NBA Player Value Model

Andrew Chau, Rish Jain, Raniel Quesada, Keon Sadeghi, Utkarsh Sharma
Department of Applied Data Science
San José State University
Emails: andrew.chau@sjsu.edu, rish.jain@sjsu.edu, ranielangelo.quesada@sjsu.edu, keon.sadeghi@sjsu.edu, utkarsh.sharma01@sjsu.edu

***Abstract*—This project examines the gap between NBA player salaries and their actual on-court performance by combining five seasons of advanced statistics with contract data. We built a complete end-to-end data pipeline using Amazon S3, Apache NiFi, and Amazon Redshift to ingest, clean, transform, and store multi-season NBA datasets. The raw CSVs required extensive cleanup because the files used different column names, included duplicate rows for players who were traded mid-season, had missing values, and did not follow the same schema each year.**

Using a structured ETL process, we standardized player and team information, performed dimension lookups, enforced surrogate keys, and organized everything into a star-schema warehouse. From there, analytical SQL queries and a regression-based salary model were used to identify which advanced metrics best explain paycheck differences and to classify players as overpaid or underpaid.

The final cleaned dataset feeds into Tableau dashboards that highlight efficiency, usage, and salary fairness across the league. In general, this project brings together pipeline development, handling semi-structured/NoSQL-style data issues, warehouse design, and applied analytics, showing how data engineering and statistics can help make smarter decisions in professional sports.

I. Problem Statement

NBA teams invest millions in player contracts, yet salary decisions often do not match on-court value. Some players are paid far more than their impact while others are underpaid despite strong performance. Our goal is to study this gap by combining salary data with five seasons of advanced metrics. This project brings together concepts from sports analytics, data engineering, pipelines, data warehouse design, and statistical modeling. We aim to identify which metrics best predict player value and determine which players are underpaid, overpaid, or fair-valued.

II. Background

Advanced metrics have become central in basketball because they capture efficiency, usage, defensive impact, and overall contributions in ways that basic points or rebounds cannot. Metrics such as BPM, VORP, and Win Shares summarize a player's influence on team success, while TS pct highlights scoring efficiency and PER combines multiple box score components into a single number. Salary decisions depend on similar evaluations, yet contracts often reflect past reputation, draft status, or team context rather than pure performance. By combining analytics across multiple seasons, we can observe how consistently teams reward the attributes that matter most and where the disconnect appears.

III. Metric Selection Justification

We selected seven advanced metrics because they represent the most reliable and widely used indicators of player value. These seven metrics are PER (Player Efficiency Rating), TS% (True Shooting Percentage), USG% (Usage Percentage), WS (Win Shares), BPM (Box Plus Minus), VORP (Value Over Replacement), and MP (Minutes Played). PER, BPM, VORP, and WS measure a player's overall impact relative to league averages and replacement–level performance. TS% captures scoring efficiency, USG% reflects how involved a player is in the offense, and MP shows how durable and consistent a player is across the season. Together, these metrics provide a balanced view of efficiency, workload, and overall contribution, which makes them strong predictors of salary.

IV. Fair Value Classification Rule

To determine whether a player is underpaid, overpaid, or fair-valued, we compared each player's actual salary to the predicted salary generated by our regression model. To calculate the percentage difference between a player's actual salary and their predicted salary, we used:

$$\text{gap\_pct} = \frac{\text{Actual Salary} - \text{Predicted Salary}}{\text{Predicted Salary}}$$

We applied a fifteen percent threshold, meaning that if $\text{gap\_pct} < -0.15$, the player is classified as **underpaid**; if $\text{gap\_pct} > 0.15$, the player is classified as **overpaid**; and if $-0.15 \leq \text{gap\_pct} \leq 0.15$, the player is classified as **fair-valued**. In simpler terms, if a player's actual salary falls within fifteen percent of their predicted value, we classify them as fair-valued. This rule provides a balanced and realistic way to evaluate contract fairness across different roles and salary ranges.

V. Project Significance

This project has real value because decisions about contracts influence team building, competitive balance and long term spending. Teams need to know which players offer strong performance for their cost and which contracts may limit their ability to add talent in future seasons. Our analysis offers a way to evaluate players using a consistent, data-driven criteria rather than relying only on reputation or common statistics. This type of study supports smarter financial planning, identifies undervalued players who could be signed at a bargain and highlights areas where teams may be paying more than the

production they receive. It also shows how analytics can guide business decisions in sports which aligns with the growing role of data in professional leagues.

VI. ETL Design Decisions

Our end-to-end workflow uses a structured ETL (Extract–Transform–Load) model. We chose this approach because the NBA stats came from several raw CSV files, had inconsistencies, and needed a good amount of cleaning before they were ready for analysis. By sticking to ETL, we made sure that only clean, checked, and properly structured data was loaded into our Redshift warehouse.

A. Extract

1) Reason for Choosing ETL Instead of ELT: We selected ETL over ELT because the raw NBA files were not suitable for direct loading. They contained inconsistencies, duplicate player entries, missing data, and variations in naming conventions. Loading these files directly into Redshift would have pushed complex transformations into the warehouse, increasing query latency, storage consumption, and schema complexity. ETL allowed us to clean and standardize all data first, resulting in a more efficient warehouse.

2) Extraction Method: The extraction phase ingested season-level CSV files directly from Amazon S3 using NiFi's `ListS3` and `FetchS3Object` processors. This supports automatic ingestion of new seasons without manual triggers. Each file was processed as its own FlowFile, preserving metadata such as filename and timestamp. Using S3 ensured a scalable, cloud-native extraction pattern that is easy to maintain and extend.

B. Transform

1) Data Cleaning Logic: The raw datasets presented several issues that required targeted cleaning:

- Duplicate rows caused by mid-season trades.
- Combined team rows (e.g., "2TM", "3TM") representing full-season totals.
- Special characters and inconsistent player names.
- Placeholder values such as `-9999`.

To ensure accuracy, we preserved the aggregated multi-team rows for traded players and removed duplicates to avoid double-counting.

2) Transformations Implemented: During transformation, the following operations were performed:

- Standardizing player names by removing accents and special characters.
- Preserving combined team categories since they contain full-season totals.
- Renaming and normalizing columns (e.g., `TS%` → `TS_PER`).
- Casting fields to appropriate data types (`INT`, `DECIMAL`).
- Filtering invalid surrogate keys to maintain referential integrity.

