](#_if469ks1ooqy)

[Backend Tasks:](#_dxebblz4wefu)

[● Navigation Bot](#_uy7fqw8u1irc)

[● Redesigned ETL pipeline](#_4iv92q47qsa3)

[● Audio Summarization](#_4r2br1p1s9dq)

[● Video Summarization](#_agbkiuetuik9)

[Voice Agent Mode for Hands-Free Task Execution](#_vzfu9ler5rhp)

# Frontend Tasks:

## Download feature:

At Xnode, we had an option to copy LLMs' responses. It was good, but our users had complaints that formatting is getting changed, or images were being lost, or other issues while trying to copy and paste the agent’s responses. One of our agent heavily respond with mermaid diagrams, so it becomes difficult when dealing with this. This had created some problems for us. We had this discussion in one of our regular meetings, and I proposed adding a download button where we can copy the response into a document and keep the exact formatting. My manager was satisfied with this idea and asked someone to implement this feature.  
  
I do not have any high-priority tasks then, so I went on to take ownership of this feature. I have created a work item and started working on it. I googled for libraries that help me build this feature. I looked at html2pdf, jspdf, and html-to-image, but finalized and chose to render the chat content to canvas because it gave pixel-perfect results of the conten,t including styles. I also had the flexibility to handle multi-page content and dynamic resizing. As the mermaid diagrams sometimes are wider and sometimes completely vertical.  
  
It was well received by our users, especially for the documentation-heavy tasks. The quality of outputs matched to that of UI, which helped them to reduce editing efforts. Though it was a simple task, it became pretty helpful. I was appreciated for this task as I came forward to take this, took end-to-end ownershi,p and completed it.

## Subscription based UI:

As our application was growing, we needed to introduce subscription based plans. For this, we had implemented this in the backend and could successfully segregate users based on their plans, and they were authorized to only the eligible features. But it is all being handled in the backend and few unnecessary calls were being sent to the backend. It also does not provide a good user experience. Hence, we decided to implement it in the frontend and I was given the responsibility to implement dynamic UI restrictions based on subscription plans. My goal was to make sure users only saw the features they were allowed to use, and also improve performance by avoiding backend calls for restricted tools.  
  
I went through few articles in the google, collaborated with staff level engineers to take inputs from them and went onto use a centralized state store using NgRx. I stored the user’s subscription plans, and the features they were allowed. I fetched these details once during the login and cached them in the NgRx store. I used a custom Angular directive like \*appIfAllowed, which checked the plan from the store and it rendered, or disabled a particular feature. I also added a small lock icon and a tooltip that redirects users to subscription plans page.

## Integrated Google Drive and One Drive

## Fixed token refresh issue

# Backend Tasks:

## Navigation Bot

At Xnode, many users were getting confused about how to use different agents and workflows inside the application. They didn’t know which agent to use or where to start. To solve this problem, I worked on building a Navigation Bot that could guide them step-by-step based on their queries.

Instead of relying only on hardcoded logic or generic LLM answers, I used a Retrieval-Augmented Generation (RAG) approach. This helped the bot give responses that were more accurate and specific to our app. First, I collected internal documentation like agent descriptions, workflow steps, and feature usage guides. I cleaned and split the data into chunks and generated vector embeddings using sentence-transformers like all-MiniLM-L6-v2. These embeddings were stored in a vector store — we used FAISS during development and later switched to Chroma for better control.

When the user asked a question like “How to upload a file and summarize it?”, the bot would convert the query into an embedding, perform a similarity search in the vector DB, and fetch the most relevant chunks. These chunks were passed to an LLM like GPT-4 with a carefully designed prompt, which generated a final answer with real agent names and exact UI steps.

We built the backend using Python and FastAPI, and the frontend in Angular with a chat-style interface. This setup helped us keep the responses grounded, fast, and scalable. We also added metadata like agent names and last updated timestamps to help with version control and filtering. To measure performance, we tracked metrics like intent coverage, response correctness, and user feedback scores.

Overall, using RAG helped us build a smart and helpful assistant that made the app easier to use, reduced support requests, and improved the overall user experience.

## Redesigned ETL pipeline

At Xnode, I was once doing a code review of a component where it handles summarization of uploaded files. I observed that the feature handles multiple files like docs, pdf, Images etc. But the code was written on condition based using a switch ladder. I felt it was not a correct design as it’ll be difficult to debug the code, difficult to scale and not the best practices. I took this to my manager and explained the severity of this issue. I explained that refactoring it would fix the existing bugs in the feature and also reduce the debuggin time if in case any bugs come up in this feature. Despite tight schedule, he gave thumbs up and I took ownership of redesigning it. I collaborated with the original developers and studied the logic. Then I understood that it basically contains three steps, ETL (Extracting, Transforming, and Loading) architecture—extracting raw text, cleaning and processing it, and finally summarizing it. So I made it modular so that it would become easily understandable and scalable. I used the design pattern - strategy (Factory, SRP also).. Since it has become modular, adding new file types to the feature would become easy just by writing 3 classes for Extraction, Transformation and Load. Later 3 more file types functionality was easily added using this approach. I got appreciation for turning that messy code into a modular and clean architecture.

