THE BUCHAREST UNIVERSITY OF ECONOMIC STUDIES

DATABASE SYSTEMS JOURNAL

Vol. XVII/2026

LISTED IN

RePEc, EBSCO, DOAJ, Open J-Gate,
Cabell's Directories of Publishing Opportunities,
Index Copernicus, Google Scholar,
Directory of Science, Cite Factor,
Electronic Journals Library

BIG DATA
NoSQL
ARTIFICIAL INTELLIGENCE
MACHINE LEARNING
BUSINESS INTELLIGENCE
IoT
CLOUD COMPUTING
BLOCKCHAIN
DATA WAREHOUSES
DATABASES

ISSN: 2069 – 3230
dbjournal.ro

Contact
CaleaDorobanţilor, no. 15-17, room 2017, Bucharest, Romania
Web: https://dbjournal.ro/
E-mail: editordbjournal@gmail.com

CONTENTS

A Hybrid Approach to Real-Time Recommendations using Heterogeneous Data Processing and Distributed Predictive Models

Diana – Andreea CĂUNIAC, Simona-Vasilica OPREA
Bucharest University of Economic Studies,
Bucharest, Romania
diana.cauniac@csie.ase.ro, simona.oprea@csie.ase.ro

As e-commerce platforms scale, the gap between what users expect and what static recommendation models can deliver has become hard to ignore. This paper describes the design and implementation of a hybrid recommendation system built on a distributed cloud infrastructure, tested on a dataset of over 3.7 million products, 1.9 million reviews, and 1.1 million user interactions. The system combines collaborative filtering through matrix factorization with sentiment analysis, emotion detection, and topic modeling applied to user reviews, identifying six recurring themes that reflect real purchasing experiences. Geographic proximity and recent behavioral signals are incorporated as contextual features to further refine recommendations. The stack includes Google BigQuery, Vertex AI Feature Store, Pinecone, Apache Kafka, Apache Flink, and BigQuery ML. A feedback loop ties user interactions back to model updates, keeping recommendations relevant as behavior changes. Results suggest that decoupling the processing pipelines reduces latency without sacrificing recommendation quality.

Keywords*: recommendation systems, real-time processing, hybrid architecture, collaborative filtering, heterogeneous data, sentiment analysis, topic modeling, distributed computing.*

1 Introduction

Recommendation systems have become a core part of modern e-commerce platforms, directly influencing conversion rates and user retention [1]. As consumer behavior grows more dynamic, the relevance of a recommendation depends heavily on the system's ability to react quickly to new interactions, rather than relying on scheduled model updates [2], [3].

Traditional batch-based approaches process data at fixed intervals and perform reasonably well in stable environments, but struggle to keep up when user preferences shift mid-session. When a user suddenly explores an unfamiliar product category, the system needs to pick up on that signal immediately, something static models are not built to do.

Moving toward distributed architectures capable of ingesting continuous event streams becomes a natural response to this limitation [3], [4]. The challenge is further complicated by the diversity of available data: behavioral interactions, unstructured textual reviews, geographic signals, session contexts, and device metadata, each requiring a different processing approach [5], [6].

To handle these complexities, we propose a hybrid recommendation system deployed on a scalable cloud infrastructure, designed to process large volumes of heterogeneous data with low latency. The framework separates concerns across three functional layers (online, nearline, and offline) allowing the system to serve requests in real time while continuously updating features and periodically retraining models, without disrupting live service.

Rather than relying solely on purchase histories and clickstream data, the system combines collaborative filtering, semantic analysis, and topic modeling within a unified processing pipeline [7]. The results show that real-time responsiveness does not have to come at the cost of analytical depth or recommendation accuracy [8].

2 Data Modeling and Preprocessing

Data preparation is a critical prerequisite for any functional recommendation system. The dataset combines real sources, Hugging Face, Open Food Facts, and OpenStreetMap, with simulated data, reaching over 3.7 million products, 1.9 million reviews, 1.1 million interactions, and 250,000 user profiles. This combination was designed to ensure sufficient volume and diversity for testing system stability across varied usage scenarios.

The data model is built around two core entities, clients and products, connected through user sessions. Additional layers capture device metadata, geographic coordinates, textual reviews, and recommendation history. Products define the item space from which all recommendations are drawn. Interactions record explicit user actions, while reviews contribute qualitative signal that behavioral data alone does not carry. Sessions and locations introduce temporal context and geographic proximity as ranking features. The entities, their data types, and approximate volumes are summarized in Table 1.

Table 1. Dataset entities and approximate record volumes

Entity	**Data Type**	**Approximate Volume (Records)**	**Content Description**
Products	Structured	3,700,000+	Aggregated product metadata from diverse sources.
Clients	Semi-structured	250,000+	Comprehensive user profiles and demographic attributes.
Interactions	Structured	1,100,000+	Behavioral user events (e.g., clicks, views, add-to-cart actions).
Reviews	Unstructured	1,900,000+	Unstructured textual product reviews and semantic feedback.
Sessions	Semi-structured	500,000+	Browsing sessions enriched with contextual metadata.
Devices	Structured	1,000+	Hardware specifications and client device metadata.
Locations	Geospatial	4,783+	Geographic coordinate data for commercial locations.
Recommendations_log	Structured	1,000,000 (Simulated)	Historical audit trail of delivered recommendations.

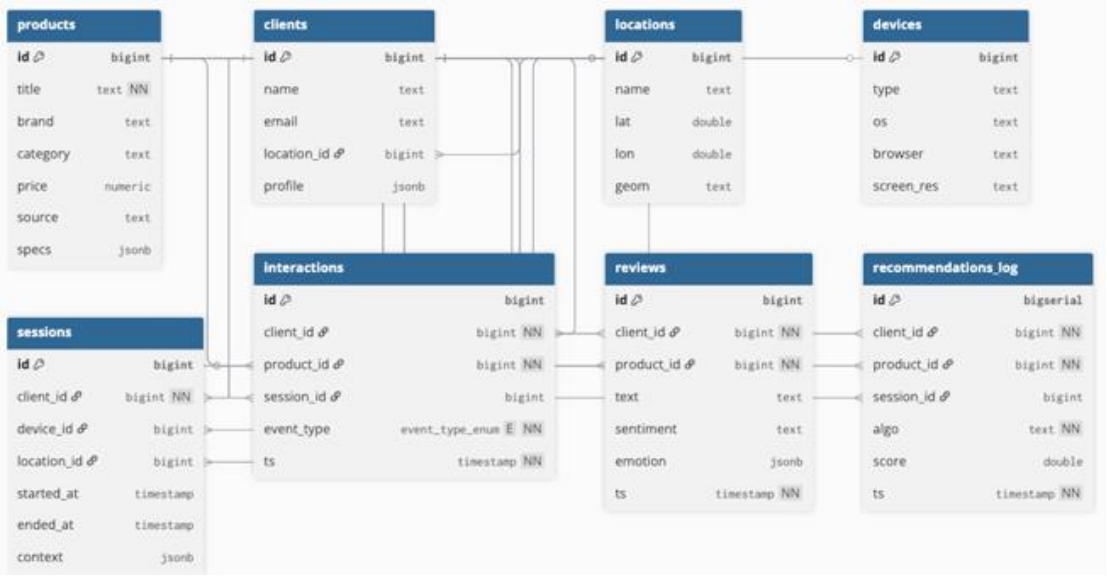


Fig. 1. Database schema and entity relationships

All data flows through Google BigQuery, which handles storage and computation within the same environment, allowing machine learning models to be called directly through SQL without moving data between systems. Data is organized into three zones: raw ingestion, curated transformations, and a semantic layer built from views used in both analysis and model training. The relational structure connecting these entities is illustrated in Figure 1.

Preprocessing is applied separately to each entity. In the products table, records missing both name and category are excluded, as these fields are required for semantic placement within the recommendation space. Missing prices are estimated using the category median: $pc = \text{median}(pi \mid i \in c)$, values deviating significantly from category norms are identified via z-scores and either adjusted or removed. Interactions falling outside their corresponding session window are discarded [9]. Incomplete client profiles are reconstructed from distributions observed among users with similar attributes. Numerical variables are normalized using min-max scaling. Session durations are standardized through z-score scaling, given their skewed distribution.

Textual reviews are cleaned and encoded into numerical representations using NLP models in BigQuery ML. Geographic records are filtered to Romanian territory, and locations in close proximity are merged to prevent duplicate entries from affecting distance calculations.

The preprocessed data is collected into a curated layer, with events ordered chronologically and semantic views built to support consistent feature extraction across all models used in later stages.

3 Sentiment Analysis and Topic Modeling in Product Evaluation

User reviews contain information that behavioral data alone does not capture [10]. A product frequently viewed but consistently criticized carries a different signal than one with similar interaction patterns and positive feedback [11]. The system addresses this through sentiment analysis and topic modeling applied directly to review text [12].

Before analysis, reviews are cleaned to remove special characters, inconsistent casing, and symbols with no semantic value. Sentiment is quantified using TextBlob, which assigns each review a polarity score: $\text{Polarity} = (\Sigma\ w\_positive - \Sigma\ w\_negative) / \text{Total words}$ [13].

The score ranges from −1 to 1, where negative values indicate dissatisfaction and positive values indicate approval. The computed sentiment score is distinct from

the explicit star rating (review_score, on a 1–5 scale), the two are correlated but measure different aspects of user feedback. Table 2 includes a case where a 4-star rating accompanied a description of a damaged product, illustrating this gap directly.

Table 2. Examples of Semantic Evaluation and Sentiment Scores for Reviews from the Dataset

review_id	fragment review_text	review_score	s_i^sent (interpretation)
RV00315788	"Very disappointed, I do not recommend."	1.0	strongly negative
RV00140820	"Poor packaging, low quality."	1.0	negative
RV00371553	"The product arrived damaged."	4.0	moderately negative
AMZ00132163	"Excellent product, I am very satisfied."	5.0	strongly positive
AMZ00721273	"Very satisfied with the product, I recommend it."	5.0	positive

A polarity score captures overall sentiment direction, but reviews frequently reference specific emotional states, disappointment, satisfaction, frustration, that a single value does not distinguish [10]. The emotion field in the dataset records this additional dimension, and Table 3 illustrates cases where similar polarity scores correspond to different emotional categories.

Table 3. Examples of Emotions Identified in Reviews from the Dataset

review_id	fragment review_text	emotion
RV00315788	"Very disappointed, I do not recommend."	disappointment
RV00140820	"Poor packaging, low quality."	neutral
RV00371553	"The product arrived damaged."	disappointment
AMZ00132163	"Excellent product, I am very satisfied."	satisfaction
AMZ00721273	"Very satisfied with the product, I recommend it."	satisfaction

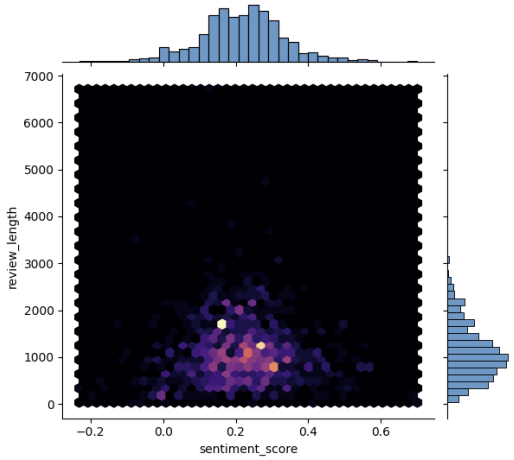


Fig. 2. Correlation between sentiment polarity and review length

The sentiment distribution across the dataset was analyzed through a hexbin plot, shown in Figure 2. Scores concentrate between 0.1 and 0.3, consistent with the tendency observed on e-commerce platforms for users to leave feedback more frequently when satisfied. Review length falls predominantly between a few hundred and 1,500 characters, sufficient for NLP models to

extract reliable semantic signal.

Product-level scores are computed by averaging across all associated reviews: $S(p) = (1 / N_p) \cdot \Sigma s_i$, where N_p is the number of reviews for product p. Arithmetic averaging reduces the influence of isolated extreme ratings on the aggregate score. The resulting values are subsequently combined with behavioral signals in the recommendation ranking stage.

4 Topic Modeling from User Reviews

Sentiment scores capture opinion polarity, but users frequently reference specific product aspects, quality, price, delivery, or functionality, within the same review. Topic modeling extracts these recurring themes and structures them into categories that the sentiment analysis alone cannot distinguish.

Each review is projected into a lower-dimensional latent space where each dimension corresponds to a recurring theme. The projection produces a probability distribution over topics, allowing a single review to be associated with multiple themes at varying intensities. At the product level, topic distributions are averaged across all associated reviews, producing a vector that describes the thematic structure of feedback for each product.

Six topics were identified across the review corpus: quality, delivery, pricing, packaging, usability, and customer support. Quality-related content carries the highest weight at 0.20, with the remaining topics spread more evenly across the corpus. The full distribution is summarized in Table 4.

Table 4. Identified Topics and Average Distribution

Topic	**Representative Terms**	**Average Distribution**
Topic 1	product, quality, good, excellent, recommend	0.20
Topic 2	delivery, shipping, arrival, delayed, damaged	0.13
Topic 3	price, value, worth, cheap, expensive	0.16
Topic 4	packaging, box, damaged, condition, arrival	0.18
Topic 5	usage, easy, works, functionality, performance	0.17
Topic 6	support, service, return, refund, customer	0.16

Figure 3 shows the relative weights across all six topics.

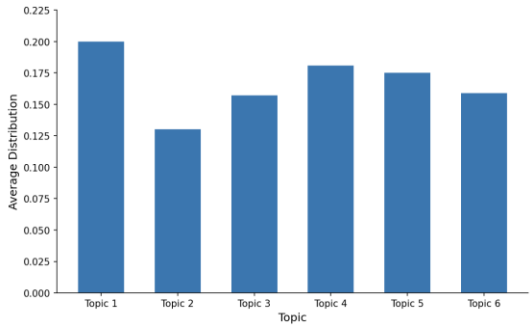


Fig. 3. Topic distribution across user reviews

t-SNE was applied to project the topic distributions into two dimensions, shown in Figure 4. Reviews covering similar themes form distinct clusters, and those near cluster boundaries tend to reference multiple product aspects within the same text.

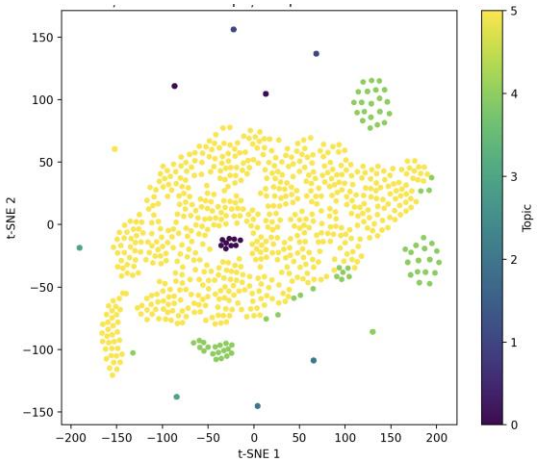


Fig. 4. t-SNE projection of review topic distributions

5 Hybrid Recommendation Pipeline

Collaborative filtering works well for users with rich interaction histories, but

breaks down in cold-start scenarios. Semantic analysis helps fill that gap, though it struggles to capture rapid shifts in user behavior. The system described here combines both approaches within a continuous processing pipeline that also incorporates live session context, geographic signals, and a feedback loop that ties user reactions back to model updates [14].

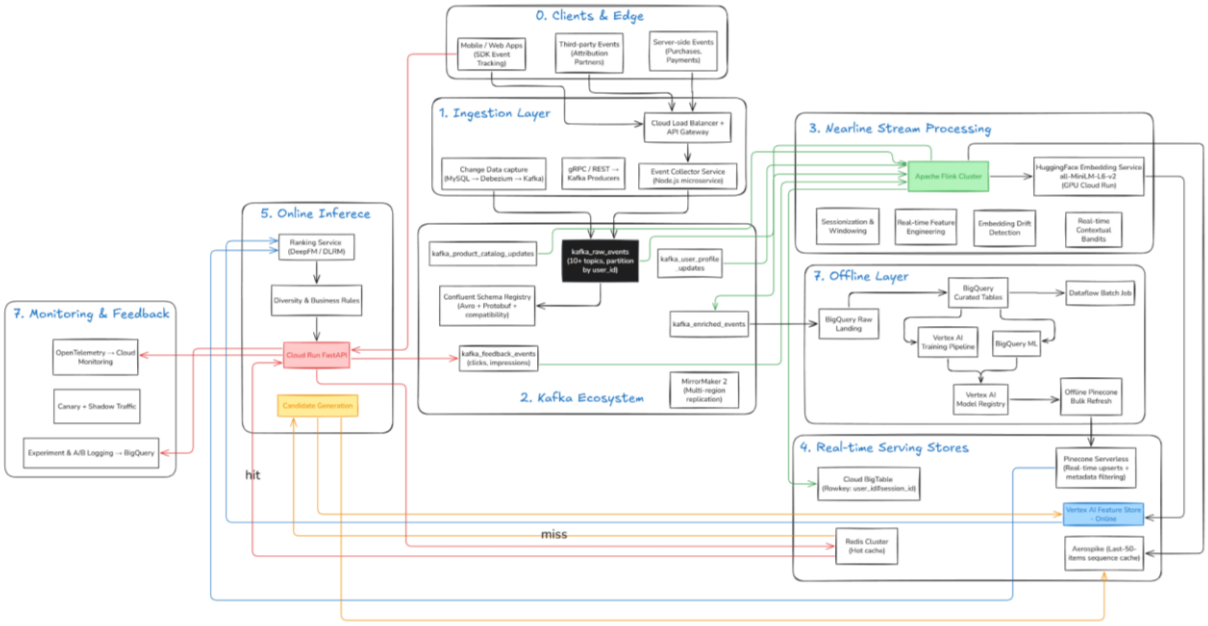


Fig. 5. End-to-end architecture of the hybrid recommendation system

The end-to-end architecture is illustrated in Figure 5, covering all eight stages of the data flow.

Stage 1 — Edge and Client Layer. Web and mobile applications emit behavioral signals via embedded SDKs. Third-party platforms contribute through integration connectors, while server-side systems append operational data on inventory and transactions. All incoming events pass through a Cloud Load Balancer and API Gateway before entering the pipeline.

Stage 2 — Ingestion. Raw events are filtered, validated, and normalized into a standardized format. Change Data Capture via Debezium handles relational database changes without affecting source systems. REST and gRPC events are processed by Node.js microservices, and all messages are validated against a Schema Registry before moving further.

