[1. Made a mistake ( Root Cause - 2, or Improved performance )](#_61unocrq8v25)

[Web sockets issue with Stockfish evaluation](#_t3zs1iunjbcn)

[2. Exceeded expectations (Complex Problem or Improved performance)](#_s6rwolorn1mz)

[Agents were not working](#_h5c6ntkqmyci)

[3. Meet Deadline - 1](#_o2ep37x4inut)

[Message Service subscriptions are duplicating](#_42dyp6x859q)

[4. Dive deep for a Complex problem ( Root Cause - 1)](#_ordo9fe1q63t)

[Inconsistent Token expiry](#_uado07iryayc)

[5. Convince the Manager and Team members](#_egzdmxt5jkak)

[Deepseek, Multi-LLM integration architecture](#_ds78bf5y3b0i)

[6. Convinced Manager -2, Improved Performance, Invent & simplify](#_4c4kwgouzbwr)

[ETL Design](#_ws7jp7wrev44)

[7. Outside comfort Zone or Customer Feedback](#_ci2210rwwn1o)

[Navigation Bot](#_kq2rjphubo2)

[8. Made more efficient or Improved performance](#_bfrimedfcka4)

[Designed Rate limiter](#_ente0xy9t8r9)

[9. Improved Performance](#_r9sz8t81fajh)

[Scenario: Securing API Access with RBAC and HMAC in IwayDispatch](#_1hbm00vzzbra)

[10. Improved system](#_dzv9o72vbuyu)

[Scenario: Ensuring Reliable Notification Delivery in BidGo Global](#_bz7u0j6gv4h)

[11. Outside of your responsibility (Instincts, Tight Deadline - 2, Managed tight schedule )](#_v29ux1ks19lj)

[Investor Data Web Scraping](#_9h1qtxx7l5td)

[12. Critical Feedback from Manager](#_hdfox346tjfj)

[Communicating with others](#_e5mnm818pve3)

What made you think this is the solution?

Did you discuss it with anyone else?

How did you figure out the exact issue?  
How did you figure out that this is the right tool to use ?

Issue detection, root cause (how did you find), what is probable fix, how did you communicate this tot he rest of the team and what was their feedback? Implementation, Note on how do you make sure that this wont happened again (Creating a doc or keep note about this issue and how to prevent these kind of issues in future) , What did you learn and what experience did you gain from this ?

## Made a mistake ( Root Cause - 2, or Improved performance )

### Web sockets issue with Stockfish evaluation

**Tell me about a time where you felt you made a mistake, what was the mistake, and how did you resolve it?**

**Situation:**

I was building a full-stack chess web application where I had integrated the Stockfish chess engine in the backend. My goal was to give users live feedback through a prediction bar that visually shows who is in a better position—white or black—after every move. For this, I used WebSockets with Socket.IO to send the FEN string from the frontend to the backend every time the board changed. This included actual moves, but also small interactions like dragging a piece or hovering. The idea was to make the experience more interactive and educational for users.

But when I started testing this setup during actual gameplay, I started noticing issues. The prediction bar was sometimes resetting to a neutral 50-50 state randomly, even when a player had a clear advantage. Worse, the backend started crashing frequently. So it became clear that something was going wrong under the hood.

**Task:**

I had to ensure that after every valid move, the board position gets evaluated by Stockfish and the prediction bar is updated accurately and consistently. It had to work smoothly across devices and not crash the backend. From a user perspective, even one wrong jump in the prediction bar would hurt the experience. I also had to be careful about not overloading the system because the backend was deployed on a basic AWS EC2 instance. So, my task was to make the evaluation reliable, stable, and error-free—without changing the quality or user experience.

**Action:**

Initially, I thought the issue was with the Stockfish engine itself. But when I started investigating more deeply, and checking the socket.io logs, I found that the real issue was that even before the stockfish engine process a request, it is being hit with few other requests which I was not aware that stockfish does not handle this situation, and causing a Race condition. So, the problem here exactly is how often and how quickly the frontend was sending FEN strings to the backend. Because I was using WebSockets, and because React triggers re-renders frequently, the frontend was emitting multiple FENs for the same position. These were overlapping, and the backend was receiving them back-to-back without any control. This caused race conditions—where two or more evaluations would run at the same time and interfere with each other.

To confirm this, I debugged the React useEffect() hooks and saw that I wasn’t cleaning up the socket listeners using socket.off(). That meant multiple listeners were active at the same time, and all of them were sending requests to Stockfish. This made the prediction bar behave unpredictably, and caused the backend to crash often.

To fix this, I had to put so many conditions if I wanted to use Socket.io, so instead I decided to drop the socket-based approach completely and switch to controlled polling. In this new approach, I only send the FEN string from the frontend once, right after a valid move is made. There’s no emission during dragging or hovering—only after the move is confirmed. On the backend, I also used a safer design. I wrapped the Stockfish engine in a block in such a way that the engine would cleanly start and shut down for each request. Since requests were now happening only once per move, there was no overlap, no race condition, and the engine could process each board state properly without getting confused.

But I didn’t stop there. I realised that even with polling, there could be situations where two requests might still come back-to-back—for example, if someone clicked rapidly or the frontend re-sent due to latency. So I added an extra safety layer by introducing a **processing flag** on both the frontend and backend.

On the **backend**, I used a simple Python flag to mark if an evaluation is in progress. If a new request comes while one is already running, I return a message like “Engine busy, try again later.” This prevents the engine from being invoked twice at the same time.

On the **frontend**, I added a state variable called isEvaluating. Before sending a request, I check if this flag is true. If yes, I hold off. Once the evaluation is done and the result is received, I reset the flag. This way, I make sure the frontend never sends multiple evaluations for the same move.

I also made sure to clean up any existing listeners in React using socket.off() for future safety, even though I had moved away from sockets.

**Result:**

After these changes, the prediction bar became stable and accurate. It updated correctly after every move without jumping around or resetting. The backend also stopped crashing because now Stockfish was getting requests only when needed, and only one at a time. The frontend was lighter and smoother, and I found the UX consistent and helpful.I made sure that I understand the request handling capacity, and rate limiting methods, any engines or models that I’m going to use in the future. Also, real-time is not always the best design choice, especially when it leads to instability. Sometimes, controlled updates at the right moment are more powerful. It also showed me how important it is to coordinate the frontend and backend correctly—understanding how communication patterns, concurrency, and evaluation logic work together. Going forward, I’ve become more careful about designing systems that are not just functional but also robust and scalable from day one.

**Key Concepts:**

**React Hooks:** Functions like useEffect() are used in React to run side effects (like API calls or subscriptions).

**Race Condition:** A bug that occurs when two or more operations run at the same time and interfere with each other.

**Long-lived Process:** A backend service or engine that stays running for a long time, ideally managing multiple requests, but can become unstable if misused.

**Socket.IO:** A library for real-time communication between frontend and backend using WebSockets.

**FEN (Forsyth-Edwards Notation):** A standard notation to represent the current position on a chessboard.

**Controlled Polling:** Sending requests at specific times or events instead of continuously or in real-time.

## Exceeded expectations (Complex Problem or Improved performance)

### **Agents were not working**

**Tell me about a time you exceeded expectations.**

In one of my recent personal projects, I was building a voice-activated inventory management system for local kirana shopkeepers. The main idea was to create something like a smart assistant, where the shopkeeper could just say things like “Add 5 kg rice” or “Remove 2 litres milk,” and the system would handle the stock automatically. Since many shopkeepers are not very comfortable with typing or using modern apps, I wanted to make an attempt to make it fully voice-based and very easy to use.

My task was to integrate an AI agent that could understand natural voice or text commands, convert them into proper operations like add, update, delete, or list, and then perform those actions — but only after confirming with the user. I integrated the LLM, gave correct instructions, and I thought the agent should work perfectly. Initially, it worked well for many of the instructions, but when I went on testing it, the agent misunderstood partial instructions like “update rice 5 kg”, it didn’t ask for confirmation before changing data, and sometimes even tried to list items when the command was actually invalid.

