Tab 1

Missed a deadline: API to parse CSV and Excel Files

Less time quick action: Oracle connector metadata extraction, special characters

Prioritising the tasks - customer focused approach. Bug, feature enhancement, new feature

Negative Feedback - React test cases

Delivered a project under tight deadline / innovation: Xplenty etl tool

Team mate disagree

While working as a Software Developer on the Intentwise Tool at Beehyv, To display the data on dashboard we ran queries to extract aggregated insights from data. These queries involved joining multiple tables with millions of records, which resulted in slow performance. Since the data in these tables only changed twice per day, running expensive joins repeatedly was inefficient. I proposed using a Materialized View to precompute the results and store them , reducing query execution time. However, a teammate disagreed, arguing that Materialized Views required manual refreshes and might not be worth the effort.

To optimize daily reporting queries while ensuring data consistency and efficiency, I analyzed the query execution plans using EXPLAIN ANALYZE and demonstrated that joins were the main bottleneck. To validate my approach, I implemented a proof of concept (PoC) where I created a Materialized View to store precomputed results and scheduled a nightly refresh using pg_cron to keep the data updated. I then compared execution times between querying the Materialized View vs. running joins on raw tables, and the results showed a 70% improvement in query execution speed. I presented these findings to my team, explaining that while a manual refresh was required, automating it would make the process seamless and efficient.

Initially, my teammate was hesitant, concerned about stale data, but after seeing the significant performance improvements and understanding that data consistency wouldn't be an issue due to the automated refresh, they agreed to move forward. As a result, we successfully implemented the Materialized View, reducing query execution time from several minutes to seconds, while the pg_cron job ensured that data was always up to date without requiring manual intervention. This optimization enhanced dashboard responsiveness and improved the overall user experience. The success of this solution also encouraged the team to adopt similar precomputed query strategies for other performance-critical reports, improving overall system efficiency.

Missed Deadline

During my time at Beehyv, I was developing an API to parse Excel and CSV files and store the extracted data into a database. Initially, I implemented a sequential processing approach, which worked fine for smaller files. However, during the testing phase, I encountered performance

issues when handling large files - the API took too long to process and, in some cases, even failed due to memory constraints. Since this issue surfaced late in the development cycle, it impacted the timeline, and I was unable to meet the original deadline.

To fix the issue, I analyzed the bottleneck and identified that loading the entire file into memory at once was the main cause of high memory usage and slow performance. To optimize the process, I refactored the API to handle files in chunks, reducing memory load. Additionally, I implemented parallel processing using multithreading, allowing multiple chunks to be processed simultaneously, which significantly improved speed. After these optimizations, I conducted extensive testing with large datasets to validate the improved performance.

As a result, the optimized API handled large files smoothly, reducing processing time by over 50% while preventing memory-related failures. Although I missed the initial deadline, I proactively communicated with the team, provided a revised timeline, and delivered the API. This experience taught me the importance of early performance testing and the need to anticipate scalability challenges in the design phase, ensuring future projects are optimized from the start.

Fast response, Calculated risk

While working on the Data Catalog Tool, we had a customer escalation where the Oracle connector's metadata extraction was failing, completely blocking the customer's workflow. Since they relied on this functionality for their operations, waiting for the next release cycle wasn't an option. I had to make a quick decision on whether to immediately fix and deploy a patch or follow the standard release process, which would have delayed the resolution. Given the urgency, I decided to take a calculated risk and push out a hotfix to unblock the customer.

To find the root cause of the issue quickly, I jumped on a call with the customer, analyzed their logs and metadata structure, and identified that the failure was due to special characters in their database object names, which the connector wasn't handling correctly. Instead of adding this to the roadmap for the next release, I listed down the special characters they were using, updated the connector code to properly handle them, and ran internal tests to ensure there is no regression. I immediately deployed the fix, allowing the customer to resume operations without further delays.

To reduce the risks, I made sure the code changes were minimal, reducing the chance of breaking existing functionality. I also tested the fix on different metadata structures and released it first to the affected customer before rolling it out more broadly. Additionally, I monitored logs post-deployment to catch any unforeseen issues early.