Stage 3 — Message Broker (Apache Kafka). Kafka organizes streaming events into topics by type: user interactions, product catalog updates, and profile changes. The kafka_enriched_events topic associates each event with product attributes, user profile metadata, and session context. MirrorMaker handles cross-region replication for availability and fault tolerance.

Stage 4 — Nearline Processing. Apache Flink consumes streams from Kafka and computes interaction frequency, session duration, recency scores, and category-level interest shifts, handling deduplication and profile updates in parallel. Outputs are written to Cloud Bigtable for low-latency access, and textual data is routed to HuggingFace models for embedding generation.

Stage 5 — Feature Store and Vector Management. Embeddings are written to Vertex AI Feature Store and Pinecone. The Feature Store maintains the numerical and vector representations used by predictive models, and Pinecone manages semantic indexing and real-time similarity searches.

Stage 6 — Offline Training. Historical data in BigQuery is used to periodically retrain

collaborative filtering, semantic analysis, and reranking models. BigQuery ML runs training directly through SQL, removing the need for separate ML infrastructure. Updated models are published to the Vertex AI Model Registry and deployed without taking the service offline.

Stage 7 — Online Inference. On receiving a recommendation request, the system first queries Redis for cached results. On a cache miss, a FastAPI microservice retrieves the current user profile from the Feature Store and runs a vector similarity search in Pinecone. Candidate products are scored by a reranking model using CTR, interaction frequency, and historical signals, then returned to the client through Cloud Endpoints.

Stage 8 — Feedback Loop. Every post-recommendation action, click, ignore, purchase, or abandonment, is fed back into Kafka. Flink correlates this feedback with the recommendations that triggered it, forwarding aggregated metrics to BigQuery for model evaluation and retraining. Refined models are versioned in Vertex AI and deployed automatically to Cloud Run, the cycle runs without manual steps.

6 Model Adaptability and Dynamic Updating

Recommendations become less relevant the moment user preferences shift and the model stops keeping up. The feedback mechanism described here closes that gap by routing post-recommendation user reactions back into the training pipeline, so behavioral changes are reflected in the models without waiting for a scheduled retraining cycle.

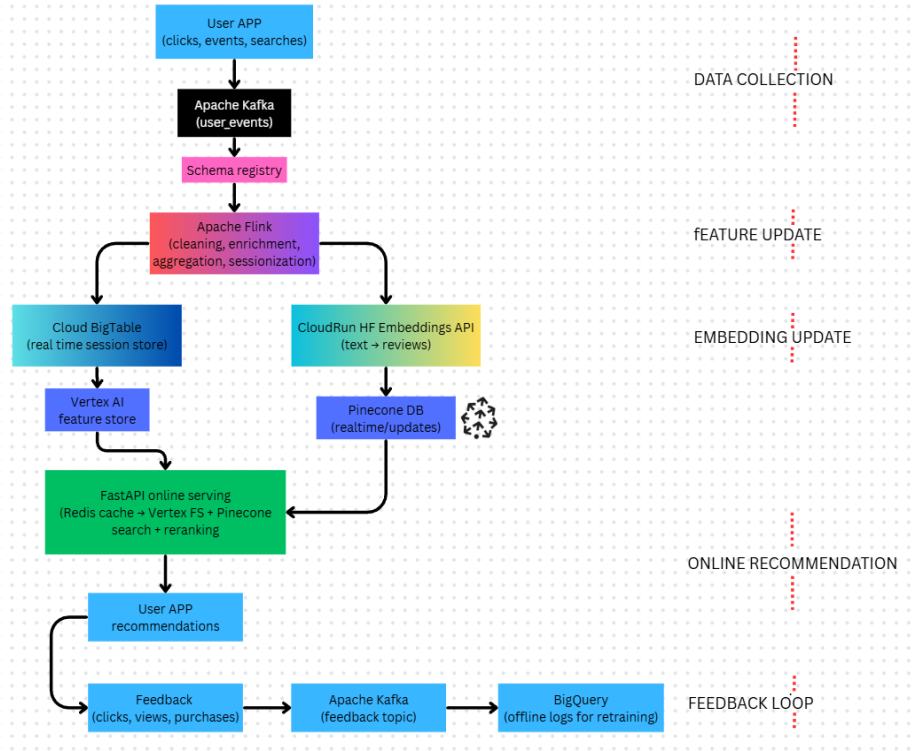


Fig. 6. Real-time feedback loop and model retraining flow

The full flow is illustrated in Figure 6, after a recommendation is displayed, the client application captures the user's response and publishes it to a dedicated Kafka topic. Messages pass through a Schema Registry that validates structure and discards incomplete or malformed payloads before they reach the processing layer.

Apache Flink picks up the validated streams and correlates displayed recommendations with subsequent user

actions, distinguishing what was clicked or purchased from what was ignored. From these associations, three metrics are computed: click-through rate, conversion rate, and an aggregate relevance score.

The aggregated results are written to BigQuery across two tables — feedback_log and recommendations_log — which together maintain a full history of user-system interactions. Model versions are also compared here using live behavioral data rather than held-out test sets.

As feedback accumulates, ranking and predictive models are retrained through BigQuery ML. Updated versions are registered in the Vertex AI Model Registry with versioning metadata and deployed automatically to Cloud Run, with no manual steps required.

Concurrently, user and product feature profiles are updated in Vertex AI Feature Store and Pinecone. If a product category gains traction, the rise in CTR propagates through the nearline pipeline and updates the associated vectors before the next offline retraining cycle, without waiting for a full model rebuild.

7 Conclusions

The results confirm the initial premise: a recommendation system can scale across heterogeneous data without sacrificing response speed for analytical depth.

Collaborative filtering covers users with rich interaction histories, while semantic signals and topic distributions help where behavioral data is sparse. Geographic proximity and session context fill in what clickstream data misses. None of these components works well in isolation, the value comes from running them together within a continuous pipeline.

Working with over 3.7 million products, 1.9 million reviews, and 1.1 million interactions showed that decoupling the processing layers has a direct impact on both latency and model freshness. The nearline layer keeps user profiles current between full retraining cycles, while the offline component handles periodic rebuilds on complete historical data.

The feedback loop tied everything together. CTR and conversion signals propagate through Kafka and Flink before the next scheduled training run, so the models stay aligned with recent behavior without waiting for a full retraining cycle.

Some questions remain worth pursuing. Adapting the spatial ranking signals for cross-border e-commerce scenarios would be a logical next step beyond the Romanian location data used here. Users with no interaction history still represent a harder cold-start case than semantic signals alone can fully address. Over longer sessions, more explicit diversity constraints in the reranking models could also help reduce filter bubble effects.

8 Acknowledgment

This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI - UEFISCDI, project number COFUND-CETP-SMART-LEM-1, within PNCDI IV.

References

[1] M. Alam, “Hybrid Recommender Systems: A Systematic Review,” Information Processing & Management, 2024.

[2] Y. Deldjoo, S. Saghafian and A. Bellogín, “Recommender Systems Leveraging Contextual Information: A Survey,” Information Fusion, pp. 101-120, 2024.

[3] L. Chen and Y. Zhang, “Explainable Recommender Systems: A Survey and New Perspectives,” Information Processing & Management, pp. 1-20, 2024.

[4] S. Zhang, “Deep Learning Based Recommender Systems: A Survey and New Perspectives,” ACM Computing Surveys, 2022.

[5] V. Kumar and S. Patel, “Context-Aware

Recommender Systems Using Big Data Analytics," Future Generation Computer Systems, pp. 88-92, 2023.
[6] J. Li and W. Wang, "Real-time Recommendation Systems: Architectures and Challenges," IEEE Access, pp. 678-692, 2024.
[7] P. Carbone, "Flink: Stateful Stream Processing," IEEE, pp. 5-12, 2023.
[8] J. Kreps, "Apache Kafka and Stream Processing Architectures," LinkedIn Engineering, pp. 10-12, 2022.
[9] N. Marz and J. Warren, "Lambda and Kappa Architectures Revisited," IEEE Software, pp. 30-40, 2023.
[10] B. Pang and L. Lee, "Opinion Mining and Sentiment Analysis," Foundations and Trends in Information Retrieval, pp. 1-36, 2008.
[11] D. Tang, Q. Qin and T. Liu, "Document Modeling with Gated Recurrent Neural Network for Sentiment Classification," Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 1422-1432, 2015.
[12] Y. Zhang and A. Zhang, "Joint Representation Learning for Top-N Recommendation with Review Text," Proceedings of the 13th ACM International Conference on Web Search and Data Mining, pp. 345-353, 2020.
[13] J. McAuley and J. Leskovec, "Using Reviews for Recommendation Systems," RecSys Proceedings, pp. 150-155, 2022.
[14] A. Group, "Alibaba Real-Time Recommendation System Architecture," Alibaba Cloud Whitepaper, pp. 11-19, 2024.

Diana-Andreea CĂUNIAC earned her bachelor's degree in Economic Informatics in 2020 and her master's degree in Databases – Support for Business in 2022. She is currently pursuing a Ph.D., focusing on big data in real time. Professionally, she worked in the field of databases and currently works as a Fullstack Developer. Her research interests include big data, database systems, software development, artificial intelligence, and modern web technologies.

Simona-Vasilica OPREA received the MSc degree through the Infrastructure Management Program from Yokohama National University, Japan, in 2007, the first Ph.D. degree in Power System Engineering from the Bucharest Polytechnic University in 2009, and the second Ph.D. degree in Economic Informatics from the Bucharest University of Economic Studies in 2017. She is currently an Associate Professor with the Faculty of Cybernetics, Statistics, and Economic Informatics with the Bucharest Academy of Economic Studies, involved in several research projects.

Data Preparation with Oracle Cloud

Cristiana COSTAN
Bucharest University of Economic Studies
Faculty of Cybernetics, Statistics and Economic Informatics
Bucharest, Romania
costancristiana21@stud.ase.ro

This article examines the Extract-Transform-Load (ETL) process as an essential method for preparing data from warehouses for analysis. It also covers the difficulties of combining data from numerous external sources, converting it for consistency, and storing it in the cloud for effective querying. Oracle Cloud Infrastructure (OCI) provides a range of services for ETL workflows, including Oracle Data Integrator (ODI), which offers a variety of code-based data processing options, and Data Transforms, also known as ODI Web, a modern no-code solution designed to simplify and automate data transformations and machine learning tasks. A practical example using a League of Legends dataset illustrates how Data Transforms and Autonomous Data Warehouse can be used for effective data preparation and analysis. The goal of this paper is to assist data scientists in simplifying and improving data workflows through the use of modern cloud ETL solutions.
Keywords: *Data Warehouse, ETL, Oracle Cloud, Data Integration, Machine Learning*

1 Introduction

Although the term itself had not yet been formally established, individuals have employed ETL-like processes since the early days of database systems. To extract data and make it suitable for analysis, it was necessary to process it into a format more appropriate for analytical tasks. These transformations can vary, ranging from duplicate removal and missing value replacement to the application of functions for generating new statistics. This process is frequently employed in data warehouses, which usually handle substantial amounts of data that are organized along several dimensions, the most significant of which is the temporal component.

Such data is usually sourced from a wide range of external systems such as APIs and must be cleaned and transformed before it can be integrated into databases for analytical use. Therefore, ETL or its more recent variant ELT are employed for extracting, transforming, and uploading data into various storage systems, or for extracting and loading data into the data warehouse and performing transformations directly within it.

OCI provides cloud-based services that facilitate both ETL and ELT workflows. Oracle Data Integrator, also known as ODI, offers extensive options for data processing. Data Transforms, another service from the OCI stack built on the ODI infrastructure, is a no-code solution capable of executing these processes as well as additional features such as machine learning model generation and graphical outputs. If needed, the code required for the transformations designed in the Data Transforms flow can be automatically generated at runtime. This represents a modern cloud-native approach that enables data warehouse processing without the need to write code manually.

2 Data Warehousing

"A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions." [1]. Depending on the specific needs of the business maintaining it, data warehouses can be utilized for storing and processing granular data for a wide range of analytical and strategic purposes.

The most important characteristic of the data warehouse is its integration. Multi-source data undergoes transformation processes in

order to become a unified corporate data representation. As the data is being loaded into the warehouse, these processes typically involve data conversion and summarization in order to guarantee consistency among datasets.

The variety of data sources and the particular needs of organizations make each implementation and application of data warehouses different. In general, data is loaded into the warehouse in bulk. Another important characteristic of data warehouses is time variance: each record is associated with a temporal element, such as a timestamp or transaction date, facilitating trend monitoring and historical analysis.

Key characteristics of a data warehouse include the structuring of data to facilitate ease of access and high-speed querying, the inclusion of large volumes of historical data, and the loading of data from multiple sources involving various transformations [2]. Additionally, data warehouse's architecture can be customized for different groups within the organization by adding data marts, which are systems designed for a particular line of business. These can be utilized to analyze historical data or make predictions for certain branches separately.

Data warehouses also support on-line analytical processing (OLAP), an essential element in decision support [3]. To enable complex analytics and visualization, data within the warehouse is typically organized in a multidimensional format, with time often being one of the primary dimensions. These dimensions commonly exhibit hierarchical structures, allowing users to explore data at different levels of granularity. Rollup (increasing aggregation level), drill-down (decreasing aggregation level or increasing detail), slice-and-dice (selection and projection), and pivot (altering the multidimensional perception of data) are examples of common OLAP procedures along one or more dimension hierarchies.

3 Overview of ETL and ELT

Originating in the early days of database systems, the Extract-Transform-Load (ETL) process was initially thought of as a programming task without a specific name or understanding of its importance [4]. Although ETL activities existed before the 2000s, it was only around that time that ETL began to be clearly defined and mentioned in well-known books. Any data processing software that modifies or filters records, performs calculations, and loads data into a different storage system than the original one can be considered a form of ETL.

ETL involves three sequential stages [5]. Data is first extracted from various sources, commonly in formats such as comma-separated values (CSV), Excel (XLS), text files (.txt), JSON, or retrieved through application programming interfaces (APIs). In the transformation stage, the extracted data undergoes a series of operations including data cleaning, normalization, duplicate removal, filtering, sorting, grouping, and the application of various functions if needed. After being transformed, the data is then imported into a specific system, such as a database, data warehouse, or data mart, in order to optimize further analytical processing.

Extract-Load-Transform (ELT) represents an alternative data-processing approach for analytical purposes, where data is initially loaded in its raw form and transformed directly within the target system [6]. Both ETL and ELT contain the same fundamental steps; however, the key difference lies within the sequence and location of the transformation step. In ETL, raw data is processed before being loaded into the target database, whereas in ELT, data is loaded directly into the data warehouse and transformed afterwards. Consequently, ETL involves moving data only once it has been fully transformed, while ELT permits multiple transformations on the data after it has been loaded.

4 Data Preparation Techniques for Machine Learning

Analysts often encounter various requirements depending on the characteristics of the data, since each dataset used to create machine learning models is naturally distinctive [7]. Regardless of the data source, the data preparation process includes general operations that are often required in addition to these specific requirements. These procedures typically include data cleaning, feature selection, data transformation, feature engineering, and dimensionality reduction.

Data cleaning, also known as data cleansing, is the process of correcting or removing inaccuracies, inconsistencies, or incomplete values from a dataset. This can consist of removing outliers, identifying and eliminating duplicates, or replacing missing values using statistical methods such as the mean, predicted values generated by models, or by utilizing values that significantly exceed a specific range.

Feature selection requires identifying the most relevant elements that influence the prediction of the target variable. This phase is essential for simplifying the model, reducing overfitting, and enhancing interpretability.

Data transformation handles variable type conversion and data scaling. Variables may be numeric (such as integers or floating-point numbers) or categorical (such as nominal, ordinal, or Boolean types). These transformations include the following conversions: numeric variables into ordinal ones ("Discretization Transform"), categorical values into integers ("Ordinal Transform"), or categorical variables into binary vectors ("One Hot Transform"). This step may also include normalization, which rescales the data to a range between 0 and 1, and standardization, which modifies values to follow a normal distribution.

Feature engineering is the process in which new variables are created from already existing data for enhancing the performance of machine learning models. Combining characteristics, computing statistics from variables, or creating dummy variables are some of the methods utilized to generate significant attributes. During this step, complex variables can be simplified by division into smaller parts or already existing variables can be extended.

Dimensionality reduction serves as an alternative to feature selection, lowering the quantity of input features while maintaining the data's fundamental structure and variance. Commonly employed methods for achieving this include Principal Component Analysis (PCA) and Singular Value Decomposition (SVD).

Another critical concern during data preparation is data leakage. This occurs whenever the preparation techniques are applied to the entire dataset prior to splitting the dataset into training and testing subsets. As a result, data from the test set can unintentionally influence the training process. For example, when calculating the mean for substituting missing values, the train set can include a mean calculated based on both sets if the splitting does not occur before this operation. Thereby, performance results may be overestimated since the model caught a part of the test data when training. Data preparation procedures should be applied independently to the training and testing data after the initial split in order to prevent data leakage.

5 Transforming Data with Oracle Cloud

Oracle Cloud Infrastructure (OCI) is a suite of cloud services for running secure, highly available applications with flexible storage [8]. For cloud-native development, OCI includes services such as API management, machine learning, serverless functions, autonomous and MySQL databases, as well as deployment, monitoring, and observability tools [9].

OCI's autonomous databases provide integrated data warehousing and analytics capabilities that support machine learning workflows [10]. Both data warehouses and data lakes are accessible through OCI's data warehouse services; the latter facilitates data

processing and analytics by incorporating built-in artificial intelligence, machine learning, graph, spatial, and other advanced features [11]. Data scientists can efficiently design and implement machine learning models within Oracle Autonomous Data Warehouse through the use of Oracle Machine Learning, which provides Python and SQL notebooks along with an AutoML user interface.

While Oracle Machine Learning is a powerful tool for in-database modeling, this article focuses on the data transformation and preparation stages within Oracle Cloud. The cleaned and transformed data will be accessed directly from the Autonomous Data Warehouse through Python for external machine learning analysis.

OCI Data Integration is primarily used by administrators, data engineers, ETL developers, and operators [12]. Since this article focuses on data transformation for machine learning, data engineers and ETL developers, who are responsible for developing, building, and testing data integration solutions, are of primary importance. The first step in the process involves data extraction from a data entity, such as a database table, view, or file. After that, various operators can be utilized to represent input sources, output targets, or transformations within the data flow. The pipeline is defined as a task, which can be executed to extract, transform, and load data into the database [13]. This flow of data between source and target datasets, including any operations performed, is defined by Data Flows, which are a component of Data Integration [14].