I went through multiple tutorials on YouTube to understand and learn a better way of creating AI agents. In one of the videos, the programmer created three agents, team member1, team member2, and a team lead agent that handles those two members. That’s a concept of multi-agent coordination. So, taking that as a reference, I thought of trying a solution ie., I decided to change the design completely. Like in the above approach, I did the same here. I split it into two agents. The first one, called the **CommandParserAgent (team member)**, was only responsible for understanding the user’s voice and converting it into a proper structured format like JSON. For example, it would give something like: operation is “add”, item is “sugar”, quantity is 5, and unit is “kg”. I gave this agent strict instructions — what kind of words to look for, how to ignore fillers like “the” or “a”, and how to extract important values. This made debugging much easier — if something went wrong, I could immediately check what exactly the model understood.

The second agent, which I called the **ExecutionAgentn (Team lead)**, handled all the backend logic. If the operation was just to “list” items, it would do it immediately. But for other commands like “add”, “update”, or “delete”, it would first store that request and then ask the user for confirmation like: “Do you want to add 5 kg sugar?” Only if the user said yes, then it would actually perform that operation. This way, there were no accidental changes to the inventory. I also added fallback conditions — for example, if the user said “yes” followed by an unrelated command like “yes remove” for add operation, it would repeat the command and ask for confirmation again. If the voice input was completely unrelated, it would simply reply “Invalid command” instead of trying to guess.

After this change, everything became much smoother and reliable. The parsing agent became very accurate, and because both agents had separate responsibilities, I could test them individually. I even had the flexibility to switch models later — like using OpenAI’s Whisper for transcription, Gemma 2.0 or ChatGPT for CommandParsingAgent. The whole system started feeling more natural — like how Alexa or Google Home works, always asking before doing anything.

More than that, it was a great learning experience in how to properly design AI systems. I realised that just using a powerful model is not enough — it’s how you guide it, split its responsibilities, and structure the logic that makes the real difference. It made me think more like a system designer than just a coder, and the final result was a very dependable and modular setup — something that can scale and improve over time.

## Meet Deadline - 1

### **Message Service subscriptions are duplicating**

**“Tell me about a time when you didn’t think you’d meet the deadline, and how did you handle it?”**

**Situation:**

At Xnode, I was working on a resource handling component where I was integrating Google Drive and One drive into our application, what this component does is, the users can upload any files like a pdf, csv, doc, an image, video or any kind of file. The component theen sends the file to the backend for ETL processing which usually takes sometime depending on the kind and size of file. After the process gets, done, the backend sends a message to an other component which handles the app notifications. The notifications component in-turn sends a message to the resources component which indicates that the component has some updates and it needs to refetch the resources list. So this component handles multiple things like fetching the resources and mapping them to respective folders and also contains multiple tabs. So one of the tab in there is RECENT RESOURCES. When this component receives the message, it will fetch the updated resources, do the resource-folder mapping and refresh the list.So, a demo was scheduled with our clients in a day and I noticed a serious issue. Every time I navigated back to the RECENT Resources tab, it started to show the duplicate entries of the recently uploaded files. The more I switched tabs, the more duplicates I saw. The screen looked broken and unreliable, and the demo was like less than 24 hours away.

**Task:**

I worked on building different features in this component and I was responsible for making sure the feature worked properly for the demo. So this feature was developed by my team mate from India whom I cannot connect to and discuss in his off hours. I checked from the activity of the work item that a proper unit testing of the feature was done but integration testing was missing due to a tight deadline. At that point, I didn’t know exactly what was causing the duplication issue, and there was not enough time to fully investigate and fix the root problem before the client presentation. But I had to somehow **make the UI stable enough to avoid any problems during the demo**, while also continuing to look into the actual issue and come up with a permanent solution.

**Action:**

I discussed about this issue with my manager and while discussing with him I could come up with a hotfix that instinctively striked in my mind for the issue which also does not effect any other functionalities. He agreed to it as it is one of the safest approach to go with considering the deadline. So the solution was to implement an additional filter that removes duplicates from the file list just before showing it in the UI. I used the file IDs to filter out repeated entries. This made sure that even if the method fetchResources() which fetched resources, was called multiple times and a resource-folder mapping is done multiple times, the user would only see clean, unique files on the screen. This gave us a short-term solution that would hold up during the demo.

**Result:**

The temporary hotfix worked smoothly for the demo, and our clients were very happy with how the demo went smoothly. After the demo, I shifted focus to debugging the real issue. After going through the code thoroughly, and testing different scenarios, and going through logs(did not find any, logs were clean), I found that the root problem was due to **repeated subscriptions** to the messaging service. I studied thoroughly about this issue from various articles and used GPT to understand more and learnt that this is one of the common issues that occurs due to the non-Idempotent nature of the PUB/SUB mechanisms. Later, I deployed the proper fix, which completely eliminated the duplication issue, and made the user interface more stable and clean. This experience taught me that sometimes we need to act quickly under pressure, but also take full **ownership** to come back and fix things the right way. It also reminded me that in real-time applications, **managing subscriptions properly** is critical to avoid memory leaks(which will have serious consequences if ignored) and performance issues. My manager and team appreciated both the short-term thinking for the demo and the long-term commitment to fixing the root cause properly.  
  
ACTUAL FIX:  
In the component, various subscriptions to different messages were made inside the constructor of the component, but we were not unsubscribing when the component was destroyed and recreated—which happened every time the user switched tabs. The message retained in the component and not removed So, each time the Resources tab was opened again, a new subscription was added, and none of the older ones were removed. This caused it to trigger fetchResources() method multiple times.

As I said this is due to the non idempotent nature of the PUB/SUB messaging queues and the developers need to add additional logic to handle this properly. So to fix this properly, I used a rxjs operator (an Angular library) - used **takeUntil()** method and a destroy$ subject (notifier)(RxJS tools used to unsubscribe from ongoing listeners or data streams to prevent memory leaks and duplicate processing). I added logic inside ngOnDestroy() life cycle hook to emit a signal that told all active subscriptions to stop. This way, when the component was destroyed, all existing listeners were also cleaned up. As a result, when the tab was reopened, only one clean subscription was created again.

**Idempotency** means performing the same operation multiple times has the same effect as doing it once.

**fetchResources()** – A function that retrieves the list of uploaded files from the backend server.

**onMessage()** – A real-time event listener that gets triggered when the backend sends a notification.

**ngOnInit() / ngOnDestroy()** – Lifecycle methods in Angular; ngOnInit() runs when the component is initialized, and ngOnDestroy() runs just before the component is removed, used to clean up resources.

**takeUntil() / destroy$** – RxJS tools used to unsubscribe from ongoing listeners or data streams to prevent memory leaks and duplicate processing.

**Subscription** – A connection to a stream of data or events, which must be cleaned up when no longer needed.

## Dive deep for a Complex problem ( Root Cause - 1)

### **Inconsistent Token expiry**

**Situation:**

One day at Xnode, I was going through customer feedback and session logs, there I personally noticed a strange issue. Users were getting logged out unexpectedly, even when they were actively using the application and they could not mention any steps to recreate as it looked very random. Some got logged out in just 10 minutes, while others stayed logged in for hours. The pattern was inconsistent, and surprisingly, this problem had not been caught during QA testing. I decided to take ownership of this issue because it was directly affecting the user experience and trust in our platform.

**Task:**

To get to the root of the issue, I connected with QA team to try and recreate this issue. I took their help and created different test scenarios—like using logging in and staying idle, or being active, logged in and logged out multiple ways and multiple times and using multiple tabs, and observed the logs in each case. I added detailed logs to track when the access and refresh tokens were being generated, refreshed, and cached in Redis where we store the refresh token. After diving deep, I found the root cause: that when a user opened multiple tabs, in some cases, all of them were trying to refresh the token at the same time or in very short intervals. Since every new token invalidates the old one, only one tab would succeed, and the others would silently fail—causing users to be logged out due to invalid tokens. I thought that there is a need to implement any debouncing mechanism to control this. In the next meet, I discussed about this issue and explained about the root cause with my manager and team. Even few of our teammates conveyed that they also felt the application was logging out sometimes abruptly and overlooked this issue.