3) Dimension Lookups and Key Matching: A major component of the transformation stage was attaching surrogate keys from the dimension tables:

- `player_key`
- `team_key`
- `season_key`
- `salary`

We implemented this using NiFi's `LookupRecord`. We discovered that `LookupRecord` is extremely strict: lookup paths must exactly match FlowFile field names in case, spelling, and spacing. Even small inconsistencies caused silent lookup failures and missing dimension keys.

4) Join Workarounds Due to NiFi Limitations: NiFi does not include a `JoinRecord` processor. When lookups became unreliable or insufficient, we used a downstream Python script to complete join operations. Python provided:

- Explicit control over join conditions,
- More predictable schema handling,
- Improved management of field mismatches,
- Easier debugging.

5) Transformation Flows:: Five different process groups were used to transform the data. The processors used were ListS3, FetchS3Object, UpdateAttribute, UpdateRecord, QueryRecord, MergeRecord, ExecuteStreamCommand, and PutS3Object.

a) Clean_salaries: places player_season_key in salaries.csv:: Flow: ListS3 → FetchS3Object_salaries → UpdateAttribute → UpdateRecord → UpdateRecord → QueryRecord → PutS3Object

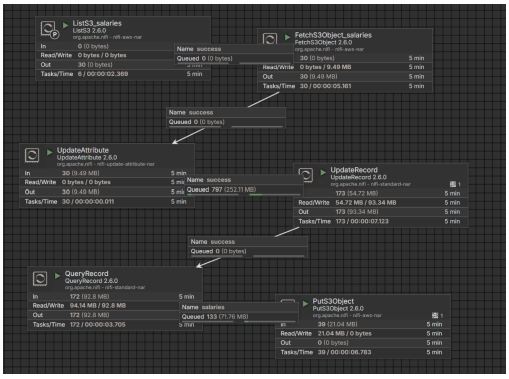

Fig. 1. Apache Nifi clean_salaries flow.

ListS3 initialized the connection to the S3 bucket while FetchS3Object_salaries retrieved the salaries.csv file inside. UpdateAttribute created a player_season_key using $Player_$Season to connect the salaries and statistics files for the fact_player_season table; the player_season_key contained each player and season for both. UpdateRecord worked at the record level and used concat(/Player, '_', /Season) to add the player_season_key to the FlowFile itself. UpdateAttribute was needed because Nifi's syntax would not have recognized the Player and Season columns without them being mentioned. QueryRecord selected every column from the

FlowFile and finally PutS3Object placed the cleaned FlowFile in salaries_cleaned.csv in the S3 bucket.

b) Dim_player_etl: creates dim_player.csv:: Flow: ListS3 → FetchS3Object → QueryRecord → UpdateAttribute → MergeRecord → UpdateAttribute → PutS3Object

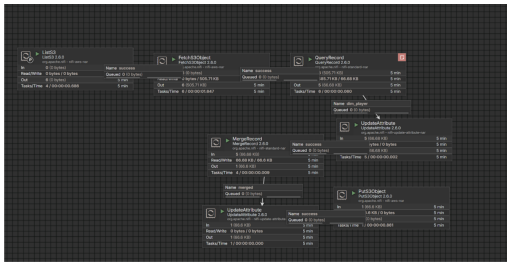

Fig. 2. Apache Nifi dim_player flow.

ListS3 initialized the connection to the S3 bucket, but in this case it used the prefix `stats/` to retrieve all 5 statistics files for each season located in that folder. FetchS3Object then made FlowFiles for each file; each processor is able to handle multiple FlowFiles simultaneously. QueryRecord selected each distinct player id and full player name. Next, UpdateAttribute used a merge key "dim" such that MergeRecord could merge all 5 files containing these 2 new columns into one file. UpdateAttribute was used again to confirm the merge, locking in the one file, and finally PutS3Object placed this `dim_player.csv` file in the S3 bucket.

c) Dim_team_etl: creates dim_team.csv:: Flow: ListS3 → FetchS3Object → QueryRecord → UpdateAttribute → PutS3Object

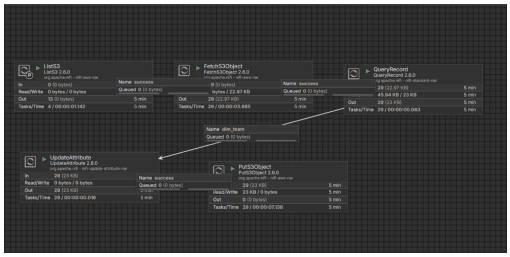

Fig. 3. Apache Nifi dim_team flow.

ListS3 initialized the S3 connection as FetchS3Object retrieved the nba_teams_conference.csv file. Next, QueryRecord selected the team key which was the team abbreviation, and team name. UpdateAttribute updated this FlowFile to confirm the change and finally PutS3Object placed this `dim_team.csv` file in the S3 bucket.

d) Dim_season_etl: creates dim_season.csv:: Flow: ListS3 → FetchS3Object → QueryRecord → UpdateAttribute → PutS3Object

ListS3 initialized the connection to the S3 bucket while FetchS3Object retrieved the salaries.csv file; this was the only file that contained every season needed in one place for the most efficient process. Next, QueryRecord selected every distinct season as the season key or end season, including

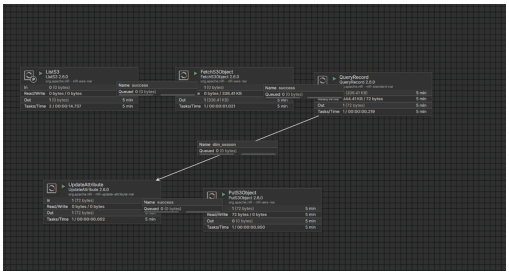

Fig. 4. Apache Nifi dim_season flow.

subtracting 1 from the season to get the start season; the end years kept were 2021 to 2025 as our data was analyzed and metrics were determined for statistics from the past 5 seasons. Last but not least, UpdateAttribute confirmed the changes and PutS3Object placed this `dim_season.csv` in the S3 bucket.

e) Fact_etl: creates fact_player_season.csv: Flow: ListS3 → FetchS3Object → UpdateAttribute → UpdateAttribute → UpdateRecord → UpdateRecord → ExecuteStreamCommand → QueryRecord → UpdateAttribute → MergeRecord → UpdateAttribute → PutS3Object

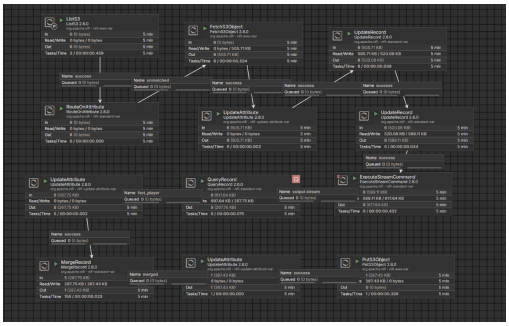

Fig. 5. Apache Nifi fact_player_season flow.