As a result, the customer was unblocked immediately, and their workflows resumed without disruption. The fix also made the Oracle connector more robust, preventing similar issues for other users. Later, we included the improvement in the next official release, ensuring long-term stability.

Looking back, I would have proactively tested for special character handling earlier by expanding our test cases and validation rules. This experience has taught me to make quick and calculated decisions in critical situations.

Negative feedback

During my internship at Beehyv, I was developing React components for a web application. While I ensured the UI was functioning correctly during manual testing, I did not write automated test cases using Cypress to validate the behavior of the components.

During a code review, my manager pointed out the lack of test coverage and emphasized the importance of writing end-to-end tests to catch regressions early.

I needed to address the feedback by writing Cypress test cases to ensure that all critical user interactions were covered and the components worked as expected.

Instead of feeling discouraged, I took the feedback as an opportunity to learn about Cypress testing for React applications.

My manager appreciated my effort in quickly improving test coverage.
The Cypress tests helped catch bugs early, reducing the risk of breaking changes in future updates.This experience taught me the importance of automated testing in frontend development, making me more disciplined in writing tests..

Multiple Tasks

While working on multiple tasks at Beehyv, I follow a customer-focused approach to prioritization. My first priority is always to address any issue that is blocking the customer, ensuring minimal disruption to their workflow. After resolving critical blockers, I move on to feature enhancements for existing functionality, and finally, I focus on developing new features that provide long-term value.

For example, while working on the **Data Catalog Tool**, I had three tasks at the same time:
1. Fixing a bug in an existing connector that was impacting a live customer.
2. Enhancing a feature in another connector to improve performance.
3. Developing a new database connector to support additional data sources.

Action Plan for Prioritization:

Step 1: Resolve the Blocking Issue First

I immediately addressed the bug in the existing connector since it was directly affecting a customer and preventing them from using the tool.

Debugged, tested, and released a fix quickly to restore normal operations.

Step 2: Implement Feature Enhancements

- Once the blocking issue was fixed, I focused on enhancing an existing connector to improve efficiency and performance.
- Since customers were already using it, this improvement added value without disrupting workflows.

Step 3: Develop New Features

- Finally, I worked on creating a new database connector, which had long-term benefits but was not impacting current users.
- Ensured proper design and testing to integrate it smoothly into the system.

Outcome:

The customer-facing bug was resolved first, preventing major disruptions.
The feature enhancement was successfully implemented, improving efficiency for existing users.
The new database connector was developed without affecting high-priority tasks.

Delivered the project on time (ETL Project)

Situation:
While working on the Intentwise Tool, the data engineering team was manually checking report statuses every day to determine which reports were enabled or disabled for each organization. This process was time-consuming and error-prone, and it involved manually adding or removing ETL packages and schedules in Xplenty to copy data from the Intentwise platform database to the customer database.

Task:
I was tasked with automating this process to eliminate the manual effort, reduce errors, and ensure the smooth running of the system. The deadline was tight, as I needed to go through the Xplenty API documentation to understand the endpoints and integrate them into the automation pipeline.

Action:
To meet the deadline, I designed and developed an automation pipeline that dynamically

checked report statuses for each organization and automatically created, updated, or removed ETL packages in Xplenty. I explored the Xplenty API documentation to identify the necessary endpoints for automation. Additionally, I added the respective packages to the schedule in Xplenty to copy data from the Intentwise platform database to the customer database. I also implemented automated scheduling for data transfers and integrated real-time Slack notifications to keep the team informed. To ensure reliability, I handled edge cases like missing reports or API failures to prevent data inconsistencies.

Result:
The automation reduced manual work by 80%, saving several hours of engineering effort daily. This eliminated human errors, ensuring only valid reports were processed, and significantly improved the accuracy and reliability of analytics. The outcome was a more efficient process that allowed engineers to focus on strategic work, rather than repetitive tasks.