**Action:**

To fix this, I have gone through multiple articles over the web and noticed that it is one of the common issue in applications that use **token-based authentication**—especially **single sign-on (SSO)** or **JWT (JSON Web Tokens)**. I learnt that there are multiple ways to fix this issue such as using shared storage(local or session storage), or centralized token refreshing(using Broadcast API to notify all tabs), and using Redis locks. I implemented atomic token refresh logic using Redis locks. This made sure that only one refresh request was allowed per user at a time and the refreshed token is sent to all tabs using Redis PUB/SUB mechanism. So, only the first tab can request and the rest all tabs will have a new token set and they do not request a new one. I also added a debounce mechanism to group multiple quick refresh attempts into one. Additionally, I set up telemetry (OpenTelemetry, commonly used for monitoring logs in Nodejs) to monitor how often refreshes were succeeding or failing, to catch similar issues early in the future.

**Result:**

After the fix, the logout problem was solved, and session stability improved a lot. Support tickets related to session issues dropped by over 90%, and users were able to use multiple tabs smoothly. This solution became part of our core authentication module and was reused by other teams too, for different portals. I also received appreciation from my team and manager for taking initiative and solving a long-standing overlooked issue. This experience really taught me very much and gave me more confidence about designing user authentication pages, and also learnt how Redis cache can be used for multipurpose things, learnt that Redis can be very handy in many cases.

-> **How did you combine multiple requests into one (Using redis Pub/Sub)**  
- First lets say, the first tab sends a request to refresh token. Redis then keeps a lock to Refresh token and any other requests from other tabs sees that token is locked and wait for it to be removed. Meanwhile, Redis generates new active token, stores it somewhere and sends it to the first tab which requested it and removes the lock. All waiting requests gets the same updated token and tabs gets refreshed.

Using Redis cache for atomic token refresh logic is a great way to ensure efficient, scalable, and secure authentication. Redis provides fast in-memory storage, making it ideal for managing token expiration and refresh operations.

**How Atomic Token Refresh Works with Redis  
Atomic Token Refreshing:** An "atomic token refresh" refers to the process of obtaining a new access token using a refresh token in a single, indivisible operation, ensuring that the entire refresh process is either completely successful or completely failed, and that the system is not left in an inconsistent state. This concept is particularly important in scenarios where the refresh token is the only mechanism for access renewal, and where errors during the refresh process could lead to a user being logged out or facing access issues.

1. Store Refresh Tokens in Redis

- When a user logs in, generate a refresh token and store it in Redis with an expiration time.

| redis.set(`refresh\_token:${userId}`, refreshToken, 'EX', 86400); // Expires in 24 hours |
| --- |

2. Atomic Token Refresh Using Redis Transactions

- Use Redis transactions (`MULTI` and `EXEC`) to ensure atomicity.

- When refreshing a token:

- Validate the existing refresh token.

- Generate a new token.

- Replace the old token atomically.

| redis.watch(`refresh\_token:${userId}`); redis.multi()  .set(`refresh\_token:${userId}`, newRefreshToken, 'EX', 86400)  .exec(); |
| --- |

3. Prevent Race Conditions with `SETNX`

- Use `SETNX` (Set if Not Exists) to prevent multiple tabs or requests from refreshing the token simultaneously.

| if (redis.setnx(`refresh\_lock:${userId}`, 'locked')) {  redis.expire(`refresh\_lock:${userId}`, 5); // Lock expires in 5 seconds  // Proceed with token refresh  } |
| --- |

4. Token Rotation for Security

- Implement refresh token rotation to invalidate old tokens when issuing new ones.

- Store a list of valid tokens in Redis and remove expired ones.

5. Broadcast Token Updates Across Tabs

- Use Redis Pub/Sub to notify all active sessions when a new token is issued.

| redis.publish('token\_refresh', JSON.stringify({ userId, newToken })); |
| --- |

## Convince the Manager and Team members

### Deepseek, Multi-LLM integration architecture

**Tell me about a time you had to convince team members on something you proposed.**

**Situation :**

Just a couple of days after DeepSeek R1 was launched, I was part of a team meeting at Xnode where we discussed improving our GenAI workflow. During that meeting, I proposed that we should start exploring the integration of multiple LLMs— The discussion started with DeepSeek but later I continued that not just DeepSeek, but also integrating other emerging models like Claude, Mistral, Gemini, and similar ones will be a very good addition to our application. I explained that different LLMs are specialized for different tasks—for example, some models generate better creative content, some are cheaper to use, and others are more accurate for summarization or rewriting. I strongly felt that if our application supported multiple LLMs, users could get the flexibility to choose what works best for their use case, and it would give us an advantage in the GenAI space.

**Task:**

There was some hesitation from the team and my manager, mainly because of concerns around infrastructure complexity and whether some of these models were production-ready. I took the initiative to address these concerns. I gathered technical documentation and presented the benchmark scores a few of these models. I used various sources like Open LLM Leaderboard by Hugging Face, [epoch.ai](http://epoch.ai), [livebench.ai](http://livebench.ai), Chatbot Arena etc... I analyzed their response times, cost implications, and behavior with different prompt types, and different capabilities. I also focused on the data privacy aspect—especially for enterprise users—by showing how self-hosted models like DeepSeek could give us full control over customer data.

**Action:**

To make the case stronger, I designed a clean and modular architecture that could support multiple LLMs in a plug-and-play fashion- just switch. I structured it in a way that each model integration would follow a standard interface so that it becomes easy to route between different agents to different models. I also included a fallback mechanism in the architecture so that in case any model failed or hit rate limits, the system could automatically switch to another one. I believed this would improve system reliability, scalability, and flexibility. I personally reached out to teammates who worked on the current agent infrastructure to understand how the integration points would fit, and I ensured that the proposal aligned with our long-term goals.

**Result:**

Finally, I submitted a proper all the benchmark details in a presentation and boosted my proposal by submitting the **multi-LLM architecture** document, I prepared, to my manager, clearly outlining how the integration could be done step by step, and how and where we can introduce in the UI. I proposed starting with DeepSeek and expanding to other models gradually later, one by one. This was a realistic plan that gave us the ability to support different GenAI use cases more effectively and position ourselves as a flexible, forward-looking platform. My manager and team convinced with my proposal and finally it was integrated later. This experience helped me understand the value of backing ideas with data and how a clear technical vision can lead to real product impact.

-> what did you benchmark it against?  
Benchmarked in different capabilities like Reasoning, Math, Problem solving, Data Anslysis, Language understanding etc..

## Convinced Manager -2, Improved Performance, Invent & simplify

### ETL Design

**Situation**

I was once assigned a bug in the summarization feature of our Xnode application. This feature allowed users to upload documents like PDFs and get summaries or do knowledge search on them. When I opened the codebase to fix the issue, I was honestly shocked. Everything—reading the file, extracting the text, cleaning it, summarizing, and even displaying the output—was written inside one huge function. When I looked into the summarization feature code, I noticed that everything — from reading the file, to extracting the text, cleaning it, summarizing it, and finally displaying the result — was written inside one large function. This kind of design created several problems:

First, it violated the Single Responsibility Principle, because the same function was handling multiple things. So, even for a small bug fix or a change in one step, I had to understand the entire function, which was very time-consuming.

Second, it was tightly coupled. None of the steps could be reused separately. For example, if I wanted to support Excel files or change the summarizer logic later, I would have to rewrite or copy large portions of the function again, which could lead to duplication and bugs.

Third, it was hard to test and debug. Since everything was mixed together, if something broke, it was difficult to isolate the issue — whether it was in file reading, cleaning, or summarizing. Writing unit tests also wasn’t possible for individual steps.

Lastly, it was not scalable. As the feature grows and more file types or AI models are added, this kind of monolithic structure would become unmanageable.