ListS3 initialized the connection to the S3 bucket by using the `stats/` prefix to access the folder containing all five statistics files. `FetchS3Object` then retrieved the five files as FlowFiles. The first `UpdateAttribute` processor created both the `player_season_key` and the `season_key` by using filename-based pattern logic to extract season information. Instead of listing the full expression here, we refer to the detailed code included in the appendix here. The following `UpdateAttribute` processor applied the extracted season value to each FlowFile before transformation.

This logic extracted the final two digits from each statistics filename and prefixed them with "20" to produce the correct season identifier. The next `UpdateAttribute` added this season column to every FlowFile, and a subsequent `UpdateAttribute` inserted the `player_season_key`. `ExecuteStreamCommand` then performed the join operation using `join_players.py`, which incorporated salary information from the cleaned salary dataset into the FlowFiles.

`QueryRecord` selected the player identifier, team abbreviation, season, all statistics, and the salary required for

metric calculation. A following UpdateAttribute confirmed the changes and added a merge key, "stat," which was used by MergeRecord to combine all FlowFiles into a single dataset. This merged output contained the player key, team key, season key, all required statistics, and salary for every player across all seasons and teams. Finally, UpdateAttribute confirmed the last modifications, and PutS3Object wrote the completed file to the S3 bucket as fact_player_season.csv.

C. Load

1) Loading Strategy Into Redshift: After transformations were completed upstream, the final stage of ETL loaded the cleaned data into Amazon Redshift. Redshift's columnar and MPP architecture makes it ideal for analytical workloads. Our warehouse follows a star-schema design with a central fact table and several supporting dimension tables, enabling efficient filtering, aggregations, and joins on player-season data.

2) Readiness for Analytics and Dashboards: Once loaded, the warehouse schema supported fast analytical queries and integrated directly with Tableau for dashboarding. Since transformations were fully completed upstream in NiFi, Redshift stored clean, validated, analysis-ready data suitable for performance evaluation, trend analysis, salary insights, and cross-season comparisons.

VII. Data Warehouse Design

Our warehouse follows a star schema centered on a fact table that stores season-level player performance and salary information. The central table, nba.fact_player_season, has a grain of one row per player, per team, per season. It contains foreign keys to the player, team, and season dimensions (player_key, team_key, season_key) and stores both raw statistics and derived analytics. The raw measures include games played (G), games started (GS), minutes played (MP), age, and the seven advanced metrics used in our model: PER, TS_PER, USG_PER, WS, BPM, VORP, and MP. The fact table also stores the actual salary for each season.

On top of the raw statistics, the fact table includes several derived fields generated by our regression model and business rules. The value column represents the model's predicted value for a player based on their performance metrics. The deserved_salary field stores the predicted fair salary, and salary_gap captures the monetary difference between the actual salary and the predicted value. We also store a numeric indicator for overpaid or underpaid status, as well as a fair_value boolean flag that identifies whether a player's contract falls within our fifteen percent threshold. Keeping these derived metrics inside the fact table enables efficient queries and dashboard generation without the need to recompute the model during analysis.

The fact table links to three compact dimension tables. The nba.dim_player table holds a stable player_key and the player's full name, which helps resolve naming inconsistencies across seasons. The nba.dim_team table contains team_key, team_name, and conference information, allowing dashboards to slice results by franchise or conference. The nba.dim_season table stores the season_key and season start year, enabling trends to be grouped over time. This design keeps the dimensions narrow and descriptive while concentrating numeric measures in the fact table, following standard dimensional modeling practices and supporting efficient analytical queries in Redshift.

Fig. 6. Star Schema.

VIII. Lessons Learned

Throughout the development of our end-to-end NBA analytics pipeline, we encountered several technical challenges that led to meaningful lessons in ETL design, data modeling, NiFi processor configuration, and warehouse integration. These issues significantly improved our understanding of how real world data engineering systems behave and how small configuration details can break downstream workflows.

A. The Importance of Inspecting FlowFile Content in NiFi

One of the most impactful lessons was learning to inspect FlowFile content using Data Provenance before debugging QueryRecord processors. Many of our QueryRecord failures occurred because the fields we assumed existed (e.g., Team, Season, PER vs. PERCE) were not actually present in the transformed FlowFiles. NiFi is very strict about field names, and any mismatch immediately breaks SQL queries. This taught us that data should be validated at each stage of the flow, and real-time schema inspection is very important for avoiding errors.

B. LookupRecord Requires Perfect Input Keys

LookupRecord is very strict about input keys. We learned that if the lookup path does not match the field name exactly—for example, using /Team when the CSV column is actually /Tm, or when NiFi renames fields during processing—the lookup will fail. When this happened, team_key, season_key, and salary values did not get populated

downstream. It was a reminder that dimension keys need to be defined carefully and the lookup paths must match the FlowFile's exact column names. Even a small mismatch, such as capitalization or an extra space, is enough to break the lookup.

C. Discovering Alternative Joining Methods (Python Script Workaround)

When lookup matching became too complex or unreliable, we explored alternative methods for performing joins. One workaround was implementing a Python script downstream that could perform the join logic more directly and predictably than NiFi's `LookupRecord` processor. Python gave us:

- explicit control over join logic,
- clearer debugging,
- flexible handling of mismatched column names,
- reliable joining even when NiFi inferred schema differently.

This approach helped us complete the join operations when NiFi failed to match keys correctly.

D. Conclusion

These lessons were substantial because they came directly from the challenges we encountered while building a real, multi-step ETL pipeline. They reflect practical, hands-on insights into schema drift, processor limitations, key-matching complexity, and the need for creative workarounds.

