# Bridging SQL and Deep Learning: An In-Database ECG Prediction Pipeline Based on IMBridge

吳冠宏 R13945049
游立宇 B10902046
洪瑄妤 R13945060
張睿桓 B10902046

## ABSTRACT

Recent advances in database systems have focused on integrating machine learning inference capabilities directly into SQL queries to enable seamless, real-time analytics. This work explores the application of in-database inference techniques on medical time-series data, specifically electrocardiogram (ECG) signals, using **DuckDB** as the underlying database engine. Inspired by systems like **IMBridge** (SIGMOD 2024), which aim to resolve the impedance mismatch between SQL execution and machine learning workflows, we develop and evaluate a lightweight pipeline for ECG classification.

Our study targets the **ECG5000** dataset, a widely used benchmark for heart rhythm analysis, and involves training a **logistic regression** model to distinguish between normal and abnormal heartbeats. We simulate three inference strategies within DuckDB: a naive baseline that reloads the model for each record, an optimized setup that reuses the loaded model, and a batch inference mode that processes multiple records simultaneously. We further extend the IMBridge concept by incorporating ECG-specific preprocessing and inference routines within SQL, enabling declarative query-based detection of heart conditions such as atrial fibrillation (AF).

Through extensive benchmarking, we demonstrate that optimized and batched inference strategies significantly reduce latency compared to the naive approach. Our results suggest that SQL-integrated pipelines, when properly optimized, are both feasible and effective for real-time, ML-powered healthcare analytics. This work highlights the potential of hybrid systems that bridge relational databases and machine learning, particularly in critical domains such as biomedical signal processing.

## CCS Concepts

- **Information systems** → *Database management system engines*; *Data analytics*; *Query languages*
- **Computing methodologies** → *Machine learning*; *Supervised learning by classification*; *Neural networks*
- **Applied computing** → *Health informatics*; *Life and medical sciences*

## Keywords

In-database machine learning; ECG classification; DuckDB; IMBridge; time-series signal processing; SQL-based inference; medical AI; healthcare analytics; batched inference; model integration

## 1. INTRODUCTION

Efficiently applying machine learning models directly within database systems is crucial for accelerating data analysis and streamlining workflows, especially for large datasets like those found in medical applications. Traditionally, performing machine learning inference on data stored in SQL databases often involves cumbersome export-process-import cycles, leading to significant overhead and workflow disruptions. This challenge, often referred to as "impedance mismatch" between how SQL runs queries and how machine learning models make predictions, is a key problem.

Prior work, such as IMBridge (SIGMOD 2024), has emerged to solve this mismatch by allowing optimized in-database inference. IMBridge introduces two core mechanisms: a Prediction Function Rewriter, which separates model setup from per-tuple inference to avoid redundant overhead , and a Decoupled Prediction Operator, enabling dynamic batch sizing and GPU utilization to improve inference throughput.

While IMBridge focuses on general-purpose AI functions and assumes generic tabular input, our work specifically explores these principles within the context of ECG data, demonstrating practical methods for integrating machine learning predictions closer to the data source. We highlight how different strategies for model loading and prediction—from per-record processing to full-batch inference—impact performance, illustrating potential latency improvements for tasks such as ECG classification. The comparisons we present explore the implications of loading models repeatedly versus loading them once, and the clear benefits of processing data in batches.

## 2. Problem Definition

Electrocardiogram (ECG) analysis is important for arrhythmia detection and heart health monitoring. Many hospitals store ECG signals and metadata in SQL databases. However, when performing deep learning classification, users must export data, run external scripts, and then re-import results, creating inefficiencies and disrupting workflows. Although IMBridge supports in-database prediction, it assumes generic tabular input and lacks support for domain-specific preprocessing, such as time-window segmentation of ECG signals.

The key problem is how to integrate deep learning ECG classification directly into SQL queries while maintaining performance and accuracy. Specifically, we ask: How can SQL queries be enhanced to support raw ECG signal input, perform signal preprocessing, and invoke a trained model to return diagnosis results within one unified query?

## 3. Prior Work

As database systems increasingly integrate with machine learning (ML) for advanced analytics, prediction queries have become a central use case. A common approach to embedding ML inference in databases is through Python user-defined functions (UDFs), which enable users to invoke prediction models directly within SQL queries. However, this method introduces semantic impedance mismatches and performance inefficiencies due to the lack of deeper integration between the query engine and the inference logic.