And when I checked with a teammate who had worked on a similar CSV feature earlier, they said they had to duplicate logic because the original code wasn’t reusable. That’s when I proposed refactoring the entire logic into a proper ETL-based modular structure, so each layer handles just one job and the components stay reusable, testable, and maintainable.It had no structure, no separation of responsibilities. It was like reading a jungle of logic. It took me almost half a day just to understand what the function was doing, and even after that, fixing one bug felt like pulling one wire from a big knot. That’s when I realized—this is not just a bug, this is a design issue.

**Task**

The next day in our stand-up, I brought up this issue. I said, “Yes, I fixed the bug, but this kind of code will keep breaking, we might get recurring issues. If we want this feature to scale, we need to redesign it properly.” Initially, there was silence. Everyone was busy with their own sprint goals. My manager appreciated my point but said this work had been sitting in the backlog for months and nobody had picked it because of time constraints. I took that as a challenge. I requested him to let me take up a few related bugs and instead focus my efforts on refactoring and redesigning this module. I explained how important this change was for long-term maintainability and that future developers shouldn’t go through the same pain I faced. After some discussion, my manager agreed but gave one condition—I had to reuse the existing logic wherever possible and avoid rewriting everything from scratch.

**Action**

Before jumping into coding, I did my homework. I spoke to the India team who had written the initial version, just to understand why it was designed that way. I even scheduled a few short meetings with teammates who had worked on other file processing features. After I understood the system well, I went ahead and designed a proper architecture based on ETL—Extract, Transform, Load. I followed the Single Responsibility Principle — each layer has one job.

* The Extract layer focused only on reading raw content from uploaded files.
* The Transform layer was responsible for cleaning and preparing the text to make it ready for summarization.
* The Load layer passed this clean text to the summarizer engine and prepared the output for UI display.

I split these into separate modules, and I designed it in such a way that if tomorrow we want to support new file types like CSV or Excel, it would be as simple as writing just three new classes—one for each stage. No one would have to touch the existing code. As I said, I followed the Single Responsibility Principle by separating each part of the summarization flow into extract, transform, and load layers. I also made sure the design was decoupled — meaning each module doesn’t depend tightly on the others. For example, I can change the extractor logic or the summarizer logic independently. This makes the system highly maintainable and scalable.

I tested everything thoroughly, covered multiple file types and edge cases, and finally pushed the code to the dev branch. I also informed our QA team in advance and explained the new structure so they could verify it smoothly.

**Result**:

After deployment, this new ETL-based architecture became one of the **most reusable and scalable** parts of our application. One of my teammates was later assigned the task of adding support for Excel and CSV files, and he came back and told me, “Bro, it was so easy. I just had to create three small files. It took me hardly any time.” That one line made all my effort feel worth it.

This task was not originally in my scope. But I took full ownership of it because I strongly believed that **band-aid fixes are not enough—sometimes, we need to solve the root design problem**. This experience taught me that if you explain things clearly, back it up with logic, and show how it’ll help the team, you can convince your manager, your teammates, and make a real impact. And honestly, this also boosted my confidence in designing modular systems that are built to scale.

-> Need more detailed explanation regarding how the code in broken down into a more modular approach.

## Outside comfort Zone or Customer Feedback

### Navigation Bot

**Tell me a situation where you took up something outside your comfort area and built expertise.**

**Situation:**

In one of our stand-up meetings at Xnode, our founder shared some feedback from a customer. They said they were finding it difficult to understand the various workflows and AI agents in our application. Since our platform had many features and different agents for different use cases, and as it is single screen layout, it was getting confusing for first-time users. They were not sure when to use which agent or how to explore different modules.

That feedback stayed in my mind. I thought, instead of giving users a long document or video walkthrough, why not build an AI agent that could act itself act as a guide? Something that could explain features, answer doubts about workflows, and help users navigate the product better — like a virtual assistant inside the app. That’s where the idea of building a Navigation Agent came up, and I immediately proposed it to the team.

**Task:**

My goal was to design and implement this AI-based navigation agent from scratch. The challenge was — I had never built an agent before, and I wasn’t fully familiar with how our internal agent framework works. But I knew this was a great opportunity to learn something new and also solve a real customer problem. I wanted to make sure that the agent would answer only app-related queries and avoid anything irrelevant or out of scope.

**Action:**

I started with self-learning. I enrolled in a short deep learning course to understand how LLM-powered agents work. At the same time, I reached out to teammates who had already worked on our existing agents. They were very helpful and shared their design documents and code snippets, which gave me a clear idea of the overall structure.

Once I had a basic understanding, I had a detailed discussion with our product manager to gather relevant application data — like what each module does, what common user workflows look like, and what kinds of questions users typically ask.

Based on that, I designed the navigation agent to do three key things:

1. Answer questions based on actual product structure and data

2. Detect out-of-scope questions and avoid giving incorrect answers

3. Redirect users to other specific agents if their query matched a different use case

During development, I did face some challenges — especially around handling fallback logic and prompt conditioning. But I didn’t hesitate. Whenever I got stuck, I reached out to my teammates, clarified doubts, and moved forward. I also did a short prompt engineering course because I learnt that giving instructions to an agent is a very important task of designing it. I tested the agent thoroughly with various edge cases to ensure it gives accurate and safe responses. Once everything looked good, I pushed it to the dev environment.

**Result:**

The navigation agent was received very positively — both by our team and the customer. It provided an **interactive and friendly way to understand the product**, and helped new users get started without any confusion. It ended up becoming a very handy **onboarding tool** as well.

I also received appreciation from the team for coming up with the idea, taking ownership, and executing it well. Personally, it gave me a lot of confidence. I realized that even if something is completely new, if we are curious and committed to learning, we can figure it out. And it is always important that whenever I’m building a complex application, I have to look at it in a customer POV and make sure they understand the flow easily or atleast provide an additional way or additional material or an agent like in this case to make them understand the application better, only then they can make full use of the application and give value to it. And this experience really taught me that **some of the best solutions come when we listen to feedback, think creatively, and take ownership** without waiting for someone to assign the task.

## Made more efficient or Improved performance

### Designed Rate limiter

**Tell me about a time when you improved a process or made something more efficient**

**Situation:**

In the Xnode application, users interact with multiple AI agents powered by LLMs like ChatGPT. Since these model calls are expensive and go through third-party APIs, we had implemented a static rate limiter to protect our usage. But over time, we started receiving complaints — especially from enterprise users who use the system heavily during peak hours. They were getting hit with 429: Too Many Requests errors, even during their normal business workflows. This affected their experience and created frustration.

This issue was repeatedly discussed in our internal meetings — we all knew the static limiter was not scaling well, and it wasn’t fair to apply the same rule to everyone. We needed a smarter solution that could balance protecting our infra costs while still offering flexibility to our high-tier users.

**Task:**

I had to find a better way to manage usage so that we could protect our backend and LLM quota, while also making sure customers had a smooth experience—especially our premium users. I also had to think about future scalability and how this rate limiting could work not only for Xnode-provided models, but also in cases where users connect their own LLMs through BYOM (Bring Your Own Model).

**Action:**

I started with some research on modern rate limiting strategies — things like token bucket, sliding window, burst allowances, etc. After evaluating a few patterns, I decided to redesign the entire rate limiting system using Redis as the backbone, because it supports distributed counting and TTL-based expiry.

Here’s what I implemented:

* The rate limiter is now tier-aware — free users get basic limits, while enterprise users can burst temporarily beyond their base limit during peak hours.
* I stored rate usage and burst tokens in Redis, allowing fast lookups and updates.
* I also made sure that this same logic could gracefully support BYOM scenarios. Since BYOM users pay for their own usage, I applied only soft limits — mainly to avoid frontend overload and rate-based UI errors.
* Additionally, I exposed real-time usage stats in the dashboard, so users could monitor how close they were to hitting the limit. This gave transparency and also encouraged some to upgrade their plan voluntarily.
* Finally, I made the entire rate limiting logic configurable — meaning plan limits, cooldown intervals, and burst logic can be adjusted easily without code changes.