## Audio Summarization

**S – Situation:**

At Xnode, we had a feature that allowed users to summarize long audio files, like meetings or lectures. Initially, we used OpenAI’s Whisper API to transcribe audio into text before passing it to our text summarization model. It was working well but turned out to be very expensive as Whisper API pricing increased with file length.

**T – Task:**

I wanted to reduce our dependency on external APIs and make this feature more cost-efficient without compromising on quality or speed. The goal was to find an alternative transcription model that could run locally and still maintain good accuracy.

**A – Action:**

I explored open-source alternatives and found a light version of whisper-small.en on Hugging Face. I deployed it using **TorchServe** on our internal servers. I built a modular Python pipeline:

* Used ffmpeg to extract audio from video files (if needed).
* Ran the audio through Hugging Face’s Whisper model to generate text.
* Sent the text to our existing text summarization pipeline.

For testing, I used **Word Error Rate (WER)** to compare transcription quality, and **ROUGE scores** to evaluate summary relevance. I also created edge test cases—like noisy audio, accents, and silent gaps—to validate robustness.

**R – Result:**

This new setup reduced inference cost by over **70%**, and transcription time improved by nearly **30%**. The accuracy was still close to what Whisper API gave. The feature became much more scalable, and the team appreciated the clean integration and cost savings.

## Video Summarization

**S – Situation:**

Many Xnode users started uploading video files for summarization, expecting quick insights. But our system could only summarize text or audio. We needed a smarter way to handle videos—especially non-narrative ones with diagrams, visuals, or slide transitions.

**T – Task:**

My goal was to design a video summarization module that considers **both** audio and visual content—so even if there is no clear narration, the summary is still meaningful. I wanted to build a **multimodal pipeline** that uses both frame-level image context and speech content.

**A – Action:**

I studied how **multimodal LLMs** like Flamingo, GPT-4o, and Gemini work. Based on that research, I created a local pipeline:

* Used ffmpeg to extract video frames every **10 seconds**.
* Used **CLIP** model from Hugging Face to caption each frame.
* Transcribed the audio using the same Whisper model as earlier.
* Used **SentenceTransformers** to convert both text and image captions into vector embeddings.
* Fused the embeddings and sent them to a lightweight local LLM (like **Mistral-7B** or **LLaMA** with LoRA fine-tuning) to generate final summaries.

I also introduced contextual scoring to prioritize key frames and remove redundancy. I tested the system using **frame coverage**, **summary relevance score**, and **manual coherence review**.

**R – Result:**

The system worked well and could summarize product demo videos, lectures, and interviews with great accuracy. The summaries were 40–50% more informative than simple audio transcription-based summaries. The team appreciated this feature as it opened the door for **chapter-wise summaries**, **timestamp generation**, and even AI-assisted video previews.

## Voice Agent Mode for Hands-Free Task Execution

SITUATION:

At Xnode, we already had an AI agent capable of executing internal tasks like sending messages, creating files, or fetching documents. However, during internal UX reviews, some team members highlighted that for multitasking professionals—especially those using the app in meetings or while working with hands occupied—a hands-free mode would significantly enhance accessibility and ease of use.

**TASK:**

My goal was to build a **Voice Agent Mode** on top of the existing command-execution agent. The voice agent needed to be triggered via a button or a wake word, listen to the user’s command, convert speech to text, pass it to the existing backend agent, and finally **speak back the response** in natural language. The voice interaction needed to feel natural, responsive, and secure—while also handling background tasks reliably.

**ACTION:**

I broke the problem into clear steps:

1. **Wake-Up Trigger:** I implemented two options—manual button trigger and wake-word detection using Porcupine by Picovoice.
2. **Speech-to-Text:** Used OpenAI’s Whisper model (initially via API, later self-hosted version for cost control) to transcribe the command. I added retry logic and confidence scoring to improve accuracy.
3. **Command Routing:** Parsed the transcribed text and passed it to the existing agent using the same payload format as the UI-based inputs. Reused the execution flow to avoid duplication.
4. **Response Narration:** For reading the response aloud, I integrated gTTS (Google Text-to-Speech) and later tested Coqui and Bark for better naturalness. I cached common responses to reduce latency.
5. **Frontend Integration:** Built a floating voice widget using React and Tailwind CSS that shows waveform animations while recording and has minimal interference with existing UI.
6. **State Handling:** Used Redux to maintain the voice agent state (e.g., listening, processing, speaking), and ensured fallback to manual input in case of failure.
7. **Security:** Ensured that the voice mode respected existing role-based access and command eligibility—so that the agent wouldn’t perform unauthorized tasks due to accidental voice commands.

**RESULT:**

The feature was released in internal beta, and users appreciated the hands-free interaction, especially for quick actions like **“Send message to Rahul,” “Create meeting notes,” “Summarise this document,” and “Email me the task list.”** The response time was kept under 3 seconds in most cases. This project was seen as an important step toward building a **conversational AI assistant** within our platform, and the team plans to extend it with multi-turn conversations and confirmation flows in the future.