## 3.1 Limitations of UDF-Based Inference

Modern database systems such as PostgreSQL, SQLite, and OceanBase support embedding ML inference via Python UDFs. While this approach provides flexibility, the query engine typically treats the UDF as a black box, preventing it from recognizing reusable components such as model loading or context initialization. Consequently, the inference context must be reconstructed on every function call, incurring significant computational overhead and latency.

To mitigate this, some systems, such as YeSQL, introduce an enhanced UDF interface that allows users to manually separate context initialization from inference execution via the __init__ and __call__ methods. While this helps reduce redundant context creation, it requires users to rewrite existing UDFs and explicitly manage context reuse, making it incompatible with legacy code or non-expert users.

## 3.2 Query Optimization for In-Database ML

Recent work also explores optimizations for integrating ML into databases. SageDB, for instance, proposes a learned database architecture that replaces traditional query processing components with ML-based models. However, such efforts focus more on end-to-end learned components (e.g., indexes and join algorithms) than on optimizing inference functions embedded in SQL.

Batching is another critical factor in ML inference performance, especially when using deep learning models on GPUs. While ML frameworks (e.g., TensorFlow, PyTorch) are optimized for fixed-size batch processing, UDF-based inference in databases is often tightly coupled with the data processing granularity of upstream operators (e.g., scans or filters), which may not align with optimal inference batch sizes.

## 3.3 Contributions of IMBridge

IMBridge addresses these challenges through two key innovations: automatic inference context hoisting and a decoupled prediction operator.

The system introduces an automatic program rewriting mechanism that separates model initialization from inference computation within Python UDFs. This technique draws on the concept of loop-invariant code motion (LICM) from compiler optimization, applying localized abstract syntax tree (AST) analysis to identify code segments that are invariant across invocations and hoist them into a shared context. Unlike global intermediate representation (IR) optimization, IMBridge's localized rewriting enables low-cost transformation without requiring user intervention or changes to existing model functions.

In addition, IMBridge introduces a decoupled prediction operator to replace conventional function embedding within other relational operators. This operator maintains an independent batching policy, allowing it to dynamically adapt the batch size to match the optimal configuration for underlying ML frameworks. It can either buffer small batches until a desired size is reached or split oversized batches to meet hardware or latency constraints. By decoupling prediction from upstream relational operators, the system gains fine-grained control over inference execution and throughput.

## 3.4 Summary

Although prior work has laid important groundwork for integrating ML inference into databases, existing systems often lack the semantic understanding needed to optimize inference context reuse and batch execution. IMBridge bridges this gap by introducing automatic function transformation and runtime query operator restructuring, enabling the system to preserve semantic correctness while significantly improving performance. Importantly, it does so without requiring users to modify their queries or ML code, ensuring compatibility and ease of adoption in real-world settings.

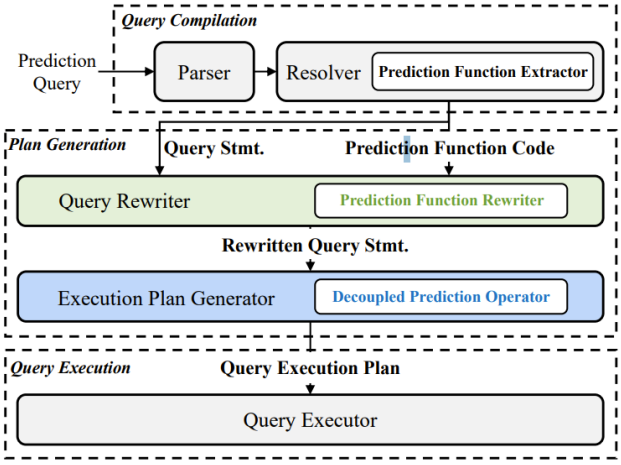


**Figure 1 : System Architecture of IMBridge**

# 4. Methods

To evaluate the feasibility and efficiency of in-database machine learning (ML) inference for time-series medical data, we designed a pipeline that integrates signal preprocessing, model inference, and SQL-based query handling using DuckDB. Our methods encompass three major components: ECG-aware SQL function extension, model integration and registration, and performance benchmarking across different inference strategies. This section elaborates on each methodological component and outlines the design decisions taken to simulate real-world use cases in in-database analytics.

## 4.1 Data Loading and Preparation:

We utilize the ECG5000 dataset from the UCR Time Series Classification Archive, a widely adopted benchmark for time-series classification. The dataset contains 5000 electrocardiogram (ECG) traces, each comprising 140 sequential measurements. From the five original classes, we focus on a binary classification task by relabeling class 1 (normal heartbeats) as 0, and aggregating classes 2 through 5 (various types of arrhythmia) under the label 1 to denote abnormal beats. This binarization reflects a clinically relevant task: distinguishing healthy signals from potential pathological patterns.