**Result:**

The results were very promising. Within weeks, rate-limit complaints dropped by over 60%, and our support team received fewer tickets related to 429 errors. Two enterprise clients even upgraded their plans after realizing they could scale more predictably. The system became fairer and more reliable — it didn’t treat all users the same, and gave higher flexibility to power users.

I received appreciation from my manager and teammates for taking the initiative and redesigning a long-standing problem that had been left open for quite some time. This experience taught me that sometimes the best way to fix a customer pain point is to look deeper into infra-level problems, and that smart engineering can directly impact business outcomes and customer satisfaction.

**mHow the Static Rate Limiter Was Originally Implemented**

**Why Use Redis for Rate Limiting?**

✅ **Distributed Across Multiple Servers** → Ensures consistent rate limiting in a **multi-instance architecture**. ✅ **Fast & Low Latency** → Redis operates in-memory **but is shared across instances**, making it nearly as fast as local memory. ✅ **Atomic Operations** → Prevents concurrency issues when multiple API servers need to track request counts. ✅ **Persistence & Auto-Reset** → TTL-based expiry ensures smooth resetting without manual intervention. ✅ **Easy to Scale** → Works across cloud deployments, making it ideal for high-traffic applications. Using locks, it can prevent Race condition.

**What the Rate Limiter Does**

The rate limiter controls the number of requests a user can make within a defined time period.

- It prevents system overload by ensuring fair access to resources.

- It allows bursts for enterprise users temporarily while enforcing long-term limits.

- It resets limits periodically so users can send fresh requests after the cooldown period.

Your implementation uses Redis to track and enforce these limits dynamically based on user tiers.

**How It Works (Step-by-Step)**

1️⃣ **Define Rate Limits Per User Tier**

- Free users: 100 requests per hour

- Enterprise users: 500 requests per hour + Temporary burst of 200 extra requests

2️⃣ **Track Requests Using Redis**

- A Redis key is assigned per user (e.g., `"user:12345:requests"`).

- Each request increments a counter stored in Redis.

3️⃣ **Check Limits Before Allowing Requests**

- If the user's request count is below their tier’s limit, they can proceed.

- If enterprise users haven't exceeded their burst threshold, they can exceed the standard limit temporarily.

4️⃣ **Apply Rate Limiting Logic in Code**

- Redis retrieves the request count.

- If within allowed limits, Redis increments the request count and allows the request.

- If exceeded, the request is blocked or delayed.

5️⃣ **Automatically Reset Counters Using TTL**

- The counter expires every hour, meaning the limit resets for the next cycle.

- Enterprise users can burst again once the TTL refreshes.

**Code Implementation in Redis**

| import redis # Connect to Redis redis\_client = redis.Redis(host='localhost', port=6379, db=0)  def is\_request\_allowed(user\_id, tier):  limits = {"free": 30, "enterprise": 500}  burst\_threshold = {"enterprise": 200} # Additional burst allowance  bucket\_key = f"user:{user\_id}:requests"   # Get current request count  current\_count = redis\_client.get(bucket\_key)  current\_count = int(current\_count) if current\_count else 0   # Allow temporary burst if under burst threshold  if tier == "enterprise" and current\_count < (limits[tier] + burst\_threshold[tier]):  redis\_client.incr(bucket\_key) # Increment request count  redis\_client.expire(bucket\_key, 3600) # Reset every hour  return True   return current\_count < limits[tier] ``` |
| --- |

**Key Advantages of Using Redis**

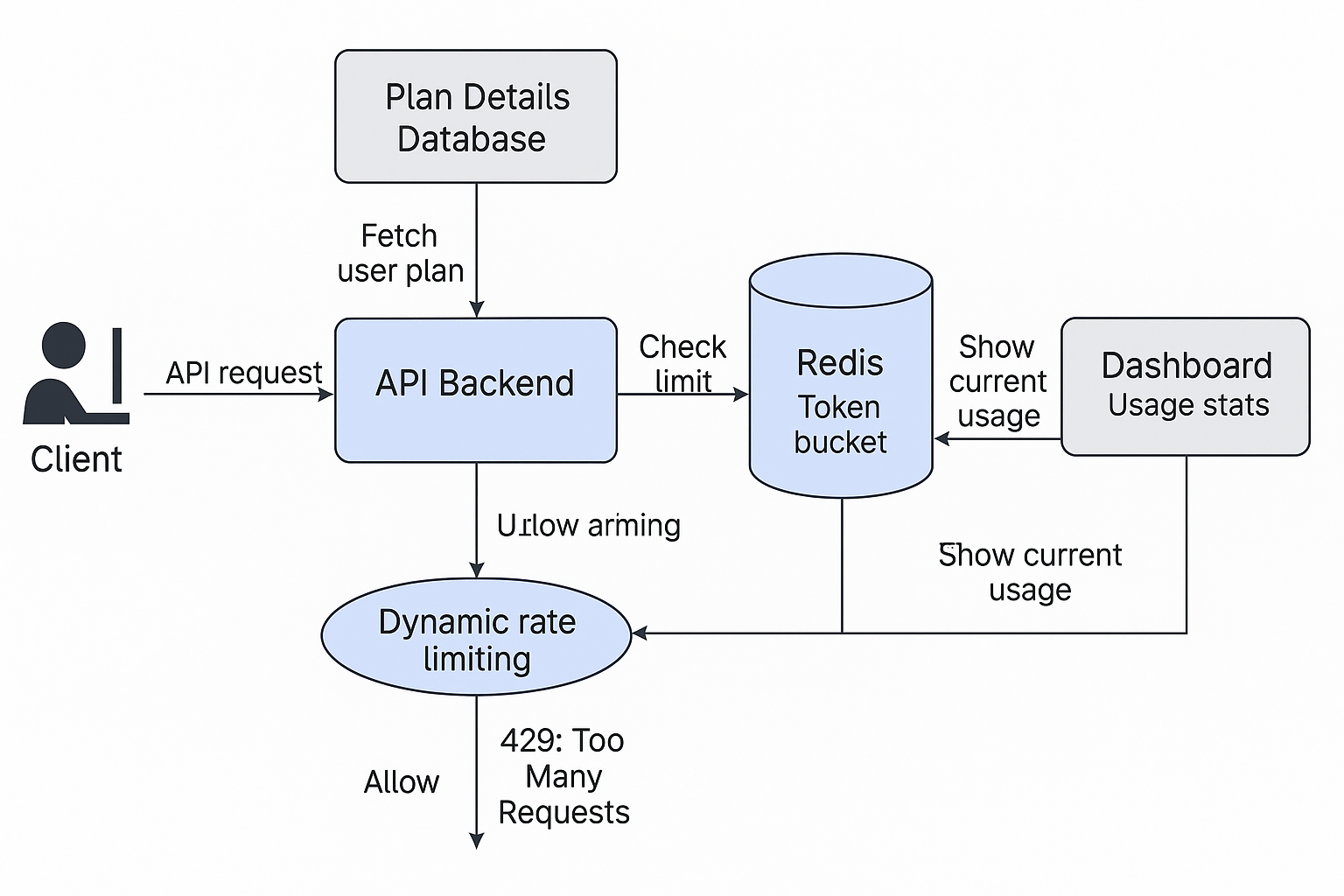
✅ Fast & efficient → Operates in-memory for low-latency enforcement.

✅ Scales across multiple servers → Ensures consistency in a distributed system.

✅ Automatically resets → Redis TTL removes expired keys for fresh request cycles.

✅ Supports adaptive logic → Enterprise users get temporary bursts dynamically.

**Simple** **architecture diagram** :



## Improved Performance

### **Scenario: Securing API Access with RBAC and HMAC in IwayDispatch**

**Leadership Principles:** *Insist on the Highest Standards*, *Ownership*, *Dive Deep*

**Situation:**

While working on the IwayDispatch platform, I was doing a routine backend code review before an upcoming internal security audit. We were preparing for a demo and also wanted to ensure things were tight before giving access to some external partners. As I was going through the authentication flow, something caught my eye — although we had JWT-based login in place, many critical API routes were only protected at the login level, not at the role level.