In Data Integration, the first step in designing a data flow involves selecting appropriate data operators, such as the source operator, which provides the input data, and the target operator, which defines the destination for the transformed data [15]. In order to further modify data as it passes through the pipeline, shaping operators are additionally utilized. These operators facilitate a variety of data processing tasks, such as filtering, joining, the use of expressions and aggregations, returning distinct values, and sorting. They also enable relational operations such as UNION, MINUS, and INTERSECT, along with additional transformations such as split, pivot, lookup, flatten, and the use of table function operators. The flatten operator is utilized to unnest hierarchical data structures in formats like JSON, multi-line JSON, Avro, and Parquet, with the ability to unnest data in the form of an Array. In order to build expressions, functions can be utilized with operators in these data flows; these functions are aggregate functions (COUNT, MAX, MIN, SUM, AVG, LISTAGG), analytic functions (FIRST_VALUE, LAG, LAST_VALUE, LEAD, RANK() OVER, ROW_NUMBER() OVER), arithmetic functions (ABS, CEIL, FLOOR, MOD, POWER, ROUND, TRUNC, TO_NUMBER), array functions, conditional functions, date and time functions, hash functions, hierarchical functions (SCHEMA_OF_JSON, FROM_JSON, TO_JSON, TO_MAP, TO_STRUCT, TO_ARRAY), higher-order functions, operator (comparison) functions, string functions and unique ID functions. Table function operators also provide many operations employed in data processing such as deduplication, N/A drop, N/A fill, N/A replace, and others. Attributes can also be processed in data flows through transformations, the most relevant ones being changing data types and filling up nulls.

Moreover, OCI supports the building, training, and management of machine learning models via the Data Science service, a fully managed, serverless platform [16]. Equipped with Python-centric tools, libraries, and packages, data scientists can easily acquire, profile, prepare and visualize data, perform feature engineering, train models, and evaluate, explain and interpret their results. Functions, Data Flow, Autonomous Data Warehouse (ADW), and Object Storage are among the other OCI stack components that this service integrates with. Data Science is accessible through the Console, REST API,

Software Development Kits (SDKs), or Command Line Interface (CLI). Model creation and saving can be performed using Accelerated Data Science (ADS) SDK, OCI Python SDK, or the Console, with programmatic creation and saving being the recommended method [17].

Data preparation and machine learning workflows can also be developed using Oracle Data Transforms, a graphical, no-code interface that enables the design of data loads, data flows, and workflows within Oracle Cloud [18]. Mappings created with Data Transforms for loading data into Autonomous Databases can be executed immediately or scheduled for later deployment. Upon execution, Data Transforms automatically generates the necessary underlying code. In the Data Flow editor, the available actions include Data Transform, Data Preparation, Machine Learning, Text, and Oracle Spatial [19]. Oracle Data Transforms supports both ETL and ELT processes [20].

When selecting an appropriate data integration tool, it is worth noting that Data Transforms represents the latest evolution of Oracle Data Integrator (ODI), being built upon its powerful engine [20, 21]. While Oracle Data Transforms shares many foundational concepts with ODI, it utilizes different terminology [22], and is sometimes informally referred to as ODI Web in the Oracle environment. Designed as a cloud-native data integration solution with a modern user interface, it prioritizes ease of use and rapid accessibility.

6 ETL Implementation in Oracle Cloud

For the practical part, I utilized a dataset containing over 50,000 ranked matches from the Europe West (EUW) server of the online multiplayer game League of Legends. The dataset is available on Kaggle [23] and includes three accompanying JSON files used for mapping champion and summoner spell IDs to their corresponding names. The data was collected via the official Riot Games API [24], and it contains historical ranked match information.

My approach involves uploading the files to the Oracle Cloud ADW, executing the necessary data processing, and loading the processed data into a database table located in a different schema within the created warehouse for further analysis.

Using Oracle Data Transforms, I uploaded the required files as objects to Oracle Cloud through the “Load Data” option, which imported them into an ADW instance previously configured for this project. The ADW was organized into two schemas: ETL_SOURCE, designated for the source files, and ETL_OUTPUT, intended for the creation of the target table. The JSON files were loaded from my computer into separate tables named CHAMPION_INFO, CHAMPION_INFO_2, and SUMMONER_SPELL_INFO, with their contents stored as CLOB data types. Because the JSON files contained hierarchical data, I used SQL Developer to flatten them by parsing and keeping the information that I considered relevant for this project. The data from the tables corresponding to the JSON files with champion information was combined into a single table to have the required champion data in one place. After uploading the CSV file, I checked the option to convert invalid numeric values to null and then the file was loaded into a table named GAMES. All of these tables were placed within the ETL_SOURCE schema. A corresponding target table, GAME_INFO, was created in the ETL_OUTPUT schema, aligned with the mapping defined in Figure 1. I decided to keep the match IDs, game duration, the winning team, as well as the champions, their primary roles, and the summoner spells used by each player.

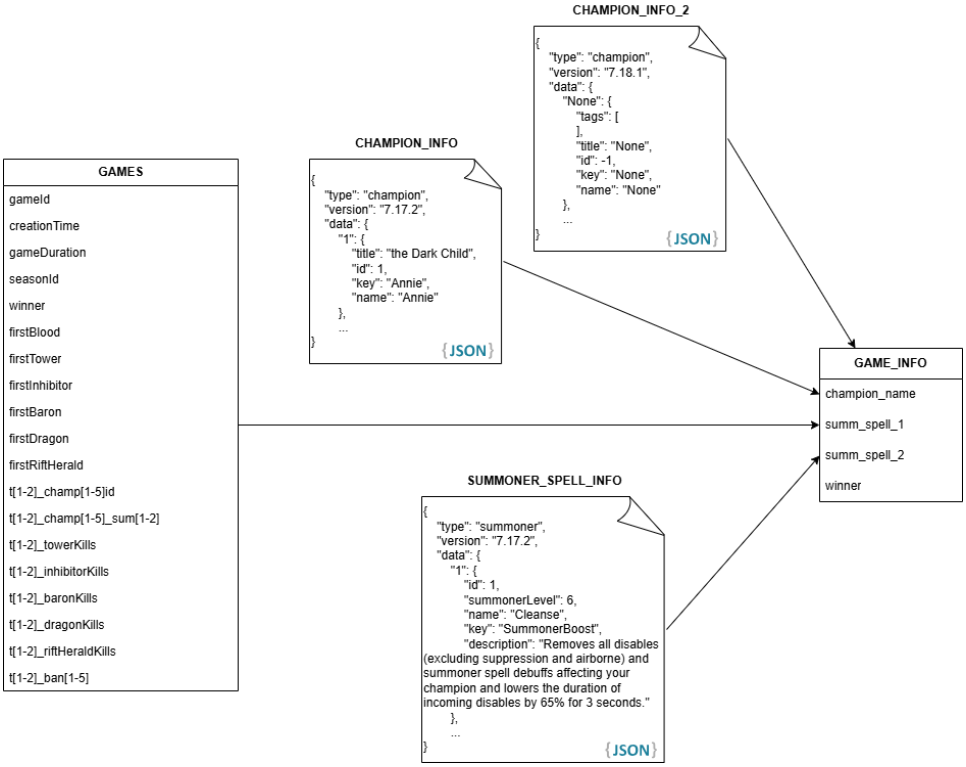


Fig. 1. Data Mapping

Following the data upload, I established two connections via the "Connections" page: ORACLE_SOURCE, linked to the source schema, and ORACLE_OUTPUT, linked to the target schema. I then navigated to the "Data Entities" tab to import the metadata corresponding to the uploaded objects from both schemas. Once the import was done, I created a new project under the "Projects" section and accessed "Data Flows" within it for designing a pipeline to integrate data obtained from the source objects into the target GAME_INFO table, which would then be utilized for analysis.

To eliminate matches where the winner value was equal to 0, which indicates that neither team won and the game ended up as a remake, a "Filter" operation was first used. Next, the "Expression" operator from the "Data Transform" panel was used to perform an unpivot operation, transforming the dataset by expanding values from multiple columns into multiple records, each row ending up with match data for only one player. Specifically, each row in the original dataset represented ten champions per match. Through the use of expressions, I extracted the game ID, winning team, champion name, and the two summoner spells associated with each champion.

Data for Team 1 and Team 2 were combined into a single dataset using the "Set" operator with the UNION operation. Before merging the two obtained sets, I transformed the winner column using the DECODE function to create separate binary indicators for each team: for Team 1, a value of 1 indicated a win (originally 1) and 0 indicated a loss (originally 2); for Team 2, a value of 1 indicated a win (originally 2) and 0 indicated a loss (originally 1). This approach made it possible to show each team's win and loss records separately. This transformation was performed prior to unifying the data to simplify the process and avoid complications related to column prefixes such as "T1_" for the first team and "T2_" for the second one. Following this, left outer joins were executed with the CHAMPION_INFO and SUMMONER_SPELL_INFO tables created from the flattened JSON files to retain all relevant data. Since the SUMMON-

ER_SPELL_INFO table contained two columns representing summoner spells, two join operations were necessary. Finally, the transformed data was loaded into the target table GAME_INFO, with the column mapping illustrated in Figure 2.

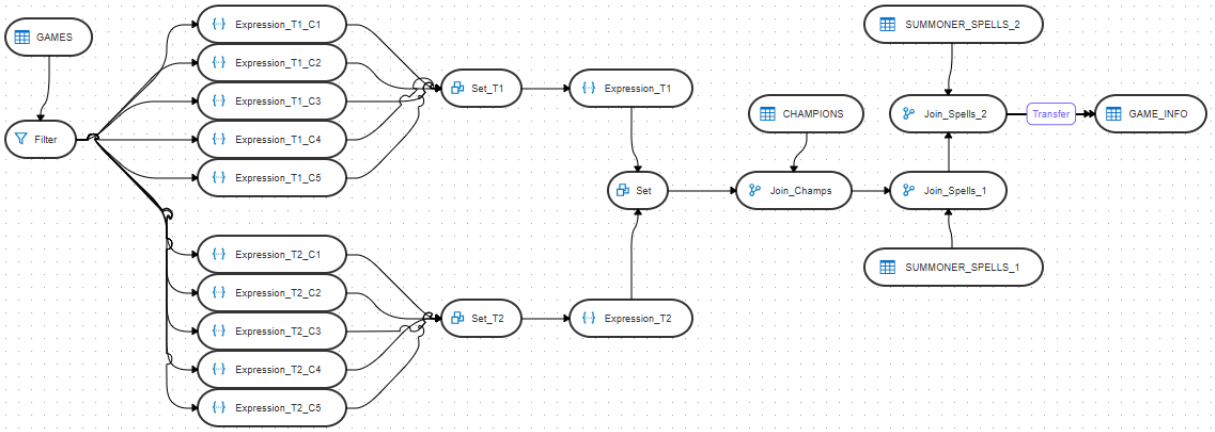


Fig. 2. ETL Workflow

This process resulted in a dataset comprising over 500,000 rows. The final structure of the GAME_INFO table is displayed in Figure 3.

Fig. 3. ETL Output

7 Integrating Data with Machine Learning

To highlight the importance of data preparation for machine learning, I created a simple model utilizing the data from the obtained table in order to estimate the probability of winning in League of Legends. Due to having limited access rights on Oracle Cloud, I couldn't fully utilize its tools designed for machine learning. Therefore, I developed a project on my local machine, in which I connected to the Oracle ADW and performed the necessary operations programmatically in Python.

For example, instead of creating an expression within the ODI Web to insert and utilize a random parameter for splitting the data, I directly performed an 80-20 split of the dataset in Python, dividing it into training and testing sets. Similarly, rather than using the *data_cleanse* feature from the "Data Preparation" tab, I manually handled missing values by replacing empty strings with "" and filling empty numeric fields with mean values, programmatically replicating the cleaning process displayed in Figure 4.

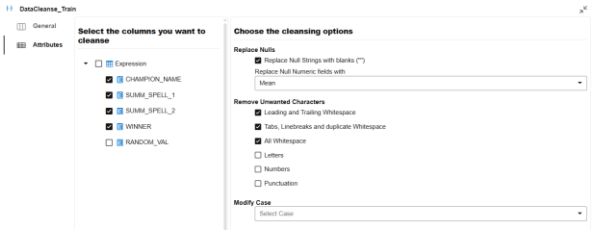


Fig. 4. ODI Web Data Cleanse

Moving on to the modeling stage, I implemented a simple logistic regression to analyze the binary dependent variable WINNER in relation to independent variables representing the champions and summoner spells used in each match, focusing exclusively on pre-game data. As a result, the model produced a low Area Under the Curve (AUC) of 0.52 on the Receiver Operating Characteristic (ROC) curve (Figure 5). According to standard interpretation, an AUC close to 0.5 suggests performance equivalent to random guessing, indicating

that the model had no discriminating ability [25].

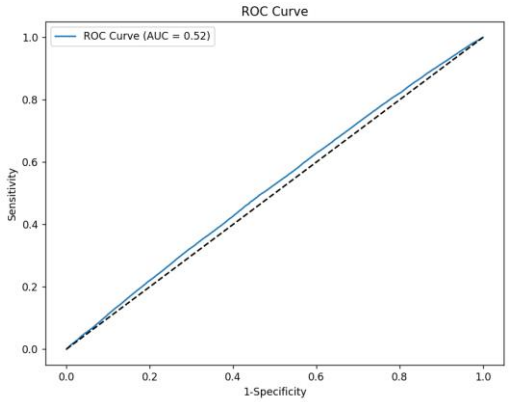


Fig. 5. ROC Curve

The shape of the ROC curve further supports this interpretation, as it leans toward 1-specificity axis, showing a higher rate of false positives [26]. The model incorrectly predicted negative outcomes as positive in several cases. This outcome implies that the selected features are not enough for the model to learn effectively. Consequently, the predictions resembled random classification rather than reflecting significant patterns.

This limitation is not unique; a similar pattern was observed in League of Legends Match Outcome Prediction, where models trained only on pre-game features performed close to random, while models incorporating in-game data achieved significantly better results, as measured by ROC curves [27].

8 Conclusions and Future Work

OCI provides a wide range of options for transforming large datasets from warehouses, and loading them into designated storage systems. For ETL and ELT processes, Oracle provides both ODI and Oracle Data Transforms, also known as ODI Web. The latter is built on the foundation of ODI, using a similar syntax and offering many of the same processing features in a no-code format which can also generate the code required for the defined operations. In Data Transforms, pipelines can be built in the Data Flow interface, making development easier. In addition to the visual workflow, the tool can automatically generate the underlying code once the pipeline is complete. The integration with ADW allows users to manage storage and processing entirely in the cloud.

The ETL process used for this paper demonstrates a modern approach to preparing data from a warehouse and loading it into a target table in a database schema. However, for the analysis part, the results from the machine learning model built from the processed dataset display that the current dataset, based only on pre-game information, lacks the complexity needed to build a strong predictive model.

For future work, the utilized table information could be enhanced by keeping columns which contain events occurring during the matches. Elements such as match duration, control over key objectives like dragons and towers, and other gameplay events can strongly influence the final result of the match. Therefore, adding these variables would likely improve the model's predictive accuracy.

Overall, this paper demonstrates how ODI Web from the OCI suite can be effectively used to develop and maintain ETL operations. It represents a modern solution for data processing and analysis, as it also aids in building machine learning models.

References

[1] W. H. Inmon, *Building the data warehouse*, 4th ed. Wiley Pub., 2005.

[2] Oracle, "Introduction to Data Warehousing Concepts," Data Warehousing Guide, docs.oracle.com. https://docs.oracle.com/en/database/oracle/oracle-database/23/dwhsg/introduction-data-warehouse-concepts.html#GUID-AC9D6CAF-8698-478A-946C-6932F1FCDE99 (accessed June 2025).

[3] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65–74, Mar. 1997.

[4] A. Simitsis and P. Vassiliadis, "Extraction, Transformation, and Loading," in

Encyclopedia of Database Systems, New York, NY: Springer New York, 2018, pp. 1432–1440.

[5] S. K. Bansal and S. Kagemann, "Integrating Big Data: A Semantic Extract-Transform-Load Framework," *Computer (Long Beach Calif)*, vol. 48, no. 3, pp. 42–50, Mar. 2015.

[6] Amazon Web Services, "What's the Difference Between ETL and ELT?," aws.amazon.com. https://aws.amazon.com/compare/the-difference-between-etl-and-elt/ (accessed June 2025).

[7] J. Brownlee, "Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python," Jun. 2020.

[8] Oracle, Oracle Cloud Infrastructure Documentation, "Welcome to Oracle Cloud Infrastructure," docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/GSG/Concepts/baremetalintro.htm (accessed June 2025).

[9] Oracle, "Why Customers are Choosing OCI," oracle.com. https://www.oracle.com/cloud/why-oci/ (accessed June 2025).

[10] Oracle, "Oracle Autonomous AI Lakehouse", oracle.com. https://www.oracle.com/autonomous-database/autonomous-data-warehouse/ (accessed June 2025).

[11] Oracle, "Data Integration," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/data-integration/home.htm (accessed June 2025).

[12] Oracle, "Overview of Data Integration," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/data-integration/using/overview.htm (accessed June 2025).

[13] Oracle, "What are Data Flows," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/data-integration/using/data-flows.htm (accessed June 2025).

[14] Oracle, "Using Data Flow Operators," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/data-integration/using/using-operators.htm#using-operators (accessed June 2025).

[15] Oracle, "Data Science," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/data-science/using/home.htm (accessed June 2025).

[16] Oracle, "Creating and Saving a Model to the Model Catalog," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/Content/data-science/using/models_saving_catalog.htm#saving_models_catalog (accessed June 2025).

[17] Oracle, "The Data Transforms Page," Oracle Cloud Infrastructure Documentation, docs.oracle.com. https://docs.oracle.com/en-us/iaas/autonomous-database-serverless/doc/data-transforms.html#GUID-CD707617-5A77-4399-B4D9-0DC497141CDE (accessed June 2025).

[18] J. Mahto, "Introducing Data Transforms: Built in Data Integration for Autonomous Database," The Autonomous AI Database Insider, blogs.oracle.com, May 2023. https://blogs.oracle.com/datawarehousing/post/introducing-data-transforms-built-in-data-integration-for-autonomous-database (accessed June 2025).