IX. Challenges Faced

We encountered several challenges while building the NBA ETL pipeline and data warehouse, and each issue required us to rethink parts of our design. One of the earliest problems came from the raw data itself. Players who were traded mid-season appeared multiple times in the same file, sometimes with standard team codes and other times with combined labels such as "2TM" or "3TM." Determining which row to keep and ensuring that the final totals accurately reflected each player's full season required custom logic and repeated testing. We also faced inconsistencies in player names, including accented characters, which caused mismatches during joins and group operations.

Data type consistency across the five season files introduced another major challenge. Small formatting differences—such as salary appearing as a string in one file or advanced metrics stored as text—broke transformations and caused load failures in Redshift. Considerable time was spent standardizing decimal formats, handling null values correctly, and validating each transformation step. Although this required more effort than expected, it ultimately made the pipeline significantly more reliable.

Building the NiFi flow also came with a substantial learning curve. Understanding how processors pass FlowFiles, how `LookupRecord` performs lookups, and how schema inference behaves required multiple design iterations. In several cases, NiFi failed silently due to schema mismatches, forcing us to inspect provenance logs to identify the root cause. Configuring S3 authentication and ensuring that the entire pipeline ran automatically without manual intervention added another layer of complexity.

Despite these challenges, each obstacle improved our understanding of real-world ETL system behavior. Working through the issues strengthened our skills in pipeline design, data modeling, and end-to-end reliability, and these lessons directly influenced the stability and accuracy of the final project.

X. Analytical Queries And Business Insights

A. Team counts of overpaid/underpaid

This query looks at how well each NBA team uses its salary budget by comparing how many players are overpaid versus how many are underpaid based on their performance. It uses two CTEs, one for players whose actual salary is higher than their performance value, and one for players who deliver more value than they're paid for. To keep the results meaningful, the query filters out players with fewer than 750 minutes played. A full join is used so every team appears in the final output, even if they only fall into one of the two groups.

The results show clear trends in roster efficiency. For example, OKC (49 underpaid, 5 overpaid) and MEM (39 underpaid, 8 overpaid) show strong value across their rosters, while teams like DAL and LAC have more overpaid players, pointing to heavier or less optimized contract structures. Overall, this single query ties together performance, salary fairness, and team-level strategy to give a straightforward business insight into how efficiently each front office is spending its money.

B. Season count of overpaid/underpaid

This query compares how many players were underpaid versus overpaid in each season across the five year dataset, using a 750 minute minimum to make sure only meaningful contributors are included. Two CTEs calculate the number of overpaid and underpaid players by season, and a full join merges the results so every season appears in the output. The results show clear year-to-year patterns: seasons like 2022 (214 underpaid, 117 overpaid) and 2025 (208 underpaid, 146 overpaid) have large surpluses of underpaid players, as a result of rookie contracts and team-friendly extensions, while 2024 is more balanced (166 underpaid, 159 overpaid), highlighting the impact of salary inflation and veteran-heavy rosters. Overall, the query gives a league-wide view of salary efficiency over time and shows how market conditions, contract cycles, and cap structure shape the number of players who outperform or underperform their salaries each season.

C. Top 10 underpaid players with largest salary gap

This query finds the ten most underpaid players in the NBA by calculating the gap between what a player earned and what their performance indicates they should have been paid. It joins the fact table with the player dimension to return each player's name, their salary gap, the season, and the team they played for. To keep the results meaningful, only players with at least 750 minutes are included so small-sample players do not distort the rankings.

The results highlight players like Luka Doncic, Donovan Mitchell, Trae Young, Ja Morant, and Tyrese Haliburton, whose production far exceeded their salaries in specific seasons. This usually happens because they are on rookie contracts or early extensions. The main business insight is that teams with high-performing young players often gain major surplus value, giving them more flexibility to build strong rosters while staying within salary cap limits.

D. Top 10 overpaid players with largest salary gap

This query identifies the ten most overpaid players in the dataset by ranking those with the largest negative salary gaps, meaning their actual salary was much higher than their performance-based value. It joins the fact table with the player dimension to return each player's name, salary gap, season, and team, while filtering out low-minute players and combined-team placeholders. The results show several well-known contracts that have been widely debated in the NBA, including Bradley Beal, Ben Simmons, Paul George, Rudy Gobert, and Russell Westbrook, whose salary gaps range from roughly 28 million to 36 million depending on the season. These cases often reflect max deals given to aging stars, or injury-affected years. The repeated appearance of players such as Bradley Beal and Rudy Gobert across multiple seasons shows how long-term, high-value contracts can become financial burdens when on-court performance drops. Overall, this query provides clear insight into which teams are carrying the heaviest salary commitments relative to player value.

E. Comparison of average deserved salary vs average salary gap by age

This query compares each age group's average deserved salary to its average salary gap, which shows how contract fairness changes across a player's career. The results clearly reveal a lifecycle pattern. Players from ages 19 to about 26 have positive gaps, which means that they are underpaid relative to performance. This aligns with rookie-scale contracts and early extensions that limit how much young, high-impact players can earn. Around age 27, the trend reverses and most players in their late twenties and thirties show negative gaps, meaning they are overpaid compared to their on-court value. These negative gaps grow larger with age, reflecting long-term veteran contracts that extend past a player's peak production. A few extreme values at ages 40+ result from very small samples. Overall, this query highlights how NBA salary structures tend to undervalue young talent and overvalue aging veterans, which is an important insight for cap management and roster planning.

F. Comparison of average deserved salary vs average salary gap by position and season

This query compares the average deserved salary and average salary gap for each position across all five seasons. By grouping the fact table by position and season, it shows which roles tend to be underpaid or overpaid relative to their performance. The results reveal a few patterns. Centers and small forwards stay close to even, with small positive or negative gaps depending on the season. This suggests that these positions are generally paid in line with their value. Power forwards show more fluctuation, with certain seasons where they appear overpaid and others where they are slightly underpaid. Point guards have some of the highest deserved salaries, reflecting their larger role in shot creation and playmaking, and their gaps switch between positive and negative depending on the year. Shooting guards consistently show the largest positive salary gaps, meaning that they tend to be underpaid relative to their performance. This is likely due to strong scoring depth in the league. Overall, this query highlights how salary fairness varies across positions and seasons, offering insight into how different roles are valued in the modern NBA.