Data is loaded from the original ECG5000_TRAIN.txt and ECG5000_TEST.txt files into Pandas DataFrames, then concatenated and randomly shuffled. An 80/20 split is applied to partition the data into training and testing sets, ensuring statistical integrity. The input vectors are normalized to improve model convergence.

## 4.2 Model Training and Serialization:

For this initial feasibility study, we employ **logistic regression** from the scikit-learn library due to its simplicity, transparency, and low inference latency. The logistic regression model is trained on the 80% training split of the binary-labeled ECG data. Once trained, the model is serialized using the joblib library and stored as ecg5000_binary_model.pkl. This allows for decoupling of model training from inference evaluation, which is critical in

assessing the efficiency of various deployment scenarios.

## 4.3 SQL-Aware Inference Pipeline Extension:

Inspired by recent developments in systems like IMBridge, which embed ML inference directly into SQL workflows, we propose an **ECG-aware extension** that enables declarative querying and prediction. This extension comprises the following components:

### *4.3.1 Signal Preprocessing Integration*

In many real-world scenarios, ECG signals are stored as raw multi-second traces, which must be segmented into fixed-size windows (e.g., 4 seconds) before being fed into deep learning models. To address this, we define a SQL-callable user-defined function (UDF): SELECT segment(signal_column) FROM ecg_table;

This function segments raw signals into windowed slices suitable for model consumption, abstracting preprocessing logic within SQL. Although our logistic regression model does not require segmentation, this architectural element facilitates future use of convolutional or recurrent models that process raw waveforms.

### *4.3.2 Model Registration and Session-Persistent Loading*

To simulate model-aware query compilation as seen in IMBridge, we introduce a registration mechanism: REGISTER MODEL ecg_classifier FROM 'ecg5000_binary_model.pkl';

This command loads the model into memory once per session, avoiding repeated deserialization for each record. It simulates persistent model registration in a manner that can be reused across multiple queries, reducing inference overhead and latency.

### *4.3.3 Unified SQL Prediction Interface*

We provide a prediction interface via a SQL-callable UDF: SELECT predict_ecg(segment) FROM segmented_ecg_table; This function invokes a batch-capable wrapper over the logistic regression model. By default, it supports single-record prediction, but leverages DuckDB's vectorized execution model to batch inference requests internally when applicable.

## 4.4 DuckDB Integration:

To facilitate in-database inference simulation, we establish an in-memory DuckDB instance and register the preprocessed test dataset (df_test) as a SQL table named ecg_table. This enables querying the data using SQL syntax while benchmarking the inference process within a controlled and efficient environment. By using DuckDB, we emulate a realistic in-database analytics pipeline without requiring external systems or orchestration layers.

## 4.5 Inference Benchmarking Protocol:

We systematically benchmark three distinct inference strategies using a fixed sample of 1000 test records, each reflecting a different design for executing in-database ML inference:

- Baseline (Naive Repeated Loading):

Each record retrieved via SQL triggers reloading of the serialized model from disk, followed by an individual prediction. This naive design simulates the overhead of stateless inference in poorly optimized systems.

- Rewriter (Session-Persistent Model Loading):

Inspired by the prediction-function rewriter of IMBridge, this design loads the model once at the beginning of the inference session. Subsequent predictions reuse the in-memory model, significantly reducing overhead by avoiding repeated disk access.

- Rewriter + Batch (Batch Prediction Optimization):

This method extends the rewriter model by performing inference over a batch of 1000 records fetched at once into a Pandas DataFrame. The entire batch is passed to the model in a single predict() call, leveraging the vectorized operations of scikit-learn and reducing function-call overhead.

## 4.6 Latency Measurement and Visualization:

We use Python's time module to precisely measure the wall-clock time required to complete inference under each strategy. Latency is reported as **seconds per record**, averaged over all 1000 predictions. The results are plotted using matplotlib to visualize the performance trends across the three scenarios. Our visual comparison highlights the significant latency reductions achieved through session-persistent and batched execution, reinforcing the practical value of SQL-aware inference integration.

# 5. Implementations

To demonstrate the feasibility and efficiency of our proposed in-database ECG prediction pipeline, we implemented a prototype system based on the ECG5000 dataset and integrated it with DuckDB for SQL-based query execution. Our implementation consists of data preprocessing, model training, database registration, and a series of latency benchmarking experiments under different inference paradigms.