Anyone who had a valid token — even a basic user like a driver — could hit APIs meant for dispatchers or admins. This included highly sensitive information like live driver locations, load assignments, and fleet controls. That moment, I knew — this wasn’t a small bug, it was a serious security loophole that could easily lead to data leaks or misuse if not fixed in time.

**Task:**

I immediately decided to take ownership of the situation. My task was clear — I had to validate and secure all backend APIs. I had to make sure that:

* Every request was secure and authenticated
* The communication was encrypted
* And most importantly, that authorization was enforced based on roles — drivers, dispatchers, admins — each should only see what they’re supposed to.

We had only a few days before the audit, so I had to move fast, and move smart.

**Action:**

I started with the basics. First, I ensured all APIs were behind proper JWT validation. Then, I updated our Nginx config to enforce HTTPS strictly. I added SSL certificates via Let’s Encrypt to make sure every piece of data going in and out was encrypted.

But the bigger issue was authorization — or rather, the lack of it. We were only checking if the user was logged in, not whether they had permission to access that specific endpoint. This meant a driver could potentially access dispatcher dashboards, or even administrative APIs, just because they had a token. That was risky.

So, I took the initiative to design a full Role-Based Access Control (RBAC) system. During login, I made sure each user’s role was fetched and embedded into the JWT itself. Then, on the backend, I built a centralized middleware that checked every route — comparing the user’s role against the allowed roles for that endpoint.

For example:

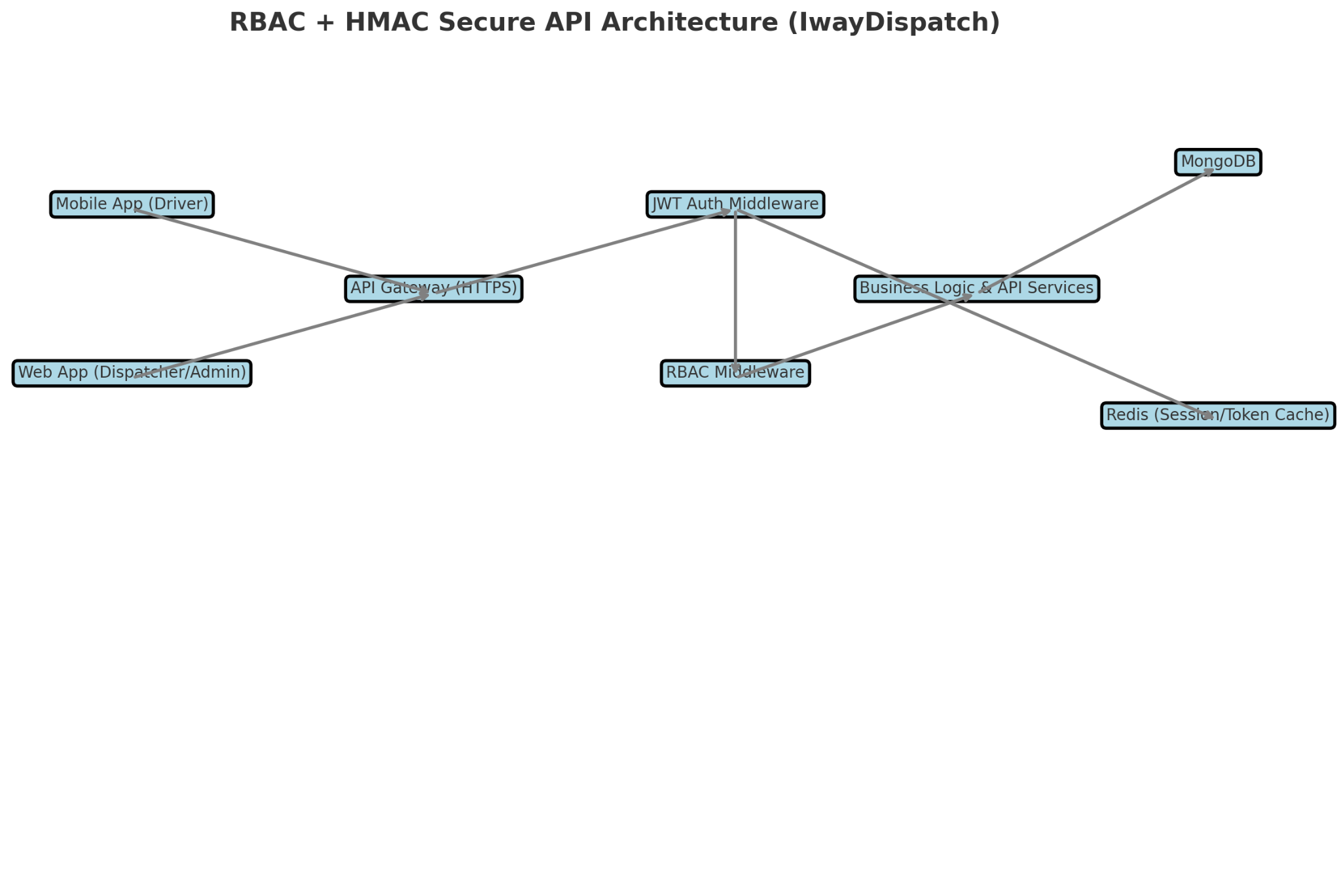
* A driver could only access their own loads and status updates
* A dispatcher could manage fleets assigned to them — not all fleets
* And only an admin could view system-wide logs or user management tools

To further tighten security, I added HMAC signing to critical payloads — especially those involving real-time GPS updates or dispatch actions. This way, even if someone somehow intercepted a request, they couldn’t replay or tamper with it — the server would reject anything that didn’t match the signature tied to the shared secret key.

**Result:**

The results were solid. Our system cleared the internal audit with no flags, and later, even passed a third-party penetration test with no critical vulnerabilities. The RBAC implementation also made it easier for developers to plug in new APIs without worrying about manual permission checks — everything was centrally managed.

I received appreciation from the security team for going beyond just standard practices and adding HMAC verification, which is not something everyone thinks about during role-based checks. More than that, I gained a deeper understanding of how a small gap in backend design can open the door to serious risks — and how taking proactive ownership can really make a difference in building secure and scalable systems.



## Improved system

### **Scenario: Ensuring Reliable Notification Delivery in BidGo Global**

**Leadership Principles: Bias for Action, Deliver Results**

**Situation:**

While working on the BidGo Global project, we were in the testing phase, and feedback started coming in — some users were not receiving important notifications. The most critical one was: when a driver accepted a user’s ride bid, the user never got to know about it. Imagine — you bid for a ride, someone accepts it, but your screen doesn’t change, and you’re just sitting there waiting. For a platform that works purely on bidding, this was a huge trust-breaking issue.

After digging deeper, I noticed it usually happened when the user had weak internet, or when Firebase Cloud Messaging (FCM) delayed the push. These weren’t rare edge cases — it was happening often enough to hurt the user experience. That’s when I realized: this wasn’t just about one missed alert — this could damage the entire platform’s reputation.

**Task:**

I decided to take full ownership of the issue, even though it wasn’t directly assigned to me. My goal was simple: make bid acceptance notifications 100% reliable — whether the user was online, offline, in poor signal, or on a slow device. The system had to work for both mobile and web, and it had to scale. We couldn’t leave something this critical to chance.

**Action:**

First, I did a complete audit of our existing flow and found the root cause: we were completely dependent on FCM for real-time push notifications. And if FCM failed (which it did, especially with Android Doze mode or poor networks), the notification was lost forever. There was no retry. No backup.

I immediately started designing a robust fallback system. I integrated AWS SQS (Simple Queue Service) into our flow. Now, whenever an important event like bid.accepted occurred, it was first pushed into an SQS queue. Then, I used an AWS Lambda function to process that message and send the notification. But here’s the trick: I built in retry logic with exponential backoff — if the first attempt failed, it would wait a bit, then try again, up to 3 times. This handled temporary network or service hiccups beautifully.