G. Comparison of average deserved salary vs average salary gap by position

This query compares the average deserved salary and average salary gap for each position across all seasons combined. It provides a high-level view of how different roles are valued in the league. Centers and small forwards sit very close to zero on average salary gap, which means teams generally pay these positions at a level that matches their performance. Power forwards show a slight negative gap, suggesting mild overpayment on average. Point guards have the highest deserved salary among all positions, reflecting their importance in ball handling and offensive creation, but their average gap stays close to even. Shooting guards stand out with the largest positive salary gap of about two million dollars, indicating that they are the most underpaid group relative to the value they produce. Overall, this query shows that most positions are paid fairly when averaged across seasons, with shooting guards being the main exception.

H. Comparison of average deserved salary vs average salary gap by season

This query compares the average deserved salary and average salary gap for each season, providing a year-by-year view of how player compensation aligns with performance. The results show that deserved salary stays relatively stable across the five seasons and ranges from about eight to ten million dollars on average. However, salary gaps show modest variation. The 2021, 2022, and 2023 seasons all show positive gaps, meaning players were generally underpaid relative to their performance in those years. The 2024 season is nearly balanced with an average gap close to zero, indicating that salaries closely matched model value during that period. The 2025 season shifts slightly positive again. Overall, this query shows that while player value remains consistent year to year, salary fairness can fluctuate depending on contract cycles, cap changes, and team spending patterns.

XI. Code Walkthrough

A. `join_nbaplayerval.ipynb`

This notebook contains the analytical portion of the project. It loads the cleaned, warehouse ready dataset and performs

regression modeling used to estimate each player's fair salary. The notebook begins by importing the combined stats and salary data, applying final filters such as removing low minute players and handling any remaining missing values. It then selects the seven advanced metrics used in our model, encodes positional data and fits a regression to predict fair value. After training the model, the notebook generates predicted salary, calculates the salary gap and labels each player as underpaid, overpaid or fair-valued using the fifteen percent threshold. The notebook also includes summary tables, visualizations and outputs used in the analytics section of the report and dashboard.

B. `join_players.py`

The script handles the final cleanup and joining step that NiFI could not do consistently. We run it through an ExecuteStreamCommand processor inside the fact_etl group and it uses the original salaries file that is mounted inside the NiFi container since that is the only way NiFi can access it. The script reads both the stats and salary data, fixes the column names and removes special characters like percent signs or hyphens because they kept breaking Calcite SQL inside QueryRecord. It also converts fields such as age, games played, games started and minutes played into integers so the data matches the structure of our fact table in Redshift. For players who were traded midseason, the script keeps only the combined season rows like 2TM or 3TM and removes the team specific versions to stop double counting. It also drops rows where the player key is `-9999` since those are league average placeholders and not real players. Once everything is aligned and cleaned, the script merges the salary information into the correct season and outputs a single, reliable dataset that we load directly into Redshift for analysis.

XII. Teamwork

Our group divided the project into clear areas while staying connected through regular check ins and shared work sessions. Rish focused on building and running the NiFi flow which served as the backbone of our ETL pipeline. Keon and Utkarsh supported this stage by researching processor options and helping troubleshoot lookup and schema issues. Rish also wrote the analytical SQL queries in Redshift and Keon and Utkarsh reviewed the results and interpreted the insights for the analytics section of the project. Rish also set up the Redshift data warehouse by creating the schema, loading the cleaned fact and dimension tables and preparing the environment for analytical queries and Tableau integration. All team members were present when working on most aspects of the data warehouse portion to give insight on what needs to be done.

Raniel and Andrew worked on the Tableau dashboard, designing visualizations that highlighted performance patterns, salary fairness and position based trends. Raniel created the framework of the worksheets and what contents the dashboard should be filled with for achieving insights with our data. Andrew worked on modifying the worksheets to make sure the contents were digestible to a viewer and arranging how the dashboard should look like. Andrew and Keon paired up on the Python metric script used to calculate derived values such as predicted salary and salary gap before loading the results into the warehouse. All five members collaborated on the final slide deck, reviewing the flow of the presentation, organizing speaking parts and making sure each component of the pipeline, warehouse and analytics was explained clearly. This shared workflow made sure the project stayed consistent across all stages and reflected contributions from the entire team.

XIII. Dashboard/Demo

Our Tableau dashboard brings the entire project together by allowing users to explore salary fairness, performance metrics, and trends for any NBA player across the five seasons. The dashboard shows whether a player was overpaid, underpaid, or fair-valued in each year and visualizes both their actual salary and the model-predicted deserved salary over time. It also includes a detailed metric panel that displays season-by-season changes in PER, TS%, USG%, MP, and other advanced statistics. Filters for player, team, and season make it easy to compare athletes or switch between analytical views. Overall, the dashboard transforms the outputs of our ETL pipeline and regression model into an interactive tool that highlights salary efficiency and player value in a simple and intuitive way.

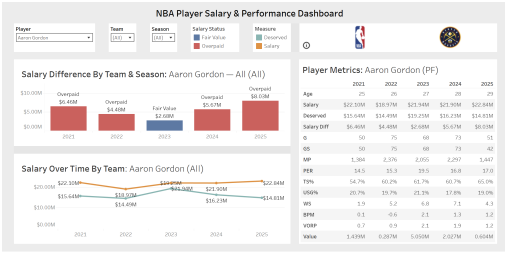

Fig. 7. Tableau Public NBA Player Value Model Dashboard.

XIV. Best Practices Learned

A. Validating data at every step

One best practice we picked up was to check the data constantly while it moved through NiFi instead of waiting until something broke at the end. Looking at the FlowFile content through Data Provenance helped us catch naming issues, missing fields and schema mismatches early which saved a lot of time and frustration.

B. Keeping transformations upstream

We also learned that it is much cleaner to handle all the messy transformations before loading anything into Redshift. Standardizing names, fixing types and cleaning rows inside NiFi made the warehouse far more stable and it meant that we did not have to fix problems after the data was already loaded.

C. Designing modular NiFi flows

Breaking the ETL into separate process groups for salaries, stats, season lookup and facts made the pipeline easier to reason about and test. Small, independent flows were much easier to debug compared to one large, tangled workflow.

D. Failing early is better than failing late

We learned that it was a good thing that things weren't working out early because it taught us that it's better to work on issues early on rather than finding out these issues when time is limited which would cause failure. We learned to use QueryRecord in small steps instead of writing long, complex queries inside one processor. Shorter transformations made errors surface immediately and simplified troubleshooting.

E. Keeping schema consistent across years

Because each season's CSV had small differences, we learned to standardize column names and data types early. Consistency prevented downstream failures and made model building much smoother.