[19] J. Francoisse and E. Lopes, "Comparing Oracle ETL/ELT Tools,"

blogs.oracle.com, May 2024. https://blogs.oracle.com/ateam/post/comparing-oracle-etlelt-tools (accessed June 2025).
[20] J. Mahto, "Selecting the best Data Integration tool – Data Transforms or Oracle Data Integrator," The Autonomous AI Database Insider, blogs.oracle.com, Mar. 2024. https://blogs.oracle.com/datawarehousing/post/selecting-the-best-data-integration-tool-data-transforms-or-oracle-data-integrator (accessed June 2025).
[21] Oracle, "Introduction to Oracle Data Transforms," Using Oracle Data Transforms, docs.oracle.com. https://docs.oracle.com/en/database/data-integration/data-transforms/using/introduction-oracle-data-transforms.html (accessed June 2025).
[22] Kaggle, "(LoL) League of Legends Ranked Games," kaggle.com. https://www.kaggle.com/datasets/datasnaek/league-of-legends (accessed June 2025).
[23] Riot Games, Riot Developer APIs, developer.riotgames.com. https://developer.riotgames.com/apis (accessed June 2025).
[24] Z. H. Hoo, J. Candlish, and D. Teare, "What is an ROC curve?," *Emergency Medicine Journal*, vol. 34, no. 6, pp. 357–359, Jun. 2017.
[25] GeeksforGeeks, "AUC-ROC Curve in Machine Learning," geeksforgeeks.org. https://www.geeksforgeeks.org/auc-roc-curve/ (accessed June 2025).
[26] L. Lin, "League of Legends Match Outcome Prediction," Stanford, CA, USA, 2016.

Cristiana COSTAN graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Academy of Economic Studies in 2024, obtaining a Bachelor's Degree in Economic Informatics. She is currently pursuing a master's degree in Databases - Business Support and will earn her degree in 2026. Her main areas of interest include data analysis, database management, and web development.

Data Visualization in Business Intelligence: A Comparison Between Power BI and Qlik Cloud Analytics

Anda-Elena SPĂTARU, Florin-Răzvan SOARE
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies
Bucharest, ROMANIA
spataruanda21@stud.ase.ro, soareflorin21@stud.ase.ro

The visualization component within Business Intelligence (BI) systems plays an essential role in transforming operational data into actionable insights for the decision-making process. BI tools facilitate data interpretation and the formulation of evidence-based conclusions, supporting organizations in monitoring performance through clear metrics and adapting rapidly to changes in the competitive environment, including in low-latency information contexts. From this perspective, the article presents general BI concepts and proposes a comparative analysis of two representative platforms, Microsoft Power BI and Qlik Cloud Analytics, highlighting their capabilities and limitations based on relevant evaluation criteria.

Keywords: *Business Intelligence, Microsoft Power BI, Qlik Cloud Analytics, Analytics and Reporting Platforms, BI Architecture, Data Warehouse, Data Mart, Comparative Analysis*

1 Introduction

In the current business environment, companies are undergoing an accelerated digitalization process, recognizing that data represents one of the most valuable available resources. This reality is reflected by a significant increase in data volume and a diversification of sources, which amplifies the need for efficient integration and analysis mechanisms. Consequently, the adoption of Business Intelligence systems becomes a necessity and a strategic factor, as it allows for leveraging data and supports real-time decision-making [3].

Data visualization is an essential element in transforming raw data into aggregated and easily interpretable information. Tools oriented towards graphical representations offer a high degree of interactivity and have, over time, integrated user-oriented functionalities, such as multi-criteria filtering and exploring data from different perspectives. By analyzing the trends of key performance indicators (KPIs) and identifying patterns (evolutions, stagnations, or deviations), organizations can formulate conclusions that contribute to understanding market dynamics and customer behavior, as well as to identifying opportunities and risks.

In this sense, BI solution developers do not just build isolated visualizations, but structure the information in a narrative approach, which facilitates the interpretation and communication of results. This approach can contribute to the consolidation of a data-driven organizational culture, in which decisions are based on accurate and consistent information, with the potential to generate competitive advantages [4].

In fields of activity such as energy, finance, and education, the choice and implementation of a visualization platform go beyond the technological dimension, becoming an organizational decision that influences processes and working methods. Since the market offers a wide range of alternative platforms, each with benefits and limitations, there is no universally suitable solution; the selection depends on the specific requirements and the operational context of each organization.

Annually, Gartner, an IT consulting and research company, publishes evaluations regarding Analytics and Business Intelligence platform providers. Through the Gartner Magic Quadrant methodology,

providers are positioned in a synthetic representation that reflects both the ability to execute and the completeness of vision. In this paper, the Magic Quadrant is used for orientation purposes, to contextualize the market, without being treated as a complete validation for all implementation scenarios [10, 11, 12].

Fig. 1 Magic Quadrant for Analytics and Business Intelligence Platforms

In the following chapters, the theoretical foundations associated with the traditional BI architecture are presented, in order to familiarize the reader with the concept and understand how it integrates with the system architecture. Subsequently, the characteristics of two representative technologies, namely Microsoft Power BI and Qlik Cloud Analytics, are detailed. Both the benefits and the constraints of each platform are highlighted, and in the final stage, a comparative analysis is performed based on a set of relevant evaluation criteria.

2 The BI Architectural Model

The concept of Business Intelligence can be structured into three main components: data extraction and integration, data storage, and data access, analysis, and visualization. The BI flow can be described as a sequential process, from information collection to analytical consumption [1].

In the first stage, organizations use internal sources, originating from operational systems (for example, SAP or Salesforce), as well as external sources, such as public data from the Internet or information provided by partners and agencies. These datasets are processed through ETL (Extract, Transform, Load) or ELT (Extract, Load, Transform) processes, within which cleaning and standardization operations are applied, with the aim of ensuring data quality.

The second component, storage, is achieved by loading the data into the Data

Warehouse. At this level, different Data Marts oriented towards specific functional areas can be defined (for example, individual Data Marts can be created for the finance, retail, and logistics departments). Through this logical separation, organizations can adapt the data structure to particular analytical requirements and improve query performance through aggregations and dedicated models [1, 5].

In the final stage, data is leveraged through the development of dashboards and reports, through detailed exploratory analyses, as well as through data mining and machine learning techniques, these representing just a few of the usual methods of analytical consumption. By integrating these capabilities, BI solutions contribute to the transformation of data into analytical conclusions that substantiate organizational decisions [2].

Next, this conceptual structure is illustrated through a traditional BI architecture, which highlights the data flow:

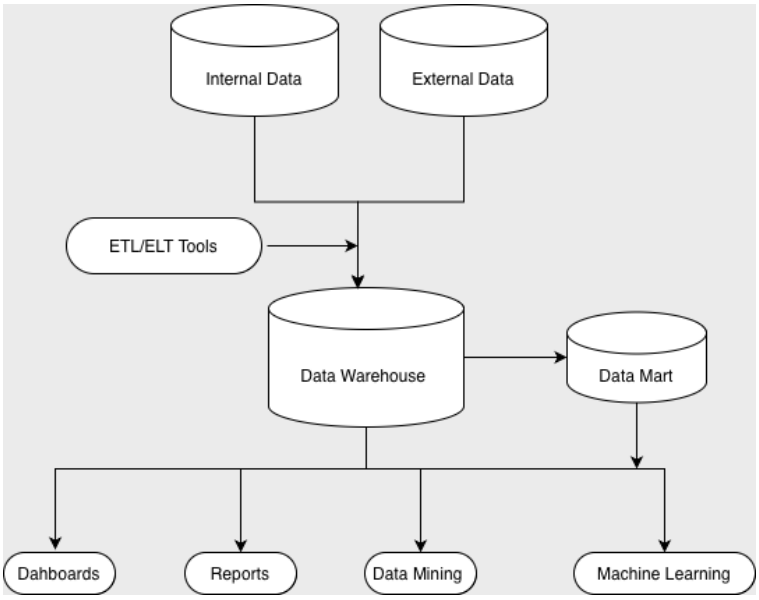


Fig. 2 Architecture of a BI System

3 Overview of the Analyzed BI Platforms: Microsoft Power BI and Qlik Cloud Analytics

Microsoft Power BI, initially launched in 2013, represents a Business Intelligence platform integrated into the Microsoft ecosystem and available within the Microsoft 365 suite. The solution is noted for a high degree of flexibility in use and a relatively low learning curve for users who already have experience with tools such as Microsoft Excel. From the perspective of its components, Power BI includes a cloud-based part, called Power BI Service, used for publishing, sharing, and administering analytical content, as well as a local component, Power BI Desktop, intended for the development of data models and reports [14].

Regarding connectivity, Power BI offers an extensive range of integration options with heterogeneous data sources, including enterprise applications (such as CRM/ERP), relational databases, on-premises sources, files (for example, Excel), and other types of specific connectors. This diversity supports the use of the platform in varied organizational scenarios, where data originates from multiple systems and formats.

For data processing and modeling, Power BI provides a set of established components: Power Query (for extraction and transformation), Power Pivot (for tabular modeling), and the DAX language – Data Analysis Expressions (for defining measures and calculations). Together, these elements allow both the preparation of data and the construction of a semantic model that can support advanced analytical calculations [2, 6].

From the perspective of user perception, the specialized literature highlights favorable evaluations for Power BI in organizational contexts. The study conducted by Kumaresan and Shalu, oriented towards understanding the perspective of organizations that have adopted Power BI across different industries, reports a high level of satisfaction from the respondents. According to the authors, over 75% of the evaluations were positive in each of the three analyzed areas: efficiency in reporting and visualization, impact on business decisions, and real-time analysis capabilities [6].

Ease of development and accessibility for non-technical users represent a primary advantage of the platform. Power BI offers a relatively easy-to-use drag-and-drop interface that allows both BI developers and business users to explore data and build visualizations. Furthermore, a significant portion of the configurations, such as models, visuals, and interactions, can be achieved through interface options, reducing the need for complex code-level interventions [5]. Another strength consists of its high capabilities for collaboration and distribution of analytical content. The platform provides flexible mechanisms for publishing and sharing reports, with the possibility of controlling data access based on the user's role. In this way, teams can work on the same reports and use a common set of indicators. The integration with the Microsoft 365 ecosystem, including Microsoft Teams, facilitates access to reports in collaborative contexts like meetings, conversations, and workspaces [6].

Regarding the challenges, scalability emerges as a primary concern. For large-sized organizations characterized by a high number of sources and significant data volumes, scalability becomes a relevant issue. The increase in data complexity and volume can lead to performance degradation, particularly concerning the update times of dashboards. This aspect represents an important limitation in contexts where the real-time availability of information is required [5, 6].

Additionally, governance and the consistency of indicators in a self-service environment pose significant challenges. The flexibility of Power BI can lead to duplicated reports and non-uniform definitions of KPIs if a common semantic model and clear rules for development and publishing are not established.

Qlik Cloud Analytics is a Business Intelligence and analytics platform offered by Qlik in a SaaS (Software as a Service) model, being hosted and managed by the provider. The solution is positioned in the "modern data stack" area, as it covers an end-to-end flow, from connecting and integrating data sets to analysis and visualization, with an emphasis on interactive and exploratory consumption [13].

A distinctive element of the platform is the use of the Qlik Associative Engine, an associativity-oriented engine that allows data exploration from multiple perspectives by highlighting the relationships between entities and contextual filtering. In this model, data is loaded and processed in memory, which facilitates fast queries and dynamic recalculations during user interaction. As a result, users can perform exploratory analyses with high granularity, quickly identifying relevant correlations, exceptions, or patterns as they modify selections and filters [7].

Unlike strictly SQL-based query approaches (joins) where the analytical context is often limited to explicitly defined relationships, the Qlik associative model allows navigating through data in a more flexible way, maintaining visibility over elements correlated and uncorrelated with the current selection.

At the interface level, after each selection, the platform instantly updates the context and highlights the possible states of the data (for example: selected values, values compatible with the selection, and incompatible values). This mechanism facilitates what-if analysis and supports the rapid formulation of additional analytical questions, as well as the identification of opportunities or risks in the data [7].

Regarding connectivity, Qlik Cloud Analytics provides support for a varied range of sources, including SaaS applications, databases, web APIs (REST), and cloud storage systems (for example, AWS S3 or Azure).

From the perspective of data preparation, the platform includes integration and transformation functionalities that can be used directly within the development environment, including through the Load Editor area, where loading and transformation flows can be defined, thus integrating ETL/ELT type components in the process of building the analytical application.

Low-latency interactive analysis represents a major strength of the platform. The associative engine allows for the dynamic recalculation of visualizations after each selection, immediately highlighting data relationships through association states (selected values, compatible values, and incompatible values).

This mechanism supports the rapid exploration of data and the identification of correlations or exceptions during analysis [7].

Furthermore, scalability and suitability for complex data models constitute another significant advantage. Qlik Cloud Analytics is well-suited for scenarios involving large data volumes and complex relationships between entities, enabling multidimensional and exploratory analysis without imposing a rigid data navigation flow.

Regarding the challenges, the complexity of development and the learning curve are notable. Development in Qlik requires familiarization with platform-specific concepts and mechanisms, including the load script and the expressions used in the interface. Consequently, the learning curve can be steeper for users without prior experience in Qlik. Additionally, constraints regarding data modeling and association control can arise. The associative model inherently relies on associating fields with the same name across tables.

This characteristic can become a limitation in situations where names are not harmonized or when there are multiple relationships between the same entities, in which case script-level interventions are necessary to avoid ambiguities and unexpected results in the analysis.

4 Comparative Analysis: Power BI vs. Qlik Cloud Analytics

To obtain a more comprehensive perspective on the two solutions presented in the previous chapter, Microsoft Power BI and Qlik Cloud Analytics, this section proposes a multidimensional comparative analysis, tracking relevant capabilities across the entire BI flow: data integration and preparation, modeling, data exploration, and creating visualizations.

From the perspective of data integration and transformation processes, Qlik Cloud Analytics offers an efficient in-app processing mechanism, particularly through the use of QVD files (Qlik native format) and the possibility of organizing the load script modularly for reuse. In Qlik's technical literature, it is mentioned that reading from QVD files is typically 10 to 100 times faster than reading from other sources, which can significantly reduce load times in certain scenarios [9]. In Power BI, the main component for data integration and preparation is Power Query, which allows applying transformations through sequential steps. This approach is advantageous due to the transparency of the transformations (step history) and accessibility for non-technical users, who can perform many operations directly from the interface.

An important difference between the two technologies is the way the data model is built. In Power BI, relationships between tables can be automatically detected and subsequently reviewed and configured manually, including choosing the filtering direction. This platform has a predominantly visual approach, and relationships are defined interactively in the Model View area. In Qlik, the associative model is based on the implicit association of fields with the same name across tables; the control of these associations is mainly achieved through scripts (for example, renaming, loading rules), not by manually adjusting relationships in the model, as seen in Power BI. In this context, the notion of the Synthetic Key also appears, which is formed when there are multiple common fields with the same names between tables in Qlik Analytics, generating ambiguities that must be resolved through remodeling, such as clarifying keys or restructuring links [15].

In the stage of validating loaded data, the tools offer different approaches. In Power BI, the user has dedicated areas for examining the model and the data, such as viewing the model in the Model View area and tabular viewing in the Table View area, which facilitates quick checks (sorting, filtering, inspecting values) and adjustments through transformations or calculated columns. In Qlik, verification is usually done by viewing the model, previews available in the load area, and often by building a control sheet (for example, simple tables) to validate content and associations. Additionally, both platforms offer data profiling functionalities at the column level, providing descriptive statistics, which are useful for quickly identifying anomalies and evaluating data quality before the visualization stage.

From the perspective of languages, Power BI uses M in the Power Query area for transformations and DAX for analytical measures and calculations. In Qlik, data preparation is performed through Qlik Script, and the analytical logic in visualizations is expressed through specific expressions in the Set Analysis section. Typically, for users without technical experience, Power BI may seem more accessible at first due to the high number of operations that can be performed through the interface; in Qlik, for advanced scenarios, the need to use scripts arises more quickly, which can steepen the learning curve [4].

At the visualization level, both platforms offer an extensive portfolio of charts and configuration options. Regarding page arrangement, Qlik uses a grid layout, which facilitates the alignment of visualizations, while Power BI offers

greater freedom of positioning on the canvas. From the perspective of style customization, Power BI generally has greater control over formatting elements, such as fonts and design details. In Qlik, colors can be selected from palettes or set through expressions and themes; there is control, but the level of customization depends on the type of visualization and the chosen approach.

From the perspective of end-users' interaction with reports, the number of actions required to complete a task directly influences the efficiency of exploratory analysis. In this regard, the BARC Benchmark comparatively evaluates Power BI and Qlik in usual scenarios, reporting differences in productivity and stability. The communications associated with the benchmark mention that Qlik requires fewer interactions (clicks) per task, contributing to higher productivity during tests: on average, a Power BI user would need approximately 1.3 times (≈30%) more interactions to accomplish the same task compared to Qlik [8]. A practical explanation relates to how each platform exposes the filtering context. In Qlik, the associative mechanism immediately highlights the state of the values (selected, possible, impossible), and the user sees the active selections at the top and can quickly remove a selection or reset the context through a "clear all selections" action, which shortens the exploration cycle. In Power BI, filters are mainly managed through the Filters pane, with filters existing at the visual, page, and report levels; the user can inspect the filters, but their visibility and management depend on how the report is configured and the filtering interface (pane or slicers), which can lead to a higher number of steps in exploration, especially in complex scenarios.

To summarize the information described in this chapter, the most important evaluation criteria have been grouped below, offering a conclusion for each of the two analyzed technologies.

Table 1. Summary of the comparison between Power BI and Qlik Analytics

Evaluation Criterion	Microsoft Power BI	Qlik Cloud Analytics
Data Integration and Preparation	Power Query (M) + step-by-step transformations; accessible approach, many operations available via the interface.	Load script + QVD; efficient for in-app processing and reuse.
Data Modeling	Visual modeling; manually configurable relationships (cardinality, cross-filter direction).	Associative model; implicit associations based on common field names, primarily controlled via script.
Exploration and Interactivity	Filter/slicer-based exploration; highly dependent on report design.	Associative exploration (selected/possible/excluded); highly suitable for exploratory analysis.
Exploration and Interactivity	High degree of freedom in arrangement and formatting; detailed customization.	Grid layout for alignment; good customization, but more dependent on palettes, themes, and expressions.
Languages and Learning Curve	M + DAX; easier initially for users familiar with Excel.	Qlik Script; steeper learning curve for advanced scenarios.