But I didn’t stop there. I asked myself — “What if FCM just doesn’t deliver at all?” So, I added a second layer of fallback. If the system didn’t receive an acknowledgment from the user’s device within 10 minutes, it would automatically send an SMS via Twilio, or an email via SendGrid — whichever contact method was available. This ensured that even if the app was closed or the push failed, the user would still be informed. No confusion, no frustration.

To make this bulletproof, I also added an acknowledgment check from the frontend. Whenever the notification was received on the device, the app would send a small confirmation back. If that didn’t come, the retry logic would kick in automatically. And everything — every send, every retry, every fallback — was logged in MongoDB. This gave us full observability and helped track delivery issues in real-time.

**Result:**

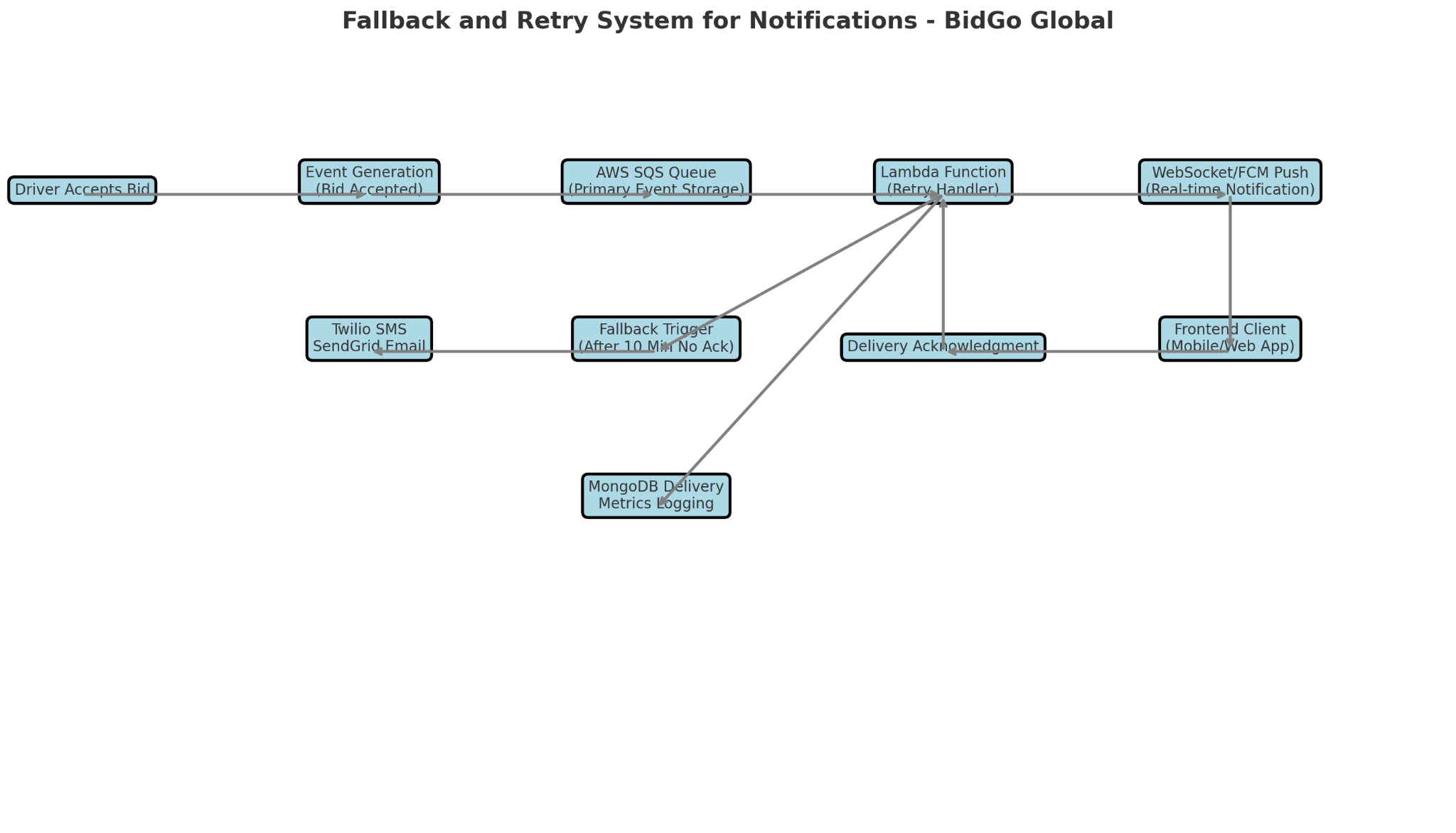
The results were incredible. Our notification delivery reliability shot up to 99.7%, and user complaints dropped to nearly zero. People trusted the system more — even if their app was closed or they were offline for a bit, they still got to know when a driver accepted their ride. The product team was so impressed that they later reused this same fallback system for other critical events like payment success and ride cancellations.

But for me, the biggest win was — I didn’t wait for the problem to explode. I noticed a pattern early, took initiative, designed a fully resilient, scalable solution, and made the system future-proof. This experience taught me that real reliability comes from thinking two steps ahead, and if you care about the user deeply, you’ll find a way to never let them feel ignored — even in bad network.  
  
Notes:  
FCM: It is used only for push-based notifications  
Choose SQS (Simple Queuing Service) if you want a managed, scalable, and AWS-integrated solution with minimal operational overhead.

Choose RabbitMQ if you need more advanced features, control over your infrastructure, and potentially wider compatibility across different environments.

Here’s the visual representation of the **Fallback and Retry Notification System** for **BidGo Global**!

This diagram shows the complete real-time and fallback flow:



* When the driver accepts a bid, an event is generated and sent to an AWS SQS queue.
* A Lambda function processes the event and sends a push notification (via WebSocket or FCM) to the user’s mobile/web client.
* If an acknowledgment is not received within a defined period, a fallback trigger is activated.
* Twilio (SMS) or SendGrid (Email) is used to ensure critical updates reach the user even if they are offline.
* All events and retries are logged into MongoDB for delivery tracking and analysis.

## Outside of your responsibility (Instincts, Tight Deadline - 2, Managed tight schedule )

### Investor Data Web Scraping

**“Tell me about a time where you had to perform a task outside of your responsibility.”**

During one of our stand-up meetings at Xnode, our founder shared a new POC task where a customer had provided a dataset containing around 1,000 investors from PitchBook. They wanted us to extract detailed team information—like names, designations, contact details, and years of experience—from each company’s website and submit it in a structured format. This task was very different from my regular development responsibilities, but since I had previous experience in web scraping, I immediately volunteered to take it up, even though scraping was not part of my daily role.

After accepting the task, I realized there was only a one-week deadline, and at the same time, I had some important sprint updates to deliver. I informed my manager, and he asked me to prioritize the customer task. But when I checked with my teammates, I found that everyone was already occupied with critical updates. I decided not to hand over my sprint tasks to others and instead managed both responsibilities myself.

For the scraping task, I initially explored two approaches. Plan A was web scraping, which was tricky due to different website structures. Plan B was using public APIs like PitchBook or RocketReach, but they required a paid subscription. Being mindful of our startup cost constraints, I trusted my instinct and went ahead with scraping. Pitchbook is costly, we wanted to do open source way as it is cost efficient, since it is 1000 companies so we were not ready to go for that solution.

Finding the correct “team” pages across different websites was a big challenge. Instead of manually checking every site or downloading junk using wget, I found an alternative. GIVE AN EXAMPLE of team page. I built a solution using **DuckDuckGo search programmatically** by querying terms like “Company Name team” or “Company Name founders,” and it worked well. I tested it on the first 100 companies and got consistent success.

I scraped the data using **BeautifulSoup**, cleaned it with **Gemini API** to remove irrelevant information, and converted it into structured JSON and CSV files. Around 60 companies gave 403(Forbidden error)/404 (data not found) errors, so I suggested a fallback plan—splitting the remaining companies among eight teammates to handle manually. This helped us achieve 95% completion within the timeline.

The customer was very happy with the smart, low-cost solution. I earned appreciation for stepping outside my usual responsibility and delivering results without impacting my original sprint work. Later, my interest in improving this solution pushed me to explore AI agents using Phidata and GroqCloud. I even built a small AI agent to automate the scraping process further, showing my ability to continuously learn and innovate.