F. Using external scripts when tools reach limits

Another best practice we learned was knowing when to step outside NiFi. Some joins were too inconsistent for LookupRecord to handle, so dropping in a simple Python script through ExecuteStreamCommand helped us finish the job cleanly. It taught us that sometimes the best solution is mixing tools instead of forcing one tool to do everything.

Appendix A
Appendix Overview

Appendix Contents

Appendix B
Code Walkthrough

We met the Code Walkthrough requirement by clearly explaining the two main scripts that handle the analytical work and the final transformation steps in our pipeline, as we described earlier in **Section XI** . In the walkthrough, we describe `join_nbaplayerval.ipynb`, the notebook that runs the whole analytical process. This includes loading the cleaned data set, selecting the seven advanced metrics used in the regression model, encoding categorical features, training the salary prediction model, and creating fields such as predicted salary, salary gap, and fair-value labels. The notebook also builds the summary tables and visualizations that appear in the analytics section and the Tableau dashboard.

We also explain `join_players.py`, which completes the final cleanup step that NiFi could not perform consistently. The walkthrough shows how NiFi triggers this script through the `ExecuteStreamCommand` processor, how the script reads the stats and salary files, fixes inconsistent column names, removes characters that caused Calcite SQL issues, standardizes data types, preserves the 2TM and 3TM season-total rows, removes placeholder values such as `9999`, and merges the cleaned stats and salary data into a single validated data set ready for Redshift.

By describing the purpose and logic of each script and showing how they operate within the larger ETL and warehouse workflow, we provided a clear overview of how the raw NBA files are transformed into the final analytical data set.

Appendix C
Demo

We satisfied the demo requirement by providing a complete and reproducible walkthrough of our entire system, covering every step from raw data ingestion to final analytics results. As described in **Section XIII**, our workflow demonstrates how Apache NiFi automatically pulls season files from S3, applies the required transformations, performs dimension lookups, and generates a thoroughly cleaned fact table that is loaded into Redshift. We also showcased the analytical SQL queries that run in the warehouse to reveal player value, salary fairness, and multi-season patterns.

We also met the demo requirements by building an interactive Tableau dashboard that visualizes the outputs of our ETL pipeline and regression model. The dashboard allows users to explore player performance, deserved salary, salary gaps, and trend comparisons across all five seasons. It is publicly available here:

NBA Player Value Model Dashboard

Together, pipeline artifacts, warehouse output, SQL results, and Tableau dashboard serve as evidence of a complete end-to-end demonstration that satisfies the requirements for this component.

Appendix D
Version Control

We satisfied the version control requirement by using GitHub to store, organize, and track all project files. Every component of the pipeline, including SQL queries, NiFi configuration exports, warehouse scripts, screenshots, and report drafts, was added and maintained directly in the GitHub repository. This provided a clear and traceable history of our work, ensured that all changes were backed up, and made collaboration significantly easier to manage. It is publicly available here:

NBA Player Value Model Github

Appendix E
Significance to the Real World

We satisfied this requirement by providing a clear explanation earlier in **Section V**, where we discussed why our work matters for NBA decision-making. In that part of the report, we show how contract evaluation impacts team building, competitive balance, and long term financial planning. Our analysis demonstrates how teams can spot undervalued players, avoid contracts that limit their flexibility, and make data-driven decisions rather than relying on reputation or basic stats. By linking our findings with real salary planning, roster construction, and talent evaluation, we showed that our project addresses the real business challenges facing NBA front offices.

Appendix F
Lessons Learned

We satisfied the Lessons Learned requirement by giving a clear discussion earlier in **Section VIII**, where we described the main technical insights we gained while building our multi-step ETL pipeline. In that part of the report, we explained how real issues such as FlowFile schema mismatches, strict

lookup rules, NiFi processor limits, and the need for Python based join workarounds shaped our understanding of data engineering systems. These lessons were meaningful because they came from real debugging, testing, and reworking parts of the pipeline.

Our article focuses on practical ideas such as schema drift, using provenance for debugging, NiFi's strict key-matching rules, and the value of combining NiFi with Python for complex transformations. By explaining how each problem was identified and resolved, we demonstrated a strong understanding of real pipeline behavior and fully met the requirement of thoughtful, substantial lessons learned.

Appendix G
Innovation

We met the Innovation requirement in two clear ways: through the tools we used and through the originality of our project idea. On the technical side, we intentionally built our pipeline with platforms that go beyond the usual class examples. We used Apache NiFi to automate multi-file ingestion, and Amazon Redshift as a cloud data warehouse. We also produced our deliverables with a professional toolkit, LaTeX for the written report, Tableau Public for interactive analytics, and Prezi for a visual presentation. This combination of tools reflects real data engineering practices and shows that we went beyond standard classroom workflows.

Our project idea itself is also innovative. We created a complete five-season NBA Player Value Model that integrates advanced performance metrics, salary data, ETL engineering, cloud warehousing, and business-focused analytics. Instead of simply analyzing a dataset, we designed an end-to-end workflow that cleans messy multi-season data, merges statistics with contract information, builds a regression-based value model, and produces insights that NBA teams could realistically use. This mix of technical depth and real-world applicability makes our project both creative and uniquely innovative compared to typical academic projects.

Appendix H
Teamwork

We met the teamwork requirement by dividing responsibilities clearly while still working together throughout the project. As described earlier in **Section XII**, each member took ownership of a core area, including the NiFi pipeline, Redshift warehouse setup, SQL analytics, Tableau dashboards, and Python metric calculations, and we coordinated through shared work sessions. Rish led the NiFi pipeline development and completed the warehouse schema and SQL queries, with Keon and Utkarsh helping troubleshoot processors and interpret the analytical results. Raniel and Andrew built and refined the Tableau dashboards, and Andrew and Keon worked together on the Python scripts used to calculate predicted salary and salary gap. All five team members reviewed the final slides, coordinated presentation roles, and made sure the ETL process, data model, and analytics were consistent. This collaborative approach shows that each person contributed meaningfully and that the team worked together effectively to complete the full end-to-end system.

Appendix I
Technical Difficulty

We satisfied this requirement by thoroughly documenting the technical challenges encountered during the project and explaining how each one shaped our final design decisions. As described earlier in **Section IX**, we provided a detailed account of issues in the raw NBA data, inconsistencies in player identifiers, schema mismatches in NiFi, and data type problems that caused load failures in Redshift.