Given the points presented above, a comprehensive overview of the similarities and differences between the two platforms can be outlined. However, to ensure their efficient use and achieve adequate performance in an organizational context,

it is necessary to consider complementary factors, such as the infrastructure and the way the platform is integrated into the organization's existing processes and systems, the design of robust and secure data flows capable of providing real-time data according to operational requirements, the definition and application of clear data governance policies, increasing the level of data literacy for users who utilize reports and dashboards, and consolidating the skills of development teams (BI developers and analysts) to leverage existing capabilities and adopt new platform functionalities [3].

5 Conclusion

In conclusion, Business Intelligence tools play an essential role in supporting organizational decisions by transforming data from heterogeneous sources into relevant, synthetic, and easily interpretable information. In the current context, characterized by accelerated digitalization, increasing data volumes, and the diversification of sources, BI becomes a strategic factor that supports both performance monitoring through key performance indicators (KPIs) and the identification of trends, patterns, and deviations impacting the competitive environment.

The paper framed this issue by presenting the traditional BI architecture, structured into three main components: data integration and preparation (extraction, transformation), storage (Data Warehouse and, where applicable, Data Marts), and analytical consumption (reporting, exploratory analysis, and advanced methods). This framework highlights the fact that the value of BI is not derived exclusively from visualizations, but from the entire chain that ensures data quality, consistency, and availability for analysis, including in scenarios with low-latency requirements.

Based on this foundation, the comparison between Power BI and Qlik Cloud Analytics indicates that both platforms are well-established and widely used, yet they differ in their approach and in the way they support data development and exploration. Power BI stands out through its integration into the Microsoft ecosystem, its initial accessibility, and its visual modeling of relationships, making it suitable for scenarios where rapid development, standardization, and easy distribution of reports are prioritized. In contrast, Qlik Cloud differentiates itself through its associative model and interactive exploration mechanisms, which facilitate exploratory analysis and the highlighting of data relationships, particularly in contexts with complex models and iterative analytical questions.

References

[1] R. A. Khan and S. M. K. Quadri, “Business Intelligence: An Integrated Approach,” Business Intelligence Journal, vol. 5, no. 1, pp. 64–70, Jan. 2012.

[2] A. Bocevska, S. Savoska, and I. Milevski, “BI Tools Analysis According to Business Criteria as Data Integration Possibilities, Hardware Specification, Tools for Data Visualization and Comparison of Used Technologies,” in *Proc. Information Systems & Grid Technologies (ISGT’2017)*, Sofia, Bulgaria, Sep. 29–30, 2017, pp. 80–90.

[3] A. Mishra, “The Role of Data Visualization Tools in Real-Time Reporting: Comparing Tableau, Power BI, and Qlik Sense,” International Journal on Science and Technology (IJSAT), vol. 11, no. 3, pp. 1–8, Jul.–Sep. 2020.

[4] M. M. Al-Momani, “The Role of Data Visualization in Business Intelligence: A Comparative Analysis of Tools and Techniques,” OEIL Research Journal, vol. 23, no. 9, pp. 264–276, 2025.

[5] Sarode, “Comparative Study and Analysis of BI Tools,” *International Journal of Advance and Innovative Research*, vol. 6, issue 1 (XXVIII), pp. 154–159, Jan.–Mar. 2019.

[6] K. D. Jayaraman and S. Jain, “Leveraging Power BI for Advanced Business Intelligence and Reporting,” International Jour-

nal for Research in Management & Pharmacy, vol. 13, no. 11, p. 16, Nov. 2024.
[7] QlikTech International AB, "The Associative Difference: Analytical power driven by unique engine technology," Data Sheet, Qlik, 2024. [Online]. Available: https://assets.qlik.com/image/upload/v1711580182/qlik/docs/resource-library/datasheets/resource-ds-the-associative-difference-freedom-from-the-limitations-of-query-based-tools-en_ixdlwu.pdf
[8] T. Zeutschler and S. Sexl, "BARC Benchmark: Evaluating productivity and scalability of BI, Analytics & CPM software under real-world conditions. Starting with Power BI and Qlik," BARC Research, Inc., Sep. 2025. [Online]. Available: https://pages.barc.de/hubfs/Marketing/Reports/BARC-Benchmark-PowerBI-Qlik.pdf
[9] Qlik, "QVD files," Qlik Cloud Help. [Online]. Available: https://help.qlik.com/en-US/cloud-services/Subsystems/Hub/Content/Sense_Hub/Scripting/QVD-files-scripting.htm
[10] Gartner, "Magic Quadrant FAQ," Gartner. [Online]. Available: https://www.gartner.com/en/about/magic-quadrant-faq
[11] Gartner, "Magic Quadrant Research Methodology," Gartner. [Online]. Available: https://www.gartner.com/en/research/methodologies/magic-quadrants-research
[12] Gartner, "Content Compliance Policy," Gartner. [Online]. Available: https://www.gartner.com/en/about/policies/content-compliance
[13] Introducing Qlik Cloud. Qlik Help. [Online]. Available: https://help.qlik.com/en-US/cloud-services/Subsystems/Hub/Content/Global_Common/HelpSites/introducing-qlik-cloud.htm
[14] Microsoft, "What is Power BI?," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/power-bi/fundamentals/power-bi-overview
[15] Qlik, "Synthetic keys," Qlik Cloud Help. [Online]. Available: https://help.qlik.com/en-US/cloud-services/Subsystems/Hub/Content/Sense_Hub/Scripting/synthetic-keys.htm. Accessed: 07-Jan-2026

Anda Elena Spătaru graduated at the Bucharest University of Economic Studies in 2024, earning a Bachelor's degree in Economic Informatics. She is currently pursuing a master's degree in Databases for Business Support, expected to be completed in 2026. Professionally, Anda has contributed to the development of digital solutions that enhance business efficiency and support data-driven decision-making. She began her career as a Data Engineer at PPC, later transitioning into data analysis and business intelligence development.

Răzvan-Florin SOARE earned his bachelor's degree in Economic Informatics in 2024 and now he is a master's student at the Academy of Economic Studies from Bucharest, Databases – Support for Business. He will graduate from this program in 2026. Professionally, he works as a Dabase Developer in the banking sector, where he designs, optimizes, and manages complex data systems. Beyond his core expertise, he is deeply passionate about artificial intelligence and process automation.

An Intelligent Recommendation System Built on Emotional Analysis in a Kappa Architecture

Diana – Andreea CĂUNIAC, Adela BÂRA
Bucharest University of Economic Studies, 010374 Bucharest, Romania
diana.cauniac@csie.ase.ro, bara.adela@ie.ase.ro

Recommendation systems have become a cornerstone of modern e-commerce, directly shaping user experience and conversion rates. Yet most conventional approaches rely solely on behavioral history, clicks, views, purchases, without any awareness of how a user is feeling in the moment of decision. This paper presents an intelligent recommendation system that fills that gap by weaving emotional analysis of text reviews into a real-time data-processing pipeline. The solution is built on a Kappa architecture and leverages Apache Kafka, Google Cloud BigQuery, Bigtable, BigQuery ML, and a RoBERTa-based NLP model for emotion detection. Users are grouped into clusters through K-Means segmentation according to their emotional profiles and recommendations are then derived from well-established correlations between emotional states and product categories. In controlled evaluation, the system achieved a Precision@5 of 0.98, a Recall@5 of 0.94, and an F1-score of 0.96, confirming the strength of the proposed approach.
***Keywords**: Recommendation systems, sentiment analysis, BigQuery ML, Apache Kafka, Kappa architecture, RoBERTa, natural language processing, personalized recommendations.*

1 Introduction

The rapid digital transformation of e-commerce has produced an enormous and ever-growing volume of user-generated data. Every click, page view, purchase, and review contributes to a rich behavioral profile that platforms use to personalize the consumer experience. At the center of this effort sit recommender systems, tools that have proven indispensable for improving user satisfaction and driving commercial performance [1], [2], [3].

Most widely deployed systems, however, share a fundamental blind spot. Whether they rely on collaborative filtering, popularity rankings, or content-based analysis, they treat user intent as a purely behavioral signal. They can tell you what a user did, but not why [4]. Consider two people who both purchase the same book: one is looking for a relaxing weekend read, the other for professional development. Their behavior is identical; their needs are not.

This is where affective analysis comes in. Emotions are a powerful driver of purchasing behavior, and accounting for a user's current emotional state opens the door to recommendations that feel genuinely relevant rather than merely statistically probable. The potential here is significant, not just for accuracy, but for the quality of the overall user experience [5], [6].

This paper introduces a recommender system that integrates emotional analysis of textual reviews with real-time stream processing, built on a Kappa architecture and deployed within the Google Cloud Platform ecosystem [7]. The goal is a scalable, low-latency framework that delivers recommendations sensitive to both a user's transactional history and their dynamic emotional state [8].

2 System Architecture

The system is built around two guiding principles: modularity and a microservices-based design. Each component operates independently and communicates through well-defined interfaces, which makes the architecture easy to extend without disrupting what is already running.

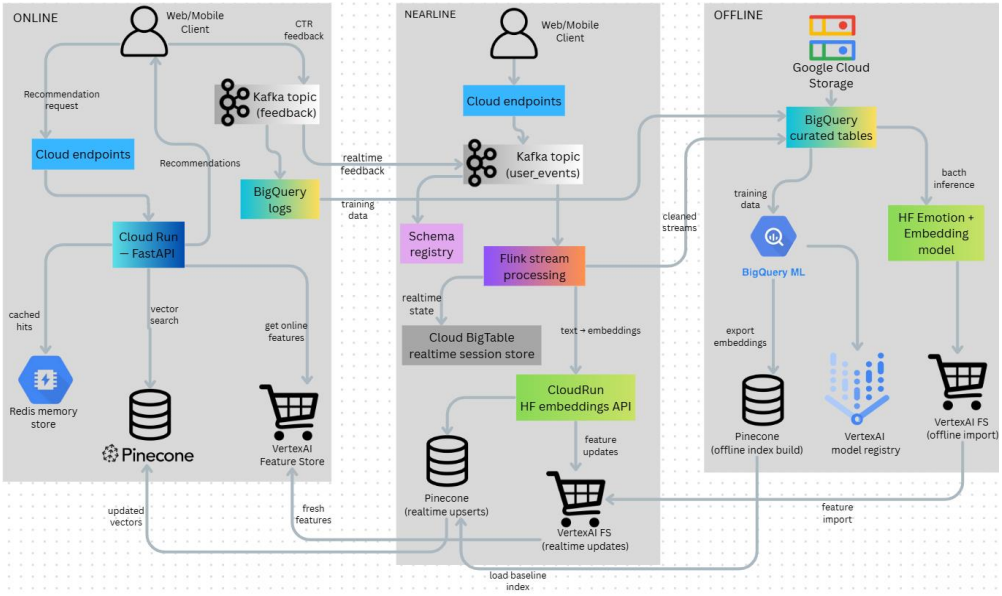


Fig. 1. Example General Architecture of the Emotion-Based Recommendation System

As illustrated in Fig. 1, the system is divided into three processing layers: online, nearline, and offline. This separation allows real-time recommendations to be served while user profiles are being continuously updated in the background, and machine learning models are retrained without any impact on the live environment.

The system is built on a Kappa architecture, which processes both historical data and real-time streams through a single unified pipeline [9], [10]. This sets it apart from the traditional Lambda architecture, where batch and streaming layers run in parallel and must be maintained independently, a separation that adds significant complexity and operational overhead. By consolidating everything into one pipeline, the Kappa approach keeps the architecture simpler and ensures the low latency that real-time recommendations require [11], [12].

All data enters the system from user interactions on web and mobile applications. These events flow continuously into Apache Kafka, which distributes them to downstream components via distinct topics, a design that decouples processes cleanly and limits inter-module dependencies [13]. Before any message is processed, a Schema Registry validates its structure against JSON or AVRO schemas. Any payload that does not conform is discarded, preventing formatting errors or malformed data from propagating through the system [14].

3 Data Ingestion and Emotional Analysis

Data ingestion is implemented through three parallel streas inventory updates, user interactions, and textual reviews, published continuously to Apache Kafka. Listing 1 presents the core ingestion logic responsible for generating and publishing these event streams concurrently.

```
  FUNCTION run_ingestion(limit):
      producer                              =
initialize_kafka_producer(broker="localh
ost:9092")

      for i in range(0, limit):
          client_id = random_integer(1,
250000)
          product_id = random_integer(1,
3700000)
          session_id = random_integer(1,
500000)

          # Stream 1: Inventory
          product                           =
generate_product(product_id,   category,
brand, price)

producer.send("inventory_updates",
key=product_id, value=product)

          # Stream 2: User Interaction
```

```
            interaction                 =
generate_interaction(client_id,
product_id, session_id,

event_type=random_choice(["view",
"click", "cart", "purchase"]),

device=random_choice(["Mobile",
"Desktop", "Tablet"])
            )

producer.send("user_events",
key=client_id, value=interaction)

            # Stream 3: Feedback (20%
probability)
            if random() > 0.8:
                review                  =
generate_review(client_id,
product_id,

text=faker.paragraph(),

emotion=random_choice(EMOTIONS)
                )

producer.send("feedback",
key=client_id, value=review)

            if i % 1000 == 0:
                producer.flush()
                log(f"Ingested       {i}
batches...")

        producer.flush()
        log("Ingestion complete.")
```

Listing 1. High-Throughput Ingestion Engine

As shown in Listing 1, the three streams operate concurrently and converge within a unified pipeline, closely mirroring a real production environment and allowing the system to be tested under genuinely demanding conditions.

Each stream serves a distinct role. The first keeps the product catalog current, continuously refreshing attributes such as category, brand, price, and dimensions. The second captures user behavior at a granular level: page views, clicks, cart additions, and completed purchases. The third processes written reviews and assigns emotional labels to each one, progressively building out the affective dimension of the user profile.

Throughput was assessed against a dataset of one million records, which produced roughly one million interaction events, one million inventory updates, and 200,000 textual reviews. The full workload completed in approximately twelve minutes and forty-three seconds, a result that speaks to the pipeline's capacity for sustained, high-volume data ingestion.

Emotion detection is powered by the SamLowe/roberta-base-go_emotions model from Hugging Face, a RoBERTa-based classifier fine-tuned on the GoEmotions dataset, a corpus of around 58,000 Reddit comments labeled across 28 emotional categories [14], [15]. For each review, the model identifies the single most prominent emotion, whether joy, sadness, anger, surprise, disgust or fear, translating raw user text into structured affective signals that the rest of the system can act on directly [6].

4. Storage and User Segmentation

The system relies on two complementary storage technologies, each chosen to meet specific performance requirements while respecting the constraints of the CAP theorem.

Google Cloud Bigtable handles active session data, providing sub-millisecond read latency. Each record uses a composite row key structured as user_id#session_id#timestamp, a schema chosen to enable fast querying and support the real-time updating of recommendations. Google Cloud BigQuery serves as the analytical data warehouse, storing the full history of user interactions along with emotional scores, geographic data, event types and training datasets for machine learning models.

User segmentation is performed directly within BigQuery using the K-Means clustering algorithm, available natively through BigQuery ML. Users are grouped into six clusters corresponding to the six dominant emotional states: joy, sadness, anger, surprise, disgust, and fear. Running the model where the data already lives eliminates the need to export datasets to external systems, a choice that reduces both operational cost and processing time substantially.

5. Recommendation Generation

Once a user has been assigned to an emotional cluster, the system maps that cluster to a curated set of product categories. These mappings are grounded in well-established findings from behavioral economics and consumer psychology, which document how emotional states systematically influence purchasing decisions.

Table 1 presents the full mapping between detected emotions and recommended categories, along with the psychological reasoning behind each association.

Table 1. Emotion-to-Category Mapping with Psychological Rationale

Emotion	Matched Categories	Psychological Rationale
Joy	Electronics, Toys, Clothing	A positive mood encourages aspirational and reward-driven purchases.
Anger	Books, Home & Kitchen, Movies	Comfort-seeking behavior favors familiar, nurturing product categories.
Surprise	Sports, Automotive, Tools	Frustration often redirects energy toward physical activity or productivity.
Fear	Gifts, Specialty Products, Electronics	Novelty-seeking behavior aligns naturally with discovery-oriented categories.
Disgust	Safety, Books, Home & Kitchen	A desire for security motivates purchases of protective and educational products.
References	Health, Tools, Garden	Drives toward cleanliness and self-improvement map onto health and environmental products.

Every recommendation generated by the system is logged in the recommendations_log table along with the rationale behind its selection. This transparency ensures full traceability of decisions and makes rigorous performance evaluation straightforward.

The system also actively mitigates the risk of creating a filter bubble. By distributing recommendations across a minimum of fourteen distinct product categories, it prevents the over-repetition of similar suggestions, promoting diversity in what users see and by extension, a richer browsing experience.

6 Performance Evaluation

Evaluation begins with a look at data distribution, specifically to confirm that the datasets used are sufficiently diverse and representative. A consistent spread across interaction types, device categories and emotional states indicates that the data is well-balanced and appropriate for training. The product catalog contains over 3.7 million records across six primary categories, providing a solid base for generating relevant recommendations.

Fig. 2 maps the six dominant emotional states against major commercial regions in Romania, revealing meaningful geographic variation in user sentiment, a reminder that emotional behavior is not uniform across populations.

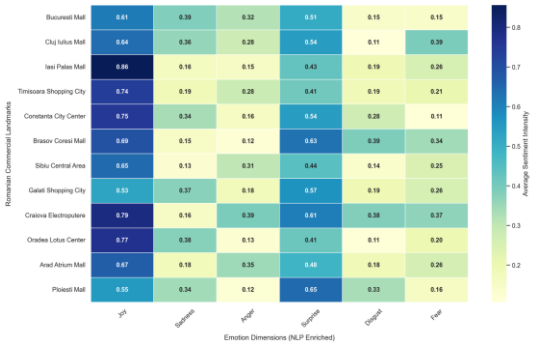


Fig. 2. High-Density Regional Sentiment Correlation

After applying the RoBERTa model, joy emerged as the emotion with the highest average score across all processed reviews, which aligns with the general tendency of online reviews to skew positive or neutral.

System performance was evaluated using the standard top-N recommendation metrics Precision@5, Recall@5, and F1-score. While Precision@5 measures the proportion of relevant recommended items among the top five suggestions, Recall@5 reflects the proportion of all relevant items successfully retrieved by the system. To provide a balanced evaluation between precision and recall, the F1-score was calculated using the harmonic mean of the two metrics as follows [16]:

$$F1 = 2 \times (Precision \times Recall) / (Precision + Recall)$$

System performance was measured using three standard metrics for top-N recommendation tasks:

- Precision@5: 0.98
- Recall@5: 0.94
- F1-score: 0.96

To put these numbers in context, the system was compared against three conventional baselines: random recommendation, popularity-based ranking and content-based filtering. Table 2 presents this comparative analysis, clearly demonstrating the superiority and enhanced accuracy of the emotion-aware approach [16].