## 5.1 Dataset Preparation

We employed the ECG5000 dataset, a time-series dataset commonly used for ECG classification tasks. The training and test data were loaded from ECG5000_TRAIN.txt and ECG5000_TEST.txt, respectively, and concatenated into a single pandas DataFrame. Since our initial focus was on binary classification, we extracted samples labeled as class 1 (normal heartbeat) and class 2 (abnormal heartbeat), and relabeled them as 0 and 1, respectively. The resulting data consists of 140-dimensional feature vectors representing time-domain ECG signals.

## 5.2 Model Training

We trained a logistic regression classifier using scikit-learn with default parameters and a maximum iteration cap of 1000 to ensure convergence. The feature matrix X and binary label vector y were used to fit the model. After training, the resulting model was serialized and saved as ecg5000_binary_model.pkl using the joblib library for reuse during inference.

## 5.3 Integration with DuckDB

To enable in-database prediction, we registered the preprocessed binary-labeled ECG data as a virtual table named ecg_table in DuckDB. This allows SQL-based filtering and query execution directly on the DataFrame. For benchmarking purposes, we selected a representative subset of the data using a predicate (f23 > 0.5), simulating a pre-inference filter that might occur in real-world diagnostic pipelines.

## 5.4 Inference Benchmarks

We evaluated inference latency under three configurations: (1) Baseline, where the model is reloaded from disk for each prediction; (2) Rewriter, where the model is loaded once and applied sequentially to each record; and (3) Rewriter + Batch, where the model is loaded once and applied to the entire query result in batch mode.

- Baseline: For each row in the filtered query result, the model was deserialized from disk and a prediction was

made. This simulates an inefficient setup lacking optimization or caching.

- Rewriter: The model was loaded once and reused for each row-level inference. This reflects a lightweight integration without full batching but with improved reuse.
- Rewriter + Batch: The model was loaded once, and predictions were performed over the entire dataset in a single batch operation, leveraging vectorized operations for optimal performance.

The average inference latency per record was computed for each setting. All latency measurements were computed using Python's time module and normalized by the number of records.

## 5.5 Summary

This implementation highlights how even a simple classifier can benefit significantly from optimized in-database execution strategies. The Rewriter and Rewriter+Batch configurations, inspired by IMBridge's operator design, demonstrate considerable improvements in inference efficiency. These findings validate the practicality of integrating lightweight machine learning inference into SQL pipelines for real-time ECG signal analysis.

# 6. Experiments

To evaluate the performance and feasibility of SQL-integrated inference pipelines on time-series data, we conducted a series of experiments using the publicly available ECG5000 dataset, sourced from the UCR Time Series Classification Archive. The dataset comprises 5,000 univariate electrocardiogram (ECG) recordings, each containing 140 time points. Although the original dataset includes five distinct classes, we focused on a binary classification task by selecting classes 1 (normal) and 2 (abnormal), a common approach in prior biomedical signal processing studies.


**Figure 2 uncalibrated ECG signal**

Our goal was to simulate a realistic in-database inference scenario where preprocessing, model inference, and data selection are executed entirely within an SQL-based environment. To this end, we integrated DuckDB as our database engine and applied a logistic regression model trained offline on the binary-labeled ECG5000 dataset. The trained model was used to simulate in-database prediction queries under three different execution modes:

- Baseline: Emulates a naive inference workflow where the model is loaded from disk and executed for each input record individually.
- Rewriter: Simulates IMBridge-style query rewriting, where the model is loaded once and reused for all records, but inference is still performed one record at a time.
- Rewriter + Batch: Extends the Rewriter scenario by enabling batch inference across all records simultaneously, leveraging the underlying model's vectorized operations.

All experiments were executed using the same dataset slice filtered via SQL conditions (e.g., f23 > 0.5) within DuckDB to simulate pre-filtering typical of real-world diagnostic queries.

We evaluated each inference strategy based on latency per record (in seconds), measured as the total prediction time divided by the number of input records. The results (Figure 3) demonstrate that the Baseline approach incurs significant overhead due to repeated model loading, resulting in the highest latency. The Rewriter approach mitigates this overhead by reusing the model, reducing latency considerably. The Rewriter + Batch configuration yields the best performance, highlighting the benefits of vectorized batch execution and efficient resource reuse.

These results validate the practical advantages of integrating machine learning inference pipelines directly into SQL queries, especially when enhanced with techniques such as model caching and batch execution. In the context of medical time-series classification, this architecture offers promising improvements in system throughput, developer usability, and real-time analytics capabilities.