Appendix J
Practiced Pair Programming

We met the pair programming requirement by working together on key parts of the project during shared Discord calls, as shown in the attached picture. In these calls, one member would share his screen while the others researched and helped troubleshoot the search. Together, we reviewed the code, fixed processor issues, and walked through the transformations together.

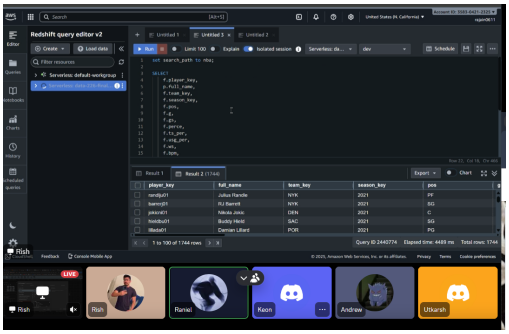

Fig. 8. Pair programming session on Discord.

Appendix K
Practiced Agile / Scrum (1-Week Sprints)

Our team followed an agile workflow, running one-week sprints from early October through mid-November. Each sprint included planning, task assignment, mid-sprint check-ins, and end-of-week reviews to evaluate progress and identify blockers. We documented every meeting in a shared Google Sheet, which served as our sprint log and evidence of continuous collaboration.

MEETING LOG		
Date:	Time On	Time off
10/7	2:00 PM	3:00 PM
10/9	1:00 PM	1:30:00 PM
10/12	1:00 PM	1:30:00 PM
10/14	1:00 PM	2:00:00 PM
10/18	1:00 PM	2:00:00 PM
10/19	1:00 PM	2:00:00 PM
10/23	1:00 PM	2:00:00 PM
10/28	2:00 PM	10:00 PM
10/30	2:00 PM	3:00 PM
11/6	2:14 PM	10:05 PM
11/12	4:23 PM	5:16 PM
11/13	12:56 PM	11:44 PM
11/14	12:20 PM	8:32 PM
11/15	3:36 PM	6:49 PM
11/17	12:07 PM	10:34 PM
11/18	4:59 PM	11:09 PM
11/19	1:05 PM	10:19 PM

Fig. 9. Team Meeting Log Used as Evidence of Weekly Sprints.

Appendix L
Slides

We satisfied the slide requirement by creating a complete and well-organized Prezi presentation that summarizes all major parts of the project. The slide deck covers the ETL pipeline, the NiFi workflow, the Redshift warehouse, the analytical SQL queries, the regression model, and the Tableau dashboards. It presents each part clearly and in a structured way. Using Prezi helped us highlight the flow of the project from raw data ingestion to final business insights, which made the presentation easy to follow and visually polished. All team members helped review and refine the slides to ensure that everything was accurate and consistent with the written report. The Prezi slides can be found here:

Prezi Slides

Appendix M
Used Unique Tools

We satisfied the requirement of using unique tools by incorporating several specialized platforms that extended beyond the standard technologies covered in class. First, the entire project report was written natively in LaTeX, not converted from Word or any other format. All sections, figures, tables, labels, and citations were authored directly in the source `.tex` file, following IEEE-style formatting guidelines. This shows familiarity with a professional technical writing tool commonly used in academic and industry research.

In addition to LaTeX, we use Apache NiFi as our primary ETL orchestration tool. NiFi was not taught in class, but was fully integrated into our workflow to automate the ingestion, transformation, lookup, and merging of multi-season NBA datasets. We also used Amazon Redshift as our cloud data warehouse and Tableau Public to build the final interactive dashboard. Furthermore, we created our project presentation in Prezi rather than standard slide-deck tools, leveraging its non-linear layout to clearly communicate the pipeline flow.

The combination of LaTeX, NiFi, Redshift, Tableau, and Prezi shows that we adopted a diverse set of tools to build, analyze, and present our end-to-end data engineering project, fully meeting the rubric requirement for using unique tools not covered in class.

Appendix N
Performed Substantial Analysis Using Data Warehouse and Pipeline Techniques

We met the requirement for substantial analysis by combining our warehouse pipeline with clear mathematical evaluation of player value. Once the five seasons of NBA data were cleaned, merged, and stored in Redshift, we used SQL and basic statistical reasoning to study how different performance measures relate to salary. This included looking at trends across seasons, comparing groups of players, checking how metrics move together, and examining patterns in contract value.

With the integrated dataset, we calculated results such as surplus value, salary patterns by position, contract efficiency by age, and the difference between expected pay and real pay. These outcomes were based on mathematical steps such as averages, correlations, grouped comparisons, seasonal changes, and a regression model that estimated what a player should earn based on performance. We also measured fairness in the market by comparing predicted salary to actual salary and identifying which players were regularly paid more or less than their performance suggested.

Because this work blended structured warehousing with real statistical thinking, it went far beyond simple summaries. The project formed a full analytical path that moved from ingestion and transformation to storage and quantitative evaluation, ending with insights that connect performance metrics, salaries, and long term patterns.