Table 2. Comparative Performance: Superiority of BQML-Driven Logic

Strategy	Precision@5	Recall@5	F1-Score
Random Recommendation	~0.17	~0.14	~0.15
Popularity-Only Baseline	~0.35	~0.30	~0.32
Content-Only Filtering	~0.48	~0.40	~0.42
BQML Emotion-Aware	0.98	0.94	0.96

The gap is substantial. The emotion-aware system achieves roughly 3.1 times the precision of a random baseline and around 1.5 times that of a popularity-only approach. These are not marginal gains as they confirm that embedding affective signals into the recommendation pipeline produces a meaningful and measurable improvement.

Figure 3 shows the final evaluation log, tracing the pipeline's execution from initial ingestion through to performance validation. The output confirms that every stage ran cleanly and that the reported metrics reflect a complete, end-to-end process.

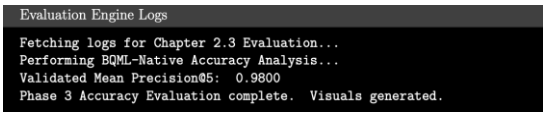


Fig. 3. Evaluation Engine Log

7 Limitations and Future Directions

Despite the promising results, the system has several limitations that deserve honest acknowledgment.

The most persistent challenge is the cold-start problem. When a new user has no interaction history, there is no basis for building an affective profile and the system must fall back to generic, non-personalized recommendations until enough behavioral data accumulates.

A second limitation concerns emotional

ambiguity. When multiple emotional categories receive similar probability scores, reducing a user's complex internal state to a single dominant label may oversimplify what they are actually feeling and potentially reduce the relevance of the resulting recommendations.

The use of synthetic data during evaluation also introduces a potential bias. Artificially generated reviews tend to lean positive or neutral, which inflates the proportion of users assigned to the joy cluster and may limit the ecological validity of the results.

Addressing these limitations paves the way for several future research directions:

• Weighted Emotion Aggregation: Rather than isolating a single emotion, the system could process the full distribution of detected emotions simultaneously, weighting each proportionally to capture a more nuanced psychological profile.

•Hybrid Recommendation Engine: Combining collaborative filtering with the emotion-aware logic could create a better balance between personalization depth and content diversity, reducing the risk of over-specialization.

• Production Deployment and A/B Testing: The ultimate test of the system's value will come from live deployment at scale. Rigorous A/B testing in production environments will be essential to validate real-world performance, scalability and measurable impact on user engagement.

8 Conclusions

This research makes a clear case for the role of affective analysis in modern recommender systems. By integrating NLP-derived emotional signals with a Kappa architecture and BigQuery ML, the proposed framework delivers a scalable, low-latency solution that responds to how users actually feel, not just what they have done in the past.

The empirical results speak for themselves: a Precision@5 of 0.98 and an F1-score of 0.96 represent a substantial improvement over every conventional baseline tested. The system is not simply more accurate, it operates from a fundamentally different understanding of what drives user behavior.

Ultimately, this work moves the needle on personalization by adding a cognitive dimension that most recommendation systems currently lack: genuine awareness of the user's emotional state. That shift, from tracking behavior to understanding motivation, opens rich possibilities for the future of intelligent personalization, across e-commerce and well beyond it.

Acknowledgment

This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI - UEFISCDI, project number COFUND-CETP-SMART-LEM-1, within PNCDI IV.

References

[1] F. Ricci, L. Rokach, and B. Shapira, Recommender Systems Handbook, 3rd ed. Springer, 2022.

[2] M. Alam, “Hybrid Recommender Systems: A Systematic Review,” Information Processing & Management, 2024, pp. 10–14.

[3] L. Chen et al., “Explainable Recommender Systems: A Survey and New Perspectives,” Information Processing & Management, 2024, pp. 1–20.

[4] Y. Deldjoo et al., “Recommender Systems Leveraging Contextual Information: A Survey,” Information Fusion, 2024, pp. 101–120.

[5] J. Li et al., “Real-time Recommendation Systems: Architectures and Challenges,” IEEE Access, 2024, pp. 678–692.

[6] B. Pang and L. Lee, “Opinion Mining and Sentiment Analysis,” Foundations and Trends in Information Retrieval, vol. 2, no. 1–2, pp. 1–36, 2008.

[7] S. Zhang, “Deep Learning Based Recommender Systems: A Survey and New Perspectives,” ACM Computing Surveys, 2022, pp. 5–7.

[8] R. Sharma, P. Agarwal, and A. Arya, "Natural Language Processing and Big Data: A Strapping Combination," in New Trends and Applications in Internet of Things (IoT) and Big Data Analytics, 2022, pp. 255–271.
[9] N. Marz and J. Warren, "Lambda and Kappa Architectures Revisited," IEEE Software, 2023, pp. 30–40.
[10] J. Kreps, "Apache Kafka and Stream Processing Architectures," LinkedIn Engineering, 2022, pp. 10–12.
[11] M. Kleppmann, Designing Data-Intensive Applications. O'Reilly Media, 2023.
[12] I. Hashem, "Big Data Processing in Cloud Environments," Journal of Cloud Computing, 2023, pp. 44–50.
[13] D. Fawzy, S. Moussa, and N. L. Badr, "The Internet of Things and Architectures of Big Data Analytics: Challenges of Intersection at Different Domains," IEEE Access, 2022, pp. 4969–4992.
[14] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2018.
[15] Y. Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach," 2019.
[16] G. Shani and A. Gunawardana, "Evaluating Recommendation Systems," in Recommender Systems Handbook, Springer, 2011, pp. 257–297.
Issue 5, May 2008, ISSN 1109-2750, pp. 473-482.

Diana-Andreea CĂUNIAC earned her bachelor's degree in Economic Informatics in 2020 and her master's degree in Databases – Support for Business in 2022. She is currently pursuing a Ph.D., focusing on big data in real time. Professionally, she worked in the field of databases and currently works as a Fullstack Developer. Her research interests include big data, database systems, software development, artificial intelligence, and modern web technologies.

Adela BÂRA graduated the Faculty of Economic Cybernetics in 2002, holds a PhD diploma in Economics from 2007. She is professor at the Economic Informatics Department at the Faculty of Cybernetics, Statistics and Economic Informatics from The Bucharest University of Economic Studies and coordinated three R&D projects. Her research interests are focused on data science, analytics, databases, IoT, big data, data mining, power systems, authoring more than 70 papers in international journals and conferences.

SQL vs NoSQL in Polyglot Persistence Architectures

Florin-Răzvan SOARE, Anda-Elena SPĂTARU, Miruna ȘOȘEA
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies
Bucharest, ROMANIA
soareflorin21@stud.ase.ro, spataruanda21@stud.ase.ro, soseamiruna21@stud.ase.ro

Polyglot persistence is increasingly adopted because large-scale applications combine correctness-critical state with high-throughput, latency-sensitive workloads that cannot be served efficiently by a single datastore. This paper contrasts SQL and NoSQL systems in terms of transactional guarantees, recovery behavior, data modeling, query expressiveness, schema evolution, and operational scaling constraints. Based on this comparison, we derive datastore selection criteria that distinguish system-of-record components from derived serving models. We then discuss integration mechanisms for polyglot architectures, emphasizing explicit data ownership, change propagation via CDC and log-based replication, and saga-style coordination with compensations to manage cross-store failures.
Keywords: *Polyglot persistence, SQL, NoSQL, scalability, CDC, sagas, ACID, BASE*

1 Introduction

Data-intensive software rarely faces a single, uniform persistence problem. Modern applications must simultaneously support correctness-critical state changes and high-volume user-facing interactions, often under global traffic and continuous delivery constraints. Social networks and large consumer platforms illustrate this tension: the same product must enforce strict rules for identity and access control, apply payments and entitlements reliably, and maintain auditability for moderation decisions, while also serving personalized feeds with low latency, collecting telemetry at high throughput, and storing semi-structured content metadata that evolves as features change. These requirements impose different demands on storage engines. Some workloads require coordination to guarantee invariants, whereas others benefit from partition-local operations to sustain responsiveness.

Relational database management systems remain foundational because the relational model introduced a disciplined way to represent data and express queries while maximizing independence between application logic and machine-level representation [1]. This separation enables long-lived systems to evolve: storage layouts can change, indexes can be added, and physical organization can be optimized without rewriting application code, as long as the logical schema and constraints remain stable. Consequently, relational systems are well suited for authoritative datasets that must be shared across services and remain interpretable over time, such as financial records, account states, and compliance-sensitive entities.

At the same time, large-scale distributed environments treat partial failure as normal. Nodes crash, hardware degrades, latency fluctuates, and network partitions occur. CAP highlights why these conditions matter: in replicated shared-data systems, partitions force explicit trade-offs between strong consistency and availability for affected operations [5]. Brewer's later clarification emphasizes that the engineering task is to define acceptable behavior under partitions rather than rely on a simplistic "two out of three" interpretation [6]. This motivates separating correctness-critical state transitions, which can afford coordination,

from latency-sensitive serving paths that benefit from partition-local designs.

A further motivation for polyglot persistence is workload specialization. When a single general-purpose DBMS is used for all workloads, it may become overburdened by heterogeneous access patterns or force compromises that degrade performance and maintainability. The “one size fits all” critique argues that specialized engines often outperform general-purpose designs when workloads differ substantially [4]. As a result, modern persistence layers are frequently structured as a portfolio: a transactional system of record, a scalable store for events or documents, and derived read models tailored to user-facing queries.

Within this context, polyglot persistence refers to deliberately using multiple datastore technologies within the same application and assigning each technology to the domain component whose requirements best match its guarantees and performance profile [11].

Furthermore, the architectural shift from monolithic software design to distributed microservices has acted as a primary catalyst for the widespread adoption of polyglot persistence. In traditional monolithic architectures, a single, centralized database, typically relational, served as the universal integration layer for all application modules. While this centralized approach simplified operational maintenance, it created a tight coupling that hindered independent scaling, bottlenecked continuous delivery, and forced all workloads into a single data model.

Concurrently, the proliferation of cloud-native infrastructure and Database-as-a-Service (DBaaS) offerings has drastically lowered the operational barrier to managing heterogeneous datastores. Historically, provisioning, tuning, and maintaining high availability for multiple distinct database engines within an on-premises data center required prohibitive operational overhead and highly specialized personnel. Today, managed cloud environments allow teams to provision a highly available relational instance alongside a serverless, horizontally scalable NoSQL table within minutes [25]. This infrastructural democratization directly addresses the demands of the modern data ecosystem, which is characterized by the "3Vs": Volume, Velocity, and Variety. Because traditional, uniformly structured systems struggle to efficiently ingest high-velocity unstructured data streams while simultaneously executing complex relational transactions, a hybridized, polyglot approach to data persistence is no longer a luxury, but an engineering necessity.

This approach can reduce architectural friction, but it introduces an integration problem: once data is split across stores, correctness depends on clear ownership boundaries, propagation semantics, and well-defined failure behavior. To address these concerns systematically, the next sections establish the rationale for SQL and NoSQL choices, compare their trade-offs under realistic workloads, and then formalize integration patterns that preserve coherence across datastore boundaries.

2 Comparison between SQL and NoSQL

SQL and NoSQL systems reflect different assumptions about data, workloads, and failure modes, and the most useful - comparison is therefore workload-driven rather than categorical. SQL systems are rooted in the relational model and a declarative query language intended to preserve stable meaning while decoupling application logic from physical storage choices [1]. Many NoSQL designs target large-scale distributed environments where partial failures and network partitions are routine, and where limiting coordination can improve availability and responsiveness under disruption [5][6].

A first differentiator is how each family represents data and preserves semantics over time. In SQL, an explicit schema (types,

keys, constraints, and relationships) acts as a shared contract that constrains how data can be written and interpreted, which supports long-lived system evolution without silent semantic drift across teams or services [1]. In NoSQL, the data model depends on the family, such as key-value, document, wide-column, or graph databases, and schemas are often flexible or implicit. This flexibility reduces friction when payloads evolve rapidly, but it shifts semantic enforcement toward the application layer (validation, versioning, and monitoring), increasing the risk of divergence if governance is weak [12].

A second differentiator concerns concurrency and correctness, particularly for invariants that span multiple entities. SQL databases commonly provide transactions as a programming model for safe state transitions, supporting atomicity and durability so that multi-step updates do not expose partial effects under concurrency or failures [2]. This is central when business rules cannot be decomposed without risk, or when multiple records must change consistently. Mature recovery designs such as ARIES further reinforce this model by providing disciplined write-ahead logging with redo/undo recovery, improving confidence in crash recovery and interpretability after incidents [3]. NoSQL systems frequently limit coordination to keep operations partition-local; when invariants cross partitions or replicas, applications often rely on idempotent commands, conflict handling, and compensating workflows rather than assuming a single coordinated transaction for all affected data. This direction aligns with the argument that distributed transactions across heterogeneous components can be operationally fragile and costly at scale, motivating alternative composition models such as saga-style sequences of local transactions with compensations [14][15].

A third differentiator is behavior under network partitions and the associated consistency-availability trade-offs in distributed replication. CAP emphasizes that partitions are not hypothetical at scale; they are a reality that forces explicit design choices in replicated shared-data systems [5]. Brewer's later clarification stresses that the practical goal is to define acceptable behavior during partitions, rather than relying on a simplistic "two out of three" rule [6]. Many NoSQL designs embed these choices into replication and coordination strategies: by avoiding global coordination on the common path, they can remain available and responsive during partial failures, often at the cost of exposing temporary inconsistency. The concept of eventual consistency characterizes this inconsistency window and clarifies what the application must handle, including retries, reordering, and convergence, when replicas are not instantly synchronized [7]. SQL systems are more commonly used where strong consistency is expected for authoritative state; in such settings, the system may prefer coordination (and potentially reduced availability for certain operations) to avoid returning conflicting or stale results for correctness-critical decisions.

Performance differences are best framed in terms of which operations are optimized. SQL reads are typically strongest when the workload requires expressive, ad-hoc queries over relationships: joins, filtering across multiple entity types, and aggregations that support auditing, support investigations, and reconciliation. This follows directly from the relational model's objective of enabling high-level queries while allowing the system to choose physical strategies internally [1]. NoSQL reads are typically strongest when the workload is predictable and can be served from a single partition or a pre-composed representation. Document and key-value approaches can make reads fast by

fetching a whole denormalized object in one operation, avoiding join-time costs at the expense of duplication and more complex update propagation. Wide-column systems explicitly shape storage around partitioned access patterns (often key- and range-oriented) to achieve predictable serving behavior at large scale [9][10].

Write performance similarly depends on coordination and workload shape. SQL writes must often pay for transactional semantics, constraint checks, and index maintenance; under contention, concurrency control can add latency, but the benefit is centralized enforcement of invariants and strong recovery guarantees [2][3]. Many NoSQL systems emphasize high write throughput by making writes partition-local and by using replication strategies suited to always-on operation under failure. Dynamo's design exemplifies an availability-focused write/read approach in which replication and conflict resolution are designed to keep the system operating during node failures and network instability [8]. Cassandra emphasizes decentralized, failure-tolerant operation without a single point of failure and is frequently associated with ingestion-heavy patterns where the read model is designed around known access paths [10]. In practical terms, NoSQL tends to outperform when writes are high-volume, mostly independent by key/partition, and the system can tolerate weaker immediate consistency or needs to prioritize availability; SQL tends to outperform when writes must preserve multi-entity invariants and when correctness is more valuable than avoiding coordination. Scalability and tail latency under load are another axis where the design philosophies diverge. "One size fits all" critiques argue that general-purpose DBMS designs can be forced into compromises when serving heterogeneous workloads at extreme scale, motivating specialized systems for distinct access patterns [4]. Many NoSQL stores operationalize specialization by constraining query patterns and scaling horizontally through partitioning and replication, which helps preserve predictable tail latency for serving workloads and ingestion pipelines [8][9][10]. SQL systems can scale substantially, but the combination of cross-entity relational queries and strong invariants can complicate horizontal distribution, especially when constraints or joins span partitions.

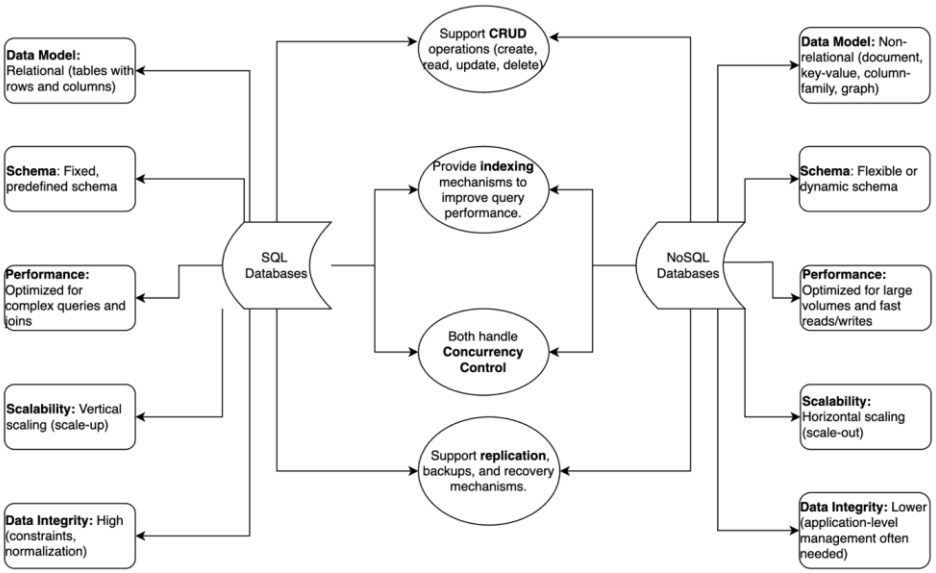


Fig. 3. Comparison between SQL and NoSQL Databases

To provide a clear and structured comparison of the concepts elaborated in the previous paragraphs, Figure 1 visually synthesizes the fundamental differences and shared characteristics between SQL and NoSQL databases. The diagram highlights how SQL systems are modeled in relational data models with fixed schemas, strong data integrity, and optimization for complex queries and joins[1], whereas NoSQL systems emphasize flexible schemas, horizontal scalability, and high-performance read/write operations for large-scale workloads[12]. At the same time, the illustration underscores important commonalities, such as support for CRUD operations, indexing mechanisms, concurrency control, and replication capabilities. By consolidating these aspects into a single conceptual view, the figure facilitates a more intuitive understanding of how the two paradigms diverge in design philosophy while converging on core database functionalities.
These contrasting strengths and trade-offs are not merely theoretical distinctions but have direct architectural implications. In practice, these trade-offs often motivate polyglot persistence, where different datastores are assigned to different bounded contexts according to their required guarantees and access patterns [11][13]. The remaining problem is ensuring coherent behavior once data and responsibilities span multiple stores, which motivates the integration mechanisms discussed next.