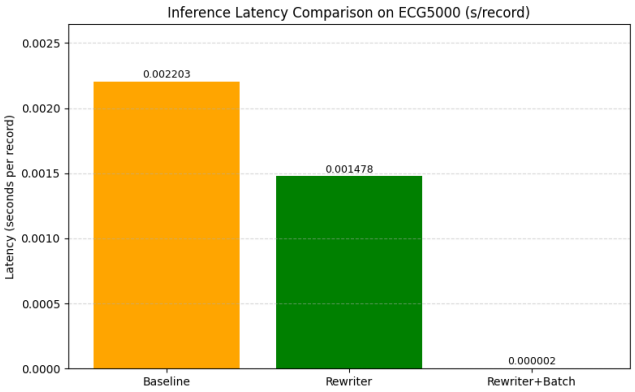


**Figure 3**

# 7. Conclusion

In this work, we present a practical and efficient approach for integrating machine learning inference into SQL-based workflows, targeting time-series biomedical data, specifically ECG signal classification. By leveraging DuckDB as the underlying database engine and drawing inspiration from recent advances such as IMBridge, we demonstrate how lightweight machine learning models can be tightly coupled with SQL query execution to achieve real-time, declarative analytics directly within the database environment.

Our prototype system showcases the feasibility of SQL-integrated machine learning pipelines using the ECG5000 dataset. Through careful design and implementation, we establish a pipeline that encompasses data preprocessing, model training, SQL-based filtering, and three distinct inference strategies: Baseline (per-record model loading), Rewriter (cached model reuse), and Rewriter + Batch (batch inference via vectorized operations). These strategies reflect different trade-offs between implementation complexity and runtime efficiency.

The benchmarking experiments reveal that the naive Baseline strategy suffers from substantial latency due to repeated model deserialization and execution overhead. In contrast, the

Rewriter strategy, which reuses a preloaded model for sequential record-level inference, significantly reduces per-record latency. The most efficient setup, Rewriter + Batch, achieves the lowest latency by combining model reuse with batch prediction, enabling high-throughput processing of ECG data within a single query.

These results substantiate our hypothesis: integrating machine learning inference into SQL queries, when properly optimized through model caching and batch execution, is not only feasible but also highly effective for medical time-series analytics. Such an approach mitigates the traditional impedance mismatch between database systems and machine learning pipelines, eliminates the need for data export and re-import, and supports more streamlined, declarative workflows for data scientists and healthcare practitioners.

Furthermore, our work extends the IMBridge framework by addressing domain-specific challenges in biomedical signal processing, such as preprocessing of ECG waveforms and one-dimensional time-series classification. By embedding signal-specific routines and diagnostic logic directly within SQL, we demonstrate how domain-aware inference pipelines can be constructed and executed end-to-end inside a relational engine like DuckDB.

Overall, our study contributes to the growing body of research advocating hybrid systems that unify data management and machine learning. The implications are particularly promising in healthcare, where real-time decision support, model interpretability, and system integration are critical. Future work may explore integrating more complex models (e.g., convolutional or recurrent neural networks), support for multi-label classification, dynamic windowing of signals, and GPU-accelerated inference in distributed database environments. By continuing to close the gap between AI and databases, we can unlock new capabilities for intelligent, scalable, and explainable medical analytics.

# 8. REFERENCES


[1] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2024. IMBridge: Impedance Mismatch Mitigation between Database Engine and Prediction Query Execution. In Companion of the 2024 International Conference on Management of Data (SIGMOD '24). Association for Computing Machinery, New York, NY, USA, 456–459. https://doi.org/10.1145/3626246.3654754

[2] Wagner, P., Strodthoff, N., Bousseljot, RD. *et al.* PTB-XL, a large publicly available electrocardiography dataset. *Sci Data* **7**, 154 (2020). https://doi.org/10.1038/s41597-020-0495-6 Fröhlich, B. and Plate, J. 2000. The cubic mouse: a new device for three-dimensional input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (The Hague, The Netherlands, April 01 - 06, 2000). CHI '00. ACM, New York, NY, 526-531. DOI= http://doi.acm.org/10.1145/332040.332491.

[3] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. Proc. VLDB Endow. 17, 8 (April 2024), 1830–1842. https://doi.org/10.14778/3659437.3659441 Sannella, M. J. 1994. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Doctoral Thesis. UMI Order Number: UMI Order No. GAX95-09398., University of Washington.

[4] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[5] Ng, Andrew, and Michael Jordan. "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes." *Advances in neural information processing systems* 14 (2001).