This experience taught me that stepping outside of my regular role, trusting my instincts, and balancing multiple tasks can lead to better solutions and personal growth.

**Tell me about a time you made a decision based on your instincts.  
Or  
Tell me about a time when you have to work under tight deadlines.**

**Situation**:

In my current role, one of our customers shared a dataset with around 1,000 investors from PitchBook. Each record had an investor ID, company name, website, and location. They wanted us to extract team member details—like names, designations, contact info, and years of experience—from each company’s website, and submit it in a structured format within one week.

**Task**:

As the sole developer reporting to the company founder, I was responsible for collecting all this information efficiently and cost-effectively. The challenge was that each website was different, and we had to find the correct team or “about us” pages for scraping.

**Action**:

Initially, I explored two approaches:

Plan A: I went with my instinct and picked Web scraping as an option, though it was tricky due to varying website structures.

Plan B: I can use Public APIs like PitchBook, Rocketreach, Hunter.io, which were accurate but required a paid subscription. But, I’m sure it is not what my manager was expecting from me.

Based on instinct and my experience in scraping, I was biased towards Plan A. And I also need to be frugal in my work as it is a startup and I should present them a solution that should be cheap and reusable in future. I discussed both approaches with my manager, and we agreed to go ahead with scraping. I faced issues identifying team page URLs, as navbars and deep links varied for each site. Instead of manually inspecting all sites or relying on wget (which fetched too much junk), I tried something different. I explored DuckDuckGo search programmatically, adding queries like “company name team” or “company name founders”. It worked well—team pages appeared in the top search results. I tested this on 100 companies and got consistent success. Then I used BeautifulSoup to scrape names, roles, bios, and stored them as JSON objects. Since the data had noise, I used LLM (Gemini API) to extract meaningful parts into a clean CSV. We still faced ~60 sites with 403/404 errors. I proposed a fallback—our team of 8 members manually handled 10 companies each. This helped complete the task with over 95% success.

**Result**:

The customer was very happy with the timely delivery and appreciated the low-cost, smart solution. My decision to trust my scraping instinct saved API cost and proved scalable. This project also improved team efficiency and we could earn trust with the client. A few days after this task, I was learning and curious about AI agents. I started working on small projects and thought I can make this task even simpler by creating an AI Agent. Then, I went on to use python libraries like Phidata and groqcloud to design an AI Agent. I than used Duckduckgo search too in Phidata which makes it easier to develop a web search agent. Then I created a pipeline CSV of investors -> get companies names and website links -> automate search using Duckduckgo AI web search agent -> get the search results data -> use Gemini API to retrieve the meaningful data -> create a JSON object -> push into database. I also feel like when I learn or come across new technologies, I automatically think of how could I use them in any of my prior projects or tasks that I worked on. It helps me to Invent, reinvent, and simplify things. It’s like creating new versions of my work as the technologies upgrade.

Or

**Tell me about a time where you have to manage time.**

**Situation:**

One day during a stand-up meeting at Xnode, my founder shared a new POC task. One of our customers had provided a dataset of around 1,000 investors from PitchBook. Each record had an investor ID, company name, website, and location. They wanted us to extract team member details—like name, designation, contact information, and years of experience—from each company’s website and submit it in a well-structured format.

**Task:**

I quickly realized that this task would require web scraping. Since I already had some experience working with scraping before, I was confident that I could take up this work. I immediately came forward and explained a possible approach in the meeting. Others didn’t have much experience with web scraping, so I decided to pick this task.

Only after picking it up, I came to know that the deadline was just one week, and unfortunately, that week was also the end of our sprint. I had a few important updates I was supposed to push. I explained the situation to my manager. He asked me to prioritize this customer task and suggested handing over my current updates to someone else.

But after checking with teammates, I realized that everyone was already working on critical deliveries. Handing over would delay the sprint further. So, I decided to manage both responsibilities myself, balancing between sprint updates and completing the POC task.

The challenge was that every company’s website was different, and finding the correct team or “about us” page for scraping was tricky.

**Action:**

Initially, I explored two possible approaches:

* **Plan A:** Go with web scraping, even though websites had different structures, making it a bit risky.
* **Plan B:** Use public APIs like PitchBook, Rocketreach, or Hunter.io, which were more accurate but required paid subscriptions.

I trusted my instinct and experience and leaned towards **Plan A**, because we were a startup, and I had to be frugal in my work. I discussed both approaches with my manager, and he agreed that scraping would be better.

I started facing issues in finding deep links to the team pages because navbars and internal links were not consistent across websites. Manual inspection would have been very time-consuming, and wget command was downloading too much unnecessary content.

So I tried a new idea. I explored using **DuckDuckGo search programmatically**. I created search queries like “Company Name team” or “Company Name founders,” and surprisingly, team pages appeared in the top 5 results. I tested this on 100 companies first and got consistent results.

Then I built a scraper using **BeautifulSoup** to extract names, designations, bios, and stored them as JSON objects. Since the raw data had a lot of noise, I used **Gemini API** to clean and extract only the meaningful parts and converted them into structured CSV files.

Still, around 60 company websites threw 403 or 404 errors. To handle this, I suggested a fallback plan—divide the remaining work among the team. Each of us manually handled around 10 companies. This helped us finish the work with over **95% accuracy** within the deadline.

**Result:**

The customer was very happy with the timely delivery and appreciated the cost-effective and smart solution. My decision to trust my scraping instincts helped us avoid costly API subscriptions and delivered a scalable, reusable solution.

A few days after completing the task, my curiosity about AI agents grew even more. I started working on small side projects and thought about how this investor scraping task could be automated even better using AI agents.

I explored tools like **Phidata** and **GroqCloud** to build web search agents. I designed a pipeline: starting from the investor CSV → getting company names and website links → automatically searching using a DuckDuckGo AI web search agent → fetching the data → summarizing it with **Gemini API** → converting it into structured JSON → and storing it in the database.

Now whenever I learn a new technology, I naturally start thinking about how I could have used it to improve my previous work. It helps me to invent, reinvent, and simplify things continuously. I feel like it’s my way of creating newer, better versions of the solutions I have built earlier.

## Critical Feedback from Manager

### Communicating with others

**Can you think of a time when you got critical feedback that changed how you operated?**

**Situation:**

When I initially joined the Xnode team as a Full Stack Engineer, I was assigned my first feature task — to integrate cloud storage systems like Google Drive and OneDrive into our product. I was excited and wanted to do everything perfectly, so I started exploring the existing codebase to find the right place to implement this feature. But there was no proper documentation, and the structure was quite complex for someone new. I spent almost half a day just tracing the flow and trying to figure things out on my own.

**Task:**

My task was to integrate the feature within the sprint timeline, but more importantly, to pick up the architecture fast and work efficiently in a startup environment. I was new, so I wanted to prove myself by solving things independently, without bothering others.

**Action:**

After I updated my progress during a stand-up, my manager gave me some honest feedback — he told me that in a fast-paced startup, spending more than a couple of hours stuck on something isn’t ideal. He said it’s okay to try independently, but if I’m blocked, I should reach out quickly. That feedback really hit me, because I realised I was treating it like a solo assignment, not a team effort.

I took that feedback positively and immediately changed my approach. From then on, whenever I got stuck beyond 1–2 hours, I didn’t hesitate to reach out to a senior engineer for a quick call or clarification. I also went one step further — since the documentation was missing, I took the initiative to start documenting useful things as I learned, and even proposed recording team discussions and architecture calls, so future new joiners wouldn’t face the same issues.

**Result:**

That one piece of feedback really changed how I work. I became more collaborative, faster at unblocking myself, and more open to asking questions early. It not only helped me deliver features on time, but also built strong relationships with my team. Today, even my teammates say that the internal docs and recorded calls have made onboarding much easier.

This experience taught me that feedback is not criticism — it’s guidance, and accepting it with the right mindset can lead to long-term growth, both technically and professionally.