3 Advanced Data Modeling and Consistency Trade-offs

To fully appreciate the architectural divergence between relational and non-relational datastores, it is necessary to examine the underlying methodologies governing data modeling and the granular spectrum of consistency models. The transition from SQL to NoSQL is not merely a change in database engines; it dictates a fundamental paradigm shift in how developers interact with data and how distributed systems agree on state.

3.1 The Object-Relational Impedance Mismatch vs. Aggregate-Oriented Storage

Relational database models organize data into two-dimensional tables, requiring normalized relations mapped via foreign keys. However, modern application logic is overwhelmingly object-oriented, dealing with complex, nested, and hierarchical data structures. Bridging this gap requires Object-Relational Mapping (ORM) frameworks (e.g., Hibernate, Entity Framework). While ORMs simplify application development by abstracting SQL queries, they frequently introduce the "Object-Relational Impedance Mismatch." Complex object graphs require multiple resource-intensive joins or result in the notorious N+1 query problem, where the application executes a disproportionate number of secondary queries to fetch related entities.
Conversely, many NoSQL systems, specifically Document and Key-Value stores, adopt an aggregate-oriented storage model [21]. An aggregate is a collection of related objects that are treated as a single unit for data manipulation. By storing the entire aggregate as a single JSON or BSON document, NoSQL databases eliminate the impedance mismatch. An application can retrieve a complex entity (e.g., a student profile with an embedded list of historical test preferences) in a single disk seek. However, this optimization shifts the burden of data integrity to the application layer. If an embedded property changes (e.g., a teacher's contact email embedded within thousands of test records), updating it requires a bulk write operation across multiple documents, whereas a normalized SQL schema would require a single row update in a Teachers table.

3.2 Entity-Driven vs. Access-Driven Schema Design

The philosophical difference between SQL and NoSQL is most apparent during the schema design phase. Relational databases employ an *entity-driven* modeling approach.

The database is designed to reflect the pure relational state of the business domain, aiming for the Third Normal Form (3NF) to eliminate data redundancy and insertion anomalies. The exact queries that will be executed are treated as an afterthought, with the assumption that SQL's expressive power, coupled with secondary indexes, can efficiently join tables to answer any future ad-hoc analytical question.

NoSQL data modeling, particularly in wide-column and partitioned key-value stores like DynamoDB or Cassandra, mandates an *access-driven* (or query-driven) approach [22]. Because distributed databases lack native support for cross-partition joins, the database schema must be meticulously designed around the specific queries the application will execute. This gives rise to paradigms such as "Single-Table Design," where heterogeneous entity types (e.g., Students, Courses, and Enrollments) are co-located within the exact same table using generic partition keys and sort keys. By pre-computing joins and utilizing materialized paths or adjacency lists, NoSQL systems achieve predictable O(1) or O(log N) read complexity regardless of the dataset's size. However, this creates a highly rigid structure; any new access pattern not anticipated during the initial design phase may require a complete data migration or the creation of costly secondary indexes.

3.3 Granular Consistency Models and Quorum Mechanics

While the CAP theorem and the concept of "eventual consistency" provide a high-level vocabulary for distributed systems, practical polyglot architectures rely on granular, tunable consistency models based on quorum mechanics. The binary distinction between "strong" and "eventual" consistency is an oversimplification.

In distributed databases, consistency is often dictated by the replication factor (N), the write quorum (W), and the read quorum (R). If the system is configured such that R+W>N, it can guarantee strict consistency, as any read operation will overlap with the most recent write operation [8]. However, setting high read and write quorums significantly increases latency and reduces availability during node failures.

To optimize performance, NoSQL systems allow architects to lower the read quorum, intentionally returning data from a single, potentially out-of-date replica. Consequently, applications must be engineered to handle intermediate consistency states [23]. These include *Read-After-Write Consistency* (ensuring a user can see their own updates immediately, often achieved by routing the read to the leader node), *Monotonic Reads* (ensuring a user never reads an older version of data after having read a newer version), and *Consistent Prefix Reads* (ensuring operations are seen in the order they were applied). For example, DynamoDB offers developers a boolean parameter to request a strongly consistent read. Doing so bypasses the storage cache and consumes twice the read capacity units, illustrating the direct financial and performance cost of demanding relational-style strong consistency in a distributed environment.

3.4 Multi-Region Active-Active Topologies and Conflict Resolution

As cloud-native applications scale globally, polyglot persistence architectures frequently extend across multiple geographic regions to ensure low-latency access and robust disaster recovery. SQL databases typically employ asynchronous primary-replica architectures for cross-region replication. In this topology, the primary region remains the single source of truth; if it fails, writes are halted until a replica is promoted. This guarantees serializable transaction logs but creates a regional bottleneck for write throughput.

In contrast, distributed NoSQL systems often support Active-Active (multi-master)

topologies, such as DynamoDB Global Tables or Cassandra's multi-datacenter replication. These systems allow concurrent writes to the same logical record in different regions, maximizing availability but making write conflicts inevitable. Resolving these conflicts requires algorithmic interventions that do not rely on global locking. Common strategies include "Last Writer Wins" (LWW) based on physical or logical timestamps, or application-defined merge logic utilizing Conflict-free Replicated Data Types (CRDTs) [24]. CRDTs guarantee that regardless of the order in which network packets arrive, all database nodes will eventually converge to the exact same state without coordination. Understanding these conflict resolution heuristics is paramount when orchestrating saga compensations across diverse datastores, as delayed cross-region replication can inadvertently cause compensating transactions to execute against stale data, leading to severe logical corruption.

4 Experimental Setup and Case Study (PostgreSQL vs DynamoDB)

In practice, complex applications often require rich relationships between entities, which are naturally supported by relational databases through joins, constraints, and structured schemas. At the same time, some application features are largely independent of cross-entity relationships and can be better served by NoSQL systems optimized for high-throughput, low-latency access patterns. To illustrate why combining relational and non-relational datastores can be beneficial in polyglot persistence architectures, we designed a small empirical evaluation based on an e-learning application scenario.

In the reference e-learning application, the primary operational dataset is naturally relational: students, classes, tests, problem banks, and teacher-owned assessments form a strongly connected domain that benefits from a uniform schema, referential integrity, and expressive join-based queries. For these correctness-critical and relationship-heavy entities, a SQL database is the appropriate system of record.

In contrast, storing and serving a student's solved-test history is largely an append-only, read-by-student workload that does not require complex joins or cross-entity constraints at query time. This makes it a good candidate for a non-relational, partition-oriented store. Therefore, our evaluation focuses on this "history" component and compares write performance under concurrent load when implemented on PostgreSQL (Amazon RDS) versus DynamoDB, highlighting how polyglot persistence assigns each datastore to the workload it matches best.

We created a DynamoDB table using the default baseline settings available under the AWS Free Tier, without workload-specific tuning, and added a Global Secondary Index on the idStudent attribute.

idSolvedProblem (String)	answers	checkedAnswers	correctAnswers	idProblem	idStudent	mainQuestion	mark	problemType	questions	solvingDate
60000000899		[{"BOOL":true}...	[{"BOOL":false ...	1911654	1879223	Main: Bo9h5pu...	22	GRILA	[{"S":"Q:...	2025-12-30T11:53:33.275587+00:00
16000003557		[{"BOOL":false ...	[{"BOOL":false ...	1893290	371266	Main: sg94BWc...	100	GRILA	[{"S":"Q:...	2025-12-22T04:25:24.618345+00:00
23000000969	[{"S":"A: p...			619926	1017489	Main: W01ILvF...	89	NONGRILA	[{"S":"Q:...	2025-12-12T22:51:26.685552+00:00

Fig. 4. DynamoDB table structure

Figure 2 illustrates a snapshot of the resulting table within the AWS infrastructure, highlighting the semi-structured nature of the stored items. Each record encapsulates solved-test data, including identifiers, answer collections, correctness indicators, and metadata such as timestamps and scores, all organized in a flexible, attribute-based format. This representation reflects DynamoDB's

schema-less design, allowing heterogeneous attributes to coexist within the same table while supporting high-throughput insert operations and efficient retrieval via indexed access patterns.

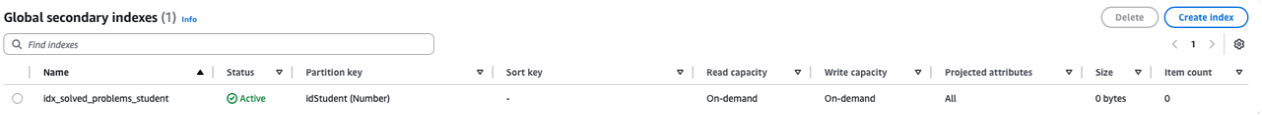


Fig. 5. DynamoDB index on idStudent

To illustrate the performance of joins and queries, we created a partitioning index. Figure 3 presents the configuration of the Global Secondary Index defined on the DynamoDB table. The index, named *idx_solved_problems_student*, uses *idStudent* as its partition key and operates in on-demand capacity mode, aligning with the baseline, auto-scaling configuration of the table. By projecting all attributes, the index enables efficient query execution without requiring additional lookups to the base table. This design supports the primary access pattern of retrieving solved-test history per student, illustrating how DynamoDB leverages secondary indexes to optimize read operations in scenarios where relational joins are not applicable[8].

The exploratory analysis on different architectural paradigms, includes an analysis on performance of a SQL database as a comparison to the previous example of NoSQL. Therefore, for PostgreSQL, we adopted a relational design that remains normalized while supporting heterogeneous problem formats (multiple-choice vs. free-text). Specifically, we created two tables:

1) `solved_problems`, which stores the shared "header" fields for each solved problem (e.g., `id_student`, `id_problem`, `mark`, `solving_date`, `problem_type`);

2) `solved_problems_data`, a volume table that stores variable-length components as rows (questions, selected answers, correct answers, or free-text answers).

This design allows each solved problem to have a different number of questions/answers without relying on JSON columns, while keeping the core metadata relational and queryable. To optimize the student-history use case, we created an index on `solved_problems.id_student` and an index on `solved_problems_data.id_solved_problem` (foreign key) to accelerate retrieval and joins. Figure 4 illustrates the table definition and indexes.

```
DO $$
BEGIN
  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'problem_type') THEN
    CREATE TYPE problem_type AS ENUM ('GRILA', 'NONGRILA');
  END IF;

  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'solved_data_type') THEN
    CREATE TYPE solved_data_type AS ENUM (
      'QUESTION',
      'CHECKED_ANSWER',
      'CORRECT_ANSWER',
      'ANSWER'
    );
  END IF;
END $$;

CREATE TABLE IF NOT EXISTS solved_problems (
  id_solved_problem BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

  main_question TEXT NOT NULL,
  id_student    BIGINT NOT NULL,
  id_problem    BIGINT NOT NULL,

  mark          SMALLINT NOT NULL CHECK (mark BETWEEN 10 AND 100),
  solving_date  TIMESTAMPTZ NOT NULL,

  problem_type  problem_type NOT NULL
);

CREATE TABLE IF NOT EXISTS solved_problems_data (
  id_solved_problem_data BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

  id_solved_problem BIGINT NOT NULL
    REFERENCES solved_problems(id_solved_problem)
    ON DELETE CASCADE,

  data_type solved_data_type NOT NULL,

  pos INT NOT NULL CHECK (pos >= 1),

  text_value TEXT,
  bool_value BOOLEAN,

  CONSTRAINT ck_value_by_type CHECK (
    (data_type IN ('QUESTION','ANSWER') AND text_value IS NOT NULL AND bool_value IS NULL)
    OR
    (data_type IN ('CHECKED_ANSWER','CORRECT_ANSWER') AND bool_value IS NOT NULL AND text_value IS NULL)
  )
);

CREATE INDEX idx_solved_problems_student
ON solved_problems (id_student);

CREATE INDEX idx_solved_problems_data_parent
ON solved_problems_data (id_solved_problem);
```

Fig. 4. Postgres tables and indexes code

As a comparison with the previously described DynamoDB configuration, figure 5 presents the configuration of the PostgreSQL instance deployed on AWS RDS. Similar to the DynamoDB setup, the database was provisioned using a baseline configuration within the AWS Free Tier, without workload-specific tuning. The instance is based on a *db.t4g.micro* class with limited compute and memory resources, single-AZ deployment, and general-purpose SSD storage with autoscaling enabled. Additional features such as encryption at rest, automated backups, and basic monitoring (Performance Insights and Enhanced Monitoring) are also configured by default. This setup reflects a typical lightweight relational deployment, providing a controlled environment for evaluating performance and behavior relative to the DynamoDB implementation under comparable resource constraints.

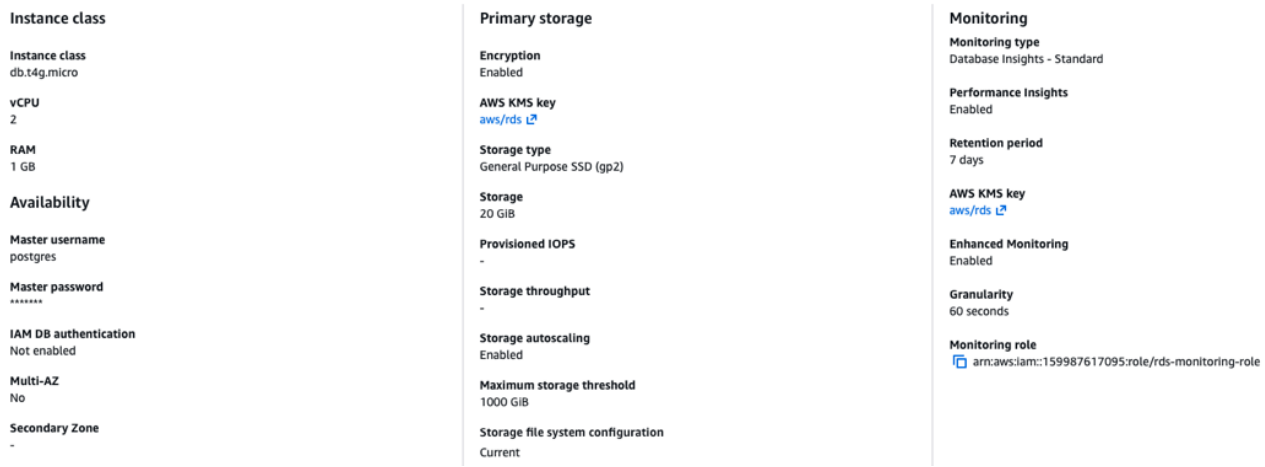


Fig. 5. Postgres database configuration

Building on the previously described infrastructure configurations, we designed a controlled experiment to evaluate and compare the insert performance of the two datastores. Specifically, we implemented a concurrent insert benchmark in Python, where multiple worker threads insert synthetic solved-problem records in batches to simulate a write-intensive workload. The benchmark measures total wall-clock time and derives end-to-end throughput (inserts per second) under load. To ensure that the results reflect database performance rather than data preparation overhead, the reported "insert-only" time excludes record generation, focusing exclusively on the duration of write operations. As illustrated in the implementation, timing is initiated immediately before the database write call and stopped after its completion, while synthetic data generation and mapping are performed outside the measured interval.

```
def main():  1 usage
    args = parse_args()
    target = args.target

    total_ops = WORKERS * PER_WORKER

    pool = None
    seq_name = None
    ddb_table = None

    if target == "sql":
        pool = ConnectionPool(conninfo=PG_DSN, min_size=1, max_size=PG_POOL_MAX, timeout=30)
        with pool.connection() as conn:
            seq_name = get_sequence_name(conn)

    if target == "nosql":
        session_kwargs = dict(
            aws_access_key_id=AWS_ACCESS_KEY_ID,
            aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
            region_name=AWS_REGION,
        )
        if AWS_SESSION_TOKEN:
            session_kwargs["aws_session_token"] = AWS_SESSION_TOKEN

        session = boto3.session.Session(**session_kwargs)
        ddb = session.resource( service_name: "dynamodb", config=DDB_BOTO_CONFIG)
        ddb_table = ddb.Table(DDB_TABLE)

    wall0 = time.perf_counter()

    insert_times = []
    with ThreadPoolExecutor(max_workers=WORKERS) as ex:
        if target == "sql":
            futures = [ex.submit(worker_sql,  *args: i, pool, seq_name) for i in range(WORKERS)]
        else:
            futures = [ex.submit(worker_nosql,  *args: i, ddb_table) for i in range(WORKERS)]

        for f in as_completed(futures):
            wid, ins = f.result()
            insert_times.append(ins)

    wall1 = time.perf_counter()
    wall = wall1 - wall0
```

Fig. 6. Main function of benchmark script

For reproducibility, the benchmark was executed separately for each datastore using the parameters *--target sql* and *--target nosql*. Both tests were run from the same client machine within the same AWS region (*eu-central-1*) to minimize network-induced variability. Each run used WORKERS = [114], PER_WORKER = [8000] (total inserts = WORKERS × PER_WORKER), and incremental batching with CHUNK = [500]. It is important to note that, in the PostgreSQL case, effective concurrency was constrained by the connection pool limit (PG_POOL_MAX = 15), whereas DynamoDB, due to its managed and distributed architecture, can accommodate a significantly higher level of concurrent client requests. For a better overview, of the script that performs the inserts, figure 6 depicts the main function of the benchmark script.

```python
def worker_sql(worker_id: int, pool: ConnectionPool, seq_name: str):  1 usage
    rng = random.Random(worker_id * 99991 + int(time.time()))
    insert_sec = 0.0

    done = 0
    while done < PER_WORKER:
        n = min(CHUNK, PER_WORKER - done)

        docs = [gen_submission(rng) for _ in range(n)]
        templates = [precompute_sql_templates(d) for d in docs]

        idx = 0
        while idx < len(templates):
            b = templates[idx: idx + SQL_BATCH_SIZE]
            t0 = time.perf_counter()
            insert_batch_postgres(pool, seq_name, b)
            t1 = time.perf_counter()
            insert_sec += (t1 - t0)
            idx += SQL_BATCH_SIZE

        done += n

    return worker_id, insert_sec


def worker_nosql(worker_id: int, ddb_table):  1 usage
    rng = random.Random(worker_id * 99991 + int(time.time()))
    insert_sec = 0.0

    base = (worker_id + 1) * 1_000_000_000

    done = 0
    while done < PER_WORKER:
        n = min(CHUNK, PER_WORKER - done)

        docs = [gen_submission(rng) for _ in range(n)]
        items = [make_ddb_item(str(base + done + i), d) for i, d in enumerate(docs)]

        t0 = time.perf_counter()
        insert_batch_dynamodb(ddb_table, items)
        t1 = time.perf_counter()
        insert_sec += (t1 - t0)

        done += n

    return worker_id, insert_sec
```