Below are the Analytical Queries that we ran:

```sql
-- Top 10 overpaid players with largest salary gap
SELECT DISTINCT p.full_name AS player,
CASE
    WHEN f.salary_gap < 0 THEN
        '-' || '$' || TO_CHAR(ABS(f.salary_gap) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(f.salary_gap / 1000000, 'FM999,990.00') || ' million'
END AS salary_gap_millions,
f.season_key AS season, f.team_key AS team
FROM fact_player_season AS f
JOIN dim_player AS p  ON p.player_key = f.player_key
WHERE f.overpaid_underpaid > 0
AND f.mp >= 750
AND f.team_key NOT LIKE '%TM%'
ORDER BY f.salary_gap ASC
LIMIT 10;
```

```sql
-- Top 10 underpaid players with largest salary gap
SELECT DISTINCT p.full_name AS player,
CASE
    WHEN f.salary_gap < 0 THEN
        '-' || '$' || TO_CHAR(ABS(f.salary_gap) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(f.salary_gap / 1000000, 'FM999,990.00') || ' million'
END AS salary_gap_millions,
f.season_key AS season, f.team_key AS team
FROM fact_player_season AS f
JOIN dim_player AS p  ON p.player_key = f.player_key
WHERE f.overpaid_underpaid < 0
AND f.mp >= 750
AND f.team_key NOT LIKE '%TM%'
ORDER BY f.salary_gap DESC
LIMIT 10;
```

```sql
-- Comparison of average deserved salary vs average salary gap by age
SELECT age,
CASE
    WHEN AVG(deserved_salary) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(deserved_salary)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(deserved_salary) / 1000000, 'FM999,990.00') || ' million'
END AS avg_value_millions,
CASE
    WHEN AVG(salary_gap) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(salary_gap)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(salary_gap) / 1000000, 'FM999,990.00') || ' million'
END AS avg_salary_gap_millions
FROM fact_player_season
GROUP BY age
ORDER BY age;
```

```sql
-- Team counts of overpaid/underpaid
WITH overpaid AS (
    SELECT team_key, COUNT(*) AS oc
    FROM fact_player_season
    WHERE overpaid_underpaid > 0
    AND mp >= 750
    AND team_key NOT LIKE '%TM%'
    GROUP BY team_key
),
underpaid AS (
    SELECT team_key, COUNT(*) AS uc
    FROM fact_player_season
    WHERE overpaid_underpaid < 0
    AND mp >= 750
    AND team_key NOT LIKE '%TM%'
    GROUP BY team_key
)
SELECT
    COALESCE(o.team_key, u.team_key) AS team,
    COALESCE(u.uc, 0) AS underpaid,
    COALESCE(o.oc, 0) AS overpaid
FROM underpaid AS u
FULL JOIN overpaid AS o ON o.team_key = u.team_key
ORDER BY team ASC;
```

```sql
-- Comparison of average deserved salary vs average salary gap by position and season
SELECT pos AS position, season_key as season,
CASE
    WHEN AVG(deserved_salary) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(deserved_salary)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(deserved_salary) / 1000000, 'FM999,990.00') || ' million'
END AS avg_value_millions,
CASE
    WHEN AVG(salary_gap) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(salary_gap)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(salary_gap) / 1000000, 'FM999,990.00') || ' million'
END AS avg_salary_gap_millions
FROM fact_player_season
GROUP BY pos, season_key
ORDER BY pos, season_key;
```

```sql
-- Season count of overpaid/underpaid
WITH overpaid AS (
    SELECT season_key, COUNT(*) AS oc
    FROM fact_player_season
    WHERE overpaid_underpaid > 0 AND mp >= 750
    GROUP BY season_key
),
underpaid AS (
    SELECT season_key, COUNT(*) AS uc
    FROM fact_player_season
    WHERE overpaid_underpaid < 0 AND mp >= 750
    GROUP BY season_key
)
SELECT
    COALESCE(o.season_key, u.season_key) AS season_key,
    COALESCE(u.uc, 0) AS underpaid,
    COALESCE(o.oc, 0) AS overpaid
FROM underpaid AS u
FULL JOIN overpaid AS o ON o.season_key = u.season_key
ORDER BY season_key ASC;
```

```sql
-- Comparison of average deserved salary vs average salary gap by position
SELECT pos AS position,
CASE
    WHEN AVG(deserved_salary) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(deserved_salary)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(deserved_salary) / 1000000, 'FM999,990.00') || ' million'
END AS avg_value_millions,
CASE
    WHEN AVG(salary_gap) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(salary_gap)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(salary_gap) / 1000000, 'FM999,990.00') || ' million'
END AS avg_salary_gap_millions
FROM fact_player_season
GROUP BY pos
ORDER BY pos;
```

```sql
-- Comparison of average deserved salary vs average salary gap by season
SELECT season_key AS season,
CASE
    WHEN AVG(deserved_salary) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(deserved_salary)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(deserved_salary) / 1000000, 'FM999,990.00') || ' million'
END AS avg_value_millions,
CASE
    WHEN AVG(salary_gap) < 0 THEN
        '-' || '$' || TO_CHAR(ABS(AVG(salary_gap)) / 1000000, 'FM999,990.00') || ' million'
    ELSE
        '$' || TO_CHAR(AVG(salary_gap) / 1000000, 'FM999,990.00') || ' million'
END AS avg_salary_gap_millions
FROM fact_player_season
GROUP BY season
ORDER BY season;
```

APPENDIX O
USED A NEW ETL OR DATA WAREHOUSE TOOL NOT COVERED IN CLASS

We met this requirement by integrating Apache NiFi into our workflow, even though it was not assigned in class. As described in **Section VI**, NiFi was used to build a fully automated pipeline that extracted raw NBA season files from S3, applied record-level transformations, performed lookups, merged outputs, and loaded the final cleaned data into Redshift. We demonstrated our understanding of this new tool through screenshots, processor configurations, and flow design evidence submitted on Canvas. These artifacts show that we were able to independently learn and apply NiFi's enterprise-grade features, such as provenance tracking, schema validation, and flow-level orchestration—thereby satisfying the rubric requirement for using a new tool not covered in the course.

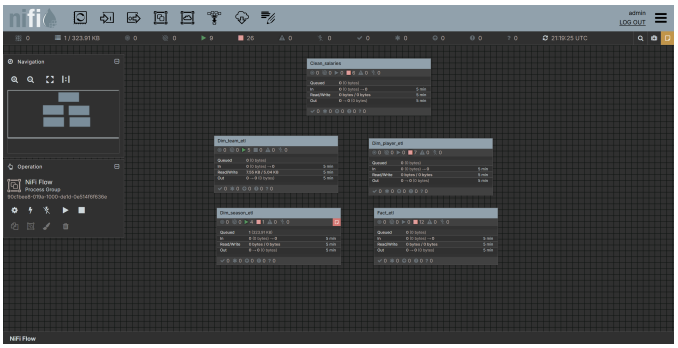

Fig. 10. Apache NiFi pipeline used for Extract–Transform–Load in our project.

APPENDIX P
DEMONSTRATED HOW ANALYTICS SUPPORT BUSINESS DECISIONS

We met the requirement to demonstrate how analytics support business decisions by including a whole section titled "Analytical Queries and Business Insights," where each query addresses a real NBA decision-making question. As described in **Section X**, we use SQL queries in the Redshift warehouse to analyze roster value, contract fairness, positional salary patterns, season-to-season market efficiency, and which players were underpaid or overpaid according to performance. These analyses highlight practical insights such as which teams benefit the most from surplus value, which veteran contracts carry financial risk, which age groups deliver the best return on investment, and which positions tend to be mispriced by the market. Each query includes a narrative explanation, screenshots, and dashboards that show how analytics directly support smarter financial and strategic decisions in the NBA.