Fig. 7. Functions to calculate time for inserts

Building on the previously described benchmark setup, we will analyse both the implementation details and the resulting performance metrics obtained for each datastore. Figure 7 illustrates the core worker functions used in the experiment, highlighting how insert time is accumulated strictly around the database write operations for both PostgreSQL and DynamoDB. In particular, each worker processes records in batches (CHUNK = 500), measuring only the execution time of the insert calls (*insert_batch_postgres* and *insert_batch_dynamodb*), thereby ensuring consistency with the "insert-only" metric defined earlier. Figures 8 and 9 report the aggregated outputs for the NoSQL and SQL runs, respectively. Under the same client-side workload configuration (114 worker threads × 8,000 inserts each, CHUNK = 500), DynamoDB completed the run in **190.61 s** total wall-clock time and achieved **4,784.61 inserts/s**, with an average per-worker **insert-only** time of **177.07 s**. PostgreSQL (Amazon RDS) required **629.88 s** total wall-clock time, corresponding to **1,447.89 inserts/s**, with an average per-worker **insert-only** time of **624.97 s**. Overall, DynamoDB achieved approximately **3.3× higher throughput** than PostgreSQL in this append-only write scenario (4,784.61 vs. 1,447.89 inserts/s). This outcome is consistent with DynamoDB's partition-oriented design for high write concurrency, whereas PostgreSQL incurs additional overhead from transactional processing and index maintenance and, in our setup, was further bounded by the connection pool (**PG_POOL_MAX = 15**), effectively capping the level of database-side concurrency despite the higher number of client threads.

```
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison % python script.py --target nosql


TOTAL wall: 190.61s
Throughput end-to-end (submissions/sec): 4784.61
Avg worker insert_only: 177.07s
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison %
```

Fig. 8. Script output for NoSQL target parameter

```
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison % python script.py --target sql


TOTAL wall: 629.88s
Throughput end-to-end (submissions/sec): 1447.89
Avg worker insert_only: 624.97s
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison %
```

Fig. 9. Script output for SQL target parameter

The experimental results highlight the strong dependency between database performance and workload characteristics. In the evaluated append-only, write-intensive scenario, DynamoDB significantly outperformed PostgreSQL, achieving approximately 3.3× higher throughput. This outcome is consistent with the design principles of distributed NoSQL systems, which prioritize horizontal scalability, partitioned data distribution, and high write concurrency, as originally demonstrated in

systems such as Dynamo [8] and Bigtable [9]. By avoiding complex transactional coordination and leveraging eventual consistency models [7], DynamoDB can efficiently absorb a large volume of concurrent write requests.

In contrast, PostgreSQL's performance reflects the disadvantages of relational systems designed around strong consistency and transactional guarantees. Mechanisms such as write-ahead logging, locking, and index maintenance, fundamental to ensuring ACID properties [2][3], introduce additional latency during write operations. Furthermore, in the experimental setup, effective concurrency was limited by the connection pool, highlighting how resource management can become a bottleneck in vertically scaled systems. The results of the experimental benchmark reinforce the argument that "one size does not fit all" in data management [4]. While NoSQL systems excel in high-throughput, relationship-light workloads, relational databases remain better suited for scenarios requiring complex queries, strict consistency, and rich integrity constraints. Therefore, the observed performance gap should not be interpreted as a universal advantage of NoSQL, but rather as evidence of the importance of aligning data storage technologies with specific workload requirements.

5 Workload-Driven Data Management Implications

The experimental findings reinforce a fundamental principle in modern data management: datastore selection must be driven by workload characteristics rather than adherence to a single technological paradigm. As argued in prior work, the "one size fits all" approach is no longer viable in the presence of diverse application requirements and data access patterns [4]. Instead, systems must be designed by carefully aligning storage technologies with the operational profiles they are expected to support.

The evaluated scenario highlights a clear distinction between workload types. Append-only, write-intensive workloads with simple access patterns, such as the solved-test history analyzed in this study, benefit from NoSQL systems optimized for horizontal scalability and high ingestion throughput. These systems typically relax strict consistency guarantees in favor of availability and partition tolerance, consistent with the trade-offs described by the CAP theorem [5][6] and the eventual consistency model [7]. In contrast, workloads involving complex relationships, integrity constraints, and transactional semantics remain better suited to relational databases, where ACID properties and structured schemas ensure correctness and consistency [2].

This contradictory division suggests that data modeling should not be approached uniformly across an application. Instead, different components of the same system may exhibit fundamentally different requirements: some demand strong consistency and relational integrity, while others prioritize scalability, latency, and flexibility. As a result, architectural decisions should begin with a precise characterization of workload dimensions, including read/write ratios, concurrency levels, data interdependencies, and query complexity.

More important, these implications extend beyond performance considerations. Choosing an inappropriate datastore for a given workload can lead to unnecessary complexity, either by forcing relational systems to scale beyond their intended design or by requiring NoSQL systems to emulate relational behavior at the application level. Therefore, effective data management requires not only selecting the right tool for each task but also acknowledging the boundaries of each paradigm.

These observations naturally motivate the adoption of polyglot persistence, where multiple datastores coexist within a system, each serving a specific role aligned with its strengths. The following section explores how such heterogeneous systems can be integrated while preserving consistency and operational reliability.

Beyond performance and scalability limits, polyglot persistence fundamentally alters the financial and operational model of data management. The experimental setup highlights two distinct provisioning paradigms: capacity-based provisioning (PostgreSQL on RDS) and consumption-based, or serverless, scaling (DynamoDB).

In traditional relational deployments, resources (CPU, RAM, storage IOPS) must be provisioned to handle peak expected loads. This often leads to underutilization during off-peak hours, creating financial inefficiency. Furthermore, vertically scaling a relational database to alleviate connection pool bottlenecks, as observed in our PostgreSQL benchmar, incurs a steep cost curve and potential downtime during instance resizing.

In contrast, managed NoSQL systems like DynamoDB offer on-demand capacity models that align costs directly with application traffic. You pay per read/write request unit, which is highly cost-effective for spiky, high-throughput ingestion workloads such as telemetry or our test-history scenario. However, this financial model introduces new risks: poorly designed partition keys can lead to "hot partitions," causing massive read/write throttling and sudden cost spikes. Therefore, workload-driven data management must also encompass a "FinOps" (Financial Operations) perspective [18]. A polyglot architecture is often economically justified when the cost of offloading high-throughput, simple-query traffic to a serverless NoSQL datastore is lower than the cost of over-provisioning a monolithic SQL cluster to handle the same traffic alongside complex transactions.

6 Integrating SQL and NoSQL in Polyglot Persistence Architectures

Once multiple datastores are introduced, correctness depends on making ownership and propagation semantics explicit. A useful baseline is to designate a system of record for each business fact: exactly one datastore is authoritative for a given domain concept, while other stores contain derived projections, indexes, or caches built from that source [11][13]. This avoids “dual truth” scenarios in which multiple systems accept independent updates for the same fact, creating inconsistencies that are difficult to detect and reconcile. The integration question then becomes how committed changes from the system of record are propagated and applied to downstream models under failures and retries.

In the context of the evaluated e-learning system, this principle can be concretely applied by treating the relational database (PostgreSQL) as the system of record for core entities such as students, tests, and results, where strong consistency and integrity constraints are required. In contrast, DynamoDB acts as a derived store that maintains a denormalized, append-only history of solved tests per student, optimized for fast retrieval. Under this model, updates are never performed independently in both systems; instead, changes originate in the system of record and are subsequently propagated to downstream projections. The integration challenge then becomes ensuring that committed changes are reliably transferred and applied across systems, even in the presence of failures and retries.

Change propagation is a second pillar of integration. Derived stores must be kept sufficiently up to date for their intended user journeys, which requires reliable synchronization mechanisms. Log-based replication and change data capture are commonly used

to propagate committed changes incrementally, enabling downstream stores to update projections with low latency [16]. For example, when a student completes a test, the result is first persisted in PostgreSQL, after which a corresponding change event (e.g., *student_solved_test_created*) can be emitted and consumed to update the DynamoDB projection. However, CDC introduces engineering challenges: extracting and interpreting log records at scale is complex, and downstream consumers must address ordering, replay, and idempotent application to avoid duplicate effects [16]. Robust implementations therefore rely on consumer offsets, replay capability, dead-letter handling, and observability metrics such as lag and error rates.

To formalize the unidirectional flow of data from the system of record to derived stores, modern polyglot architectures frequently adopt the Command Query Responsibility Segregation (CQRS) pattern paired with Event Sourcing [19]. CQRS structurally separates the write models (Commands), handled by a strongly consistent SQL database to validate business rules, from the read models (Queries), which are materialized in NoSQL databases for high-speed delivery. Instead of just persisting the current state, Event Sourcing involves storing a sequence of immutable state-changing events. In our e-learning context, instead of merely capturing a row update, the system records an explicit domain event (e.g., TestGraded, MarkAssigned). This immutable event log acts as the ultimate source of truth, from which the DynamoDB projection can be deterministically rebuilt if destroyed or structurally altered.

Furthermore, integrating heterogeneous databases amplifies data governance and compliance complexities, particularly concerning data privacy frameworks like the GDPR. In a centralized relational database, executing a "Right to be Forgotten" request involves executing a single DELETE statement with cascading foreign keys. In a polyglot architecture driven by CDC and eventual consistency, a single deletion must propagate reliably through message brokers, updating not only the SQL system of record but also purging the corresponding denormalized records in the NoSQL serving layers and any downstream data lakes [20]. This necessitates automated reconciliation jobs that periodically sweep derived stores to detect and remediate orphaned data, preventing silent compliance violations.

Cross-store operations introduce a third challenge. While distributed transactions appear attractive, many large-scale systems avoid them due to coordination overhead and operational fragility, relying instead on message-driven workflows and compensating actions [14]. In this context, sagas provide a formal model: long-lived business activities are implemented as a sequence of local transactions paired with compensations that restore invariants when later steps fail [15]. In practice, this can be illustrated by a workflow where a solved test is first recorded in the SQL system, then propagated to the NoSQL store, and finally used to trigger user-facing updates. If one of the later steps fails, compensating actions, such as retrying the projection update or marking the operation for reconciliation, ensure that the system eventually converges to a consistent state. This approach requires explicit state management, idempotent command handling, and clearly defined recovery strategies.

Finally, broader work on polyglot data management highlights that heterogeneity increases integration complexity and makes semantic alignment a first-order concern [17]. Even when integration is implemented at the application layer, similar issues recur: mapping data models, reconciling inconsistencies, and managing operational complexity across systems with different guarantees. Resilient polyglot architectures therefore combine datastore specialization with explicit contracts for ownership, consistency

expectations, propagation guarantees, and recovery strategies. While this added complexity introduces engineering overhead, it enables systems to leverage the strengths of both relational and NoSQL paradigms, achieving a balance between consistency, scalability, and performance that would be difficult to obtain with a single datastore.

7 Conclusion

This paper demonstrates that database selection in a polyglot architecture should be fundamentally driven by workload characteristics rather than adherence to a single paradigm. Starting from the well-established distinction between relational and non-relational systems, the study shows that their differences are not merely theoretical but have direct implications for system design and performance. In the evaluated e-learning scenario, the core domain is most effectively modeled using SQL, where strong relational structures and integrity constraints are essential for maintaining consistency and correctness. In contrast, the solved-test history represents an append-only, student-centric access pattern that is inherently read-optimized and does not require complex joins at query time.

Empirical evaluation under identical concurrent insert workloads on AWS indicates that DynamoDB provides superior ingestion throughput compared to PostgreSQL (RDS) for this specific component. This result reflects the architectural differences between the two systems: DynamoDB's distributed, partition-oriented design enables efficient handling of high write concurrency, while PostgreSQL incurs additional overhead due to transactional processing, index maintenance, and connection management. However, these findings should be interpreted in the context of the evaluated workload, as relational systems remain more suitable for scenarios requiring complex queries, strong consistency, and rich data relationships.

Overall, the results reinforce the argument that modern applications benefit from combining multiple data storage paradigms within a unified architecture. Polyglot persistence allows each datastore to be used according to its strengths, enabling systems to achieve both scalability and correctness without compromising on either dimension. At the same time, this approach introduces additional complexity in terms of data integration, consistency management, and operational overhead, requiring careful design of ownership boundaries and data propagation mechanisms.

Furthermore, this research underlines that architectural decisions in data management extend beyond technical metrics to include operational predictability, cost efficiency, and data governance. While integrating systems via CDC, CQRS, and sagas mitigates the risks of distributed data, it shifts complexity from the database engine to the application and infrastructure layers. Teams must be prepared to handle eventual consistency, cross-store auditing, and complex compliance enforcement (such as distributed data deletion).

Future work may extend this study by evaluating mixed workloads, read performance, and consistency trade-offs under different deployment configurations, further refining the guidelines for workload-driven data management.

References

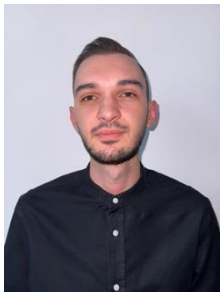
[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf, 1970.

[2] J. Gray, "The Transaction Concept: Virtues and Limitations," Tandem TR 81.3, https://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf, 1981.

[3] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting

Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems*, https://web.stanford.edu/class/cs345d-01/rl/aries.pdf, 1992.
[4] M. Stonebraker et al., "'One Size Fits All': An Idea Whose Time Has Come and Gone," *ICDE*, https://cs.brown.edu/~ugur/fits_all.pdf, 2005.
[5] E. A. Brewer, "Towards Robust Distributed Systems," PODC Keynote, https://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf, 2000.
[6] E. A. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed," *Computer*, https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/brewer-cap.pdf, 2012.
[7] W. Vogels, "Eventually Consistent," *Communications of the ACM*, https://15799.courses.cs.cmu.edu/fall2013/static/papers/p40-vogels.pdf, 2009.
[8] G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *SOSP*, https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf, 2007.
[9] F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," *OSDI*, https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf, 2006.
[10] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Operating Systems Review* / LADIS, https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf, 2010.
[11] P. J. Sadalage and M. Fowler, "*NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence,"* Addison-Wesley, https://ptgmedia.pearsoncmg.com/images/9780321826626/samplepages/0321826620.pdf, 2012.
[12] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison," https://arxiv.org/pdf/1307.0191, 2013.
[13] P. P. Khine and Z. Wang, "A Review of Polyglot Persistence in the Big Data World", *Information*, https://www.mdpi.com/2078-2489/10/4/141, 2019.
[14] P. Helland, "Life Beyond Distributed Transactions: An Apostate's Opinion," *CIDR*, https://ics.uci.edu/~cs223/papers/cidr07p15.pdf, 2007.
[15] H. Garcia-Molina and K. Salem, "SAGAS," *ACM SIGMOD*, https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf, 1987.
[16] D. Butterstein et al., "A Fast, Scalable Replication Solution for Near Real-Time (CDC/log-based replication)," *PVLDB*, https://www.vldb.org/pvldb/vol13/p3245-butterstein.pdf, 2020.
[17] F. Kiehn et al., "Polyglot Data Management: State of the Art & Open Challenges," *PVLDB*, https://www.vldb.org/pvldb/vol15/p3750-panse.pdf, 2022.
[18] J. Schleier-Smith et al., "What Serverless Computing Is and Should Become," *Communications of the ACM*, vol. 64, no. 5, pp. 76-84, 2021.
[19] M. Fowler, "CQRS," *MartinFowler.com*, https://martinfowler.com/bliki/CQRS.html, 2011.
[20] J. Duncan and C. O'Brien, "The engineering challenges of data privacy and GDPR compliance in distributed systems," *IEEE International Conference on Cloud Engineering (IC2E)*, https://www.researchgate.net/publication/351336495_The_Engineering_Challenges_of_Data_Privacy_and_GDPR_Compliance_in_Distributed_Systems ,2021.
[21] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of

Software," Addison-Wesley Professional, https://www.domainlanguage.com/ddd/reference/ , 2003.
[22] A. Chebotko, A. Kashlev, and S. Lu, "A Big Data Modeling Methodology for Apache Cassandra," *IEEE International Congress on Big Data*, https://shiyong.eng.wayne.edu/papers/bigdata2015_andrey.pdf, 2015.
[23] D. B. Terry et al., "Session Guarantees for Weakly Consistent Replicated Data," *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS)*, https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p140-terry.pdf, 1994.
[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," *Symposium on Self-Stabilizing Systems (SSS)*, https://hal.inria.fr/inria-00609399/document, 2011.
[25] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed., O'Reilly Media, https://samnewman.io/books/building_microservices_2nd_edition/, 2021.



Răzvan-Florin SOARE earned his bachelor's degree in Economic Informatics in 2024 and now he is a master's student at the Academy of Economic Studies from Bucharest, Databases – Support for Business. He will graduate from this program in 2026. Professionally, he works as a Dabase Developer in the banking sector, where he designs, optimizes, and manages complex data systems. Beyond his core expertise, he is deeply passionate about artificial intelligence and process automation.

Anda Elena Spătaru graduated at the Bucharest University of Economic Studies in 2024, earning a Bachelor's degree in Economic Informatics. She is currently pursuing a master's degree in Databases for Business Support, expected to be completed in 2026. Professionally, Anda has contributed to the development of digital solutions that enhance business efficiency and support data-driven decision-making. She began her career as a Data Engineer at PPC, later transitioning into data analysis and business intelligence development.

Miruna Șoșea graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Academy of Economic Studies in 2024, obtaining a Bachelor's Degree in Economic Informatics. She is currently pursuing a master's degree in Databases – Business Support and is expected to graduate in 2026. She currently works as a Data Engineer, focusing on data pipelines, data warehousing, and large-scale data processing. Her main areas of interest include data engineering, data analysis, distributed systems